SIEMENS
NIXDORF

INFORMIX

# INFORMIX SQL-Sprachbeschreibung V6.0

(UNIX)
Einführung

Benutzerhandbuch

## Uns interessiert Ihre Meinung zu dieser Druckschrift.

Schicken Sie uns bitte eine Kopie dieser Seite, wenn Sie uns Hinweise geben wollen:

- zum Inhalt
- zur Form
- zum Produkt.

Dafür bedanken wir uns im voraus.
Mit freundlichen Grüßen,
Ihre

Siemens Nixdorf Informationssysteme AG
Unternehmenskommunikation
81730 München

Fax: (0 89) 6 36-4 97 68

## We would like to know your opinion on this publication.

Please send us a copy of this page if you have any criticism on:

- the contents
- the layout
- the product.

We would like to thank you in advance for your comments.
With kind regards,

Siemens Nixdorf Informationssysteme AG
Corporate Communication
D-81730 München

Fax: (00 49) 89 6 36-4 97 68

---

## Ihre Meinung / Your opinion:

Bestellnummer dieser Druckschrift:
U9634-J-Z265-2

Order number of this manual:

# INFORMIX SQL-Sprachbeschreibung V6.0

(UNIX)

## Einführung

Benutzerhandbuch

Ausgabe April 1994

# Wir bieten Ihnen nicht nur Druckschriften an, sondern auch Kurse...

... zu diesem Produkt
... oder zu einem anderen Thema der Informationstechnik

Unsere Training Center stehen mit ihrem Kursangebot für Sie bereit.
Besuchen Sie uns in Berlin, Essen, Frankfurt/Main oder Hamburg,
in Hannover, München, Stuttgart, Wien oder Zürich.

Informationen zu unserem Trainingsangebot erhalten Sie über:

Fax (0 89) 6 36-4 29 45

Oder schreiben Sie an:

Siemens Nixdorf Informationssysteme AG
Training Center
D-81730 München



---

Published by: Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025



Documentation Team: Robert Berry, Mary Cole, Sally Cox, Signe Haugen, Steven Klitzing, Judith Sherwood, Rob Weinberg, Chris Willis, Eileen Wollam

# Vorwort

Das Handbuch *SQL-Sprachbeschreibung, Einführung* ist eine Einführung in die INFORMIX-Implementation der Structured Query Language (SQL), mit der Datenbanken abfragen können. In Verbindung mit den beiden Begleitbänden *SQL-Sprachbeschreibung, Nachschlagen* erläutert dieses Handbuch die Erstellung, Verwaltung und Benutzung relationaler Datenbanken mit INFORMIX-Produkten.

Die in diesem Handbuch abgedruckten Beispiele beziehen sich auf verschiedene INFORMIX-Produkte und gelten für unterschiedliche Betriebssysteme und Betriebssystemversionen.

Sie benötigen die folgende Software:

- Einen **INFORMIX-OnLine Dynamic Server**- oder einen **INFORMIX-SE** Datenbankserver. Dieser Datenbankserver muß entweder lokal auf Ihrerm Rechner installiert sein oder auf einem anderen Rechner, mit dem Ihr Rechner über Netz verbunden ist.
- Entweder ein Werkzeug zur Entwicklung von INFORMIX-Anwendungen wie z. B. **INFORMIX-4GL**, ein SQL-Anwendungsprogramm wie z. B. **INFORMIX-ESQL/C**, oder das Programm **DB-Access**; dieses Programm wird als Teil Ihres Datenbankservers ausgeliefert.

  Das Werkzeug zur Anwendungsentwicklung, das SQL-Anwendungsprogramm oder **DB-Access** ermöglichen es Ihnen, Abfragen zusammenzustellen, sie an Ihren Server weiterzureichen und sich die vom Server gelieferten Ergebnisse anzusehen. Mit **DB-Access** können Sie alle in diesem Handbuch beschriebenen SQL-Anweisungen ausprobieren.

# Zusammenfassung der Kapitel

Das Handbuch *SQL-Sprachbeschreibung, Einführung* enthält die folgenden Kapitel:

- Dieses Vorwort enthält allgemeine Informationen über das Handbuch und listet weitere Nachschlagewerke auf, die Ihnen dabei helfen können, dieVerwaltung relationaler Datenbanken zu verstehen.
- Die Einleitung beschreibt die Stellung dieses Handbuchs in der Informix-Produkt- und Manualfamilie, erklärt die Verwendung dieses Handbuchs, führt in die Beispieldatenbank ein, der die Beispiele in den einzelnen Handbüchern entnommen wurden, beschreibt das Produkt **Informix Meldungen und Korrekturen** und listet die neuen Funktionen der Version 6.0 der Informix-Server-Produkte auf.

Teil1 dieses Handbuchs führt in die Verwendung der SQL ein.Wenn Sie noch nicht mit Datenbanken und SQL gearbeitet haben, sollten Siedaher zunächst die Kapitel 1 bis 7 lesen.

- Kapitel 1 "Einführung in Datenbanken und SQL" gibt Ihnen eine Übersicht über die Datenbankterminiologie und definiert Begriffe, die Ihnen in diesem Handbuch häufig begegnen.
- Kapitel 2 "Einfache SELECT-Anweisungen" erläutert, wie Sie einfache Abfragen formulieren, um den Inhalt einer Datenbank abzufragen und ausgeben zu lassen.
- Kapitel 3 "Komplexe SELECT-Anweisungen"erläutert, wie Sie komplexe Abfragen formulieren, um den Inhalt einer Datenbank abzufragen und ausgeben zu lassen.
- Kapitel 4 "Datenmanipulation" beschreibt die Anweisungen, mit denen Sie neue Daten in die Datenbank einfügen und bestehende Daten aktualisieren und löschen können. Außerdem führt dieses Kapitel in die Themen "Datenbankberechtigungen", "Datenintegrität" und Archivierung ein.
- Kapitel 5 "SQL in Programmen" erklärt, wie Sie den Datenbankserver aufrufen, Datensätze abrufen und Daten "einbetten" können.
- Kapitel 6 "Programme zur Veränderung von Daten" vermittelt vertiefte Kenntnisse über die Anweisungen INSERT, DELETE und UPDATE und es erläutert den Gebrauch dieser Anweisungen in Programmen.
- Kapitel 7 "Programmieren für Mehrneutzer-Betrieb" gibt erschöpfende Informationen zu den Themen "Konkurrierender Zzugriff", "Isolationsstufen" und "Sperren". Teil1 führt in die Verwendung der SQL ein.Wenn Sie noch nicht mit Datenbanken und SQL gearbeitet haben, sollten Siedaher zunächst die Kapitel 1 bis 7 lesen.

Teil 2 dieses Handbuchs vermittelt einen Überblick über Design und Verwaltung einer Datenbank. Wenn Sie mehr über das Entity-Relationship-Datenmodell erfahren wollen und auch darüber, wie Sie eine Datenbank erzeugen und optimieren können, sollten siedas Kapitel 8-12 lesen.

- Kapitel 8 "Aufbau eines Datenmodells" beschreibt die Komponenten eines Datenmodells und erläutert Schritt für Schritt, wie Sie eine Datenbank aufbauen können.
- Kapitel 9 "Das Datenmodell implementieren" zeigt Ihnen, wie Sie Datenbank-Datentypen festlegen und die Datenbank erzeugen können.
- Kapitel 10 "Das Modell tunen" vermittelt Ihnen Kenntisse, wie Sie eine Datenbank effizient konzipieren. Sie erfahren hier Nähreres zur Speicher- und Indexverwaltung, sowie zur Berechnung von Tabellengrößen und darüber, wie Parallelverarbeitung maximiert werden kann.
- Kapitel 11 "Zugriff auf Datenbanken regeln" erläutert, wie Datensicherheit durch die Vergabe von Datenbankberechtigungen und den Einsatz gespeicherter Prozeduren und Views gewährleistet werden kann.
- Kapitel 12 "Datenbankserver und Netzwerke" beschreibt detailliert das Thema "Netze" und zeigt, wie Sie eine Datenbank so einrichten können, daß sie über ein Netz betrieben werden kann.

Teil 3 dieses Handbuchs wendet sich an Personen, die bereits mit der SQL vertraut sind, und bietet weiterführende Informationen "für Fortgeschrittene". Hier werden die folgenden Themen behandelt: Optimierung von Abfragen, Erzeugen und Verwenden gespeicherter Prozeduren sowie Erzeugen und Verwenden von Triggern. Wenn Sie die Performance Ihrer Datenbank verbessern wollen, sollten Sie diese Kapitel lesen.

- Kapitel 13 "Datenbankserver-Abfragen optimieren" zeigt Ihnen Methoden, die Ihnen beim Optimieren von Abfragen dienlich sein können. Dazu wird Funktion und Wirkung des Optimierer erläutert und die Zeit eingegangen, die verschiedene Abfragemethoden kosten.
- Kapitel 14 "Gespeicherte Prozeduren" beschreibt, wie gespeicherte Prozeduren erzeugt und verwendet werden können. Kapitel 14 "Gespeicherte Prozeduren" beschreibt, wie gespeicherte Prozeduren erzeugt und verwendet werden können.
- Kapitel 14 "Trigger erzeugen und verwenden" beschreibt, wie Trigger erzeugt und verwendet werden können.

# Literaturhinweise

Wenn Sie bisher noch keine Erfahrung mit der Datenbank-Verwaltung haben, lesen Sie einen einleitenden Artikel wie z.B. C. J. Date's *Database: A Primer* (Addison-Wesley Publishing, 1983) lesen. Falls Sie eher technische Informationen zur Datenbank-Verwaltung benötigen, lesen Sie die folgenden Bücher, ebenfalls von C. J. Date:

- *An Introduction to Database Systems, Volume I* (Addison-Wesley Publishing, 1990)
- *An Introduction to Database Systems, Volume II* (Addison-Wesley Publishing, 1983)

Dieses Handbuch setzt voraus, daß Sie mit Ihrem Betriebssystem vertraut sind. Wenn Sie noch wenig Erfahrung mit UNIX haben, sollten Sie Ihre Betriebssystem-Dokumentation oder eine gute Einführung lesen, bevor Sie sich mit SQL beschäftigen.

Hier einige Literaturhinweise zum Betriebssystem UNIX:

- *A Practical Guide to the UNIX System*, Second Edition, by M. Sobell (Benjamin/Cummings Publishing, 1989)
- *A Practical Guide to UNIX System V* by M. Sobell (Benjamin/Cummings Publishing, 1985)
- *Introducing the UNIX System* by H. McGilton and R. Morgan (McGraw-Hill Book Company, 1983)
- *UNIX for People* by Birns, Brown, and Muster (Prentice-Hall, 1985)

Wenn Sie mehr über SQL erfahren wollen, empfehlen wir:

*Using SQL* by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

# Inhaltsverzeichnis

**Kapitel 4** **Datenmanipulation**



**Kapitel 5** **SQL in Programmen**

# Einleitung

Die strukturierte Abfragesprache SQL(Structured Query Language) ist eine dem Englischen ähnliche Sprache, mit der man relationale Datenbanken erzeugen, verwalten und benutzen kann. Die INFORMIX-Version von SQL stellt eine Erweiterung des Standards der Abfragesprache dar.

# SQL-Produkte von INFORMIX

INFORMIX bietet eine Reihe von Werkzeugen zur Anwendungsentwicklung und **SQL-APIs** (SQL Application Programming Interface, SQL-Anwendungsprogrammschnittstelle). Zu den derzeit erhältlichen Werkzeugen zur Anwendungsentwicklung gehören Produkte wie **INFORMIX-SQL**, **INFORMIX-4GL** und **Interactive Debugger**, derzeit verfügbare **SQL-APIs** sind **INFORMIX-ESQL/C** sowie **INFORMIX-ESQL/COBOL**.

INFORMIX-Produkte arbeiten zusammen mit einem der Datenbankserver **INFORMIX-OnLine Dynamic Server**- oder **INFORMIX-SE**. **DB-Access** wird als Teil jedes dieser Datenbankserver ausgeliefert.

Wenn Sie Client-Anwendungen, die mit INFORMIX-Produkten der Versionen V4.1 oder V5.0 entwickelt wurden, über ein Netz betreiben, können Sie diese Anwendungen mit **INFORMIX-NET** an das Netz anschließen.

# Produkte in diesem Handbuch

Die in diesem Handbuch enthaltenen Informationen gelten für die folgenden Produkte. Bei Bedarf werden Unterschiede beim Gebrauch von SQL aufgezeigt:

- **INFORMIX-ESQL/C**, Version 6.0
- **INFORMIX-ESQL/COBOL**, Version 6.0
- **INFORMIX-SE**, Version 6.0
- **INFORMIX-OnLine Dynamic Server**, Version 6.0
- **INFORMIX-TP/XA**, Version 6.0

Die in diesem Handbuch enthaltenen Informationen gelten ebenfalls für ein weiteres Produkt, **INFORMIX-Gateway *with DRDA***, Version 6.0.

Darüber hinaus basieren einige Beispiele in diesem Handbuch auf INFORMIX-Produkten früherer Versionen (älter V6.0).

# Weiterführende Literatur

Die *SQL-Sprachbeschreibung* besteht aus insgesamt vier Handbüchern, die sich gegenseitig ergänzen.

- Das Handbuch *SQL-Sprachbeschreibung, Einführung* führt in die SQL-Version von INFORMIX ein. Es erläutert grundlegende Konzepte und Fachbegriffe, die zur Planung und Implementierung einer relationalen Datenbank notwendig sind.
- Im Handbuch *SQL-Sprachbeschreibung, Nachschlagen* finden Sie Informationen darüber, welche Arten von Datenbanken INFORMIX zur Verfügung stellt. Es informiert über die von INFORMIX unterstützten Datentypen, über die mit den Datenbanken verknüpften Systemtabellen, über die verwendeten Umgebungsvariablen sowie über SQL-Dienstprogramme. Darüber hinaus finden Sie in diesem Handbuch eine ausführliche Beschreibung der Beispieldatenbank **stores6** und ein Glossar.
- Das Handbuch *SQL-Sprachbeschreibung, Syntax* erläutert detailliert alle unterstützten SQL-Anweisungen, sowie alle SPL-Anweisungen für gespeicherte Prozeduren.
- Das Handbuch *SQL-Sprachbeschreibung, Syntaxübersicht* enthält Syntaxdiagramme für alle Anweisungen und alle Unterdiagramme, die in diesem Handbuch detailliert beschrieben sind.
- Beachten Sie vor jeder Installation eines INFORMIX-Produkts die entsprechende *Freigabemitteilung*.
- Abhängig davon, welcher Datenbankserver verwendet wird, benötigen Sie zusätzlich das Handbuch *INFORMIX-SE Administratorhandbuch* bzw. die Handbücher *INFORMIX-OnLine Dynamic Server Administrator's Guide* und *INFORMIX-OnLine Dynamic Server Archive and Backup Guide.*.
- Auftretende Fehler können Sie unter der zugehörigen Nummer im Handbuch *Fehlermeldungen* nachschlagen. Dort finden Sie Hinweise zu Ursache und Behebung der Fehler. Sie können die Fehlermeldungen auch in der

Online-Meldungsdatei anschauen, die im Abschnitt "ASCII- und PostScript-Fehlermeldungs-Dateien" in dieser Einleitung beschrieben ist.

- Das Handbuch *DB-Access, Interaktive Datenbankverwaltung* beschreibt, wie das Programm dazu genutzt werden kann, auf Informationen der INFORMIX-Datenbankserver zuzugreifen, zu modifizieren und abzufragen.

## Bestellen von Handbüchern

Die aufgeführten Handbücher finden Sie mit ihren Bestellnummern im *Druckschriftenverzeichnis* der Siemens Nixdorf Informationssysteme AG. Neu erschienene Titel finden Sie in den *Druckschriften-Neuerscheinungen.*

Beide Veröffentlichungen erhalten Sie regelmäßig, wenn Sie in den entsprechenden Verteiler aufgenommen sind. Wenden Sie sich bitte hierfür an Ihre zuständige Geschäftsstelle. Dort können Sie auch die Handbücher bestellen.

# Hinweise zu diesem Handbuch

Dieses Handbuch geht davon aus, daß Sie den Datenbankserver **INFORMIX-OnLine Dynamic Server** verwenden. Auf Eigenschaften und Verhalten, die spezifisch für **INFORMIX-SE** sind, wird im gesamten Handbuch hingewiesen. Die folgenden Kapitelabschnitte beschreiben Konventionen bezüglich der in diesem Handbuch verwendeten Typographie, Syntax und Beispiele.

## Typographische Konventionen

In diesem Handbuch wird eine Anzahl von Konventionen verwendet, um neue Begriffe einzuführen, Bildschirmausgaben zu erläutern, die Kommandosyntax zu beschreiben und so weiter. Die folgenden typographischen Konventionen werden im gesamten Handbuch verwendet. Sie gelten ebenfalls für alle anderen INFORMIX-Handbücher:

| | |
|---|---|
| *kursiv* | Neu eingeführte Begriffe, Hervorhebungen und Variablen sind kursiv geschrieben. |
| **fett** | Namen von Datenbanken , Tabellen und Spalten, Dienstprogramme und andere ähnliche Begriffe sind fett geschrieben. |
| `maschine` | INFORMIX-Ausgaben und Eingaben, die Sie eintippen müssen, sind in Maschinenschrift geschrieben. |
| SCHLÜSSEL | Alle Schlüsselwörter werden groß geschrieben. |

Dieses Symbol kennzeichnet eine Warnung. Warnungen geben sehr wichtige Informationen, deren Nichtbeachten zur Zerstörung der Datenbank führen können.

Außerdem gilt folgende Sprachregelung:

Wenn Sie aufgefordert werden, Text "einzugeben" oder "auszuführen", müssen Sie unmittelbar im Anschluß an Ihre Eingabe die RETURN-Taste drücken. Wenn Sie den Text "schreiben" oder eine Taste "drücken" sollen, müssen Sie die RETURN-Taste nicht betätigen.

## Konventionen in Programmbeispielen

SQL-Programmbeispiele kommen durchgängig im gesamten Handbuch vor. Wenn nicht besonders vermerkt, sind diese Beispiele nicht produktspezifisch. Enthält ein Beispiel reine SQL-Anweisungen, so werden diese nicht mit Semikolon abgeschlossen. Wollen Sie ein SQL-Beispiel für ein bestimmtes Produkt benutzen, müssen Sie die Syntaxregeln dieses Produktes beachten. Verwenden Sie z. B. die Menüoption Query-Language von **DB-Access**, so müssen Sie Anweisungsfolgen mit Semikolon abschließen. Wenn Sie ein **SQL-API** verwenden, müssen Sie EXEC SQL vor und ein Semikolon (oder andere geeignete Feldbegrenzer) nach jeder Anweisung schreiben.

Betrachten Sie z.B. die folgenden Anweisungen:

```
CONNECT TO stores6
.
.
.
DELETE FROM customer
    WHERE customer_num = 121
.
.
.
COMMIT WORK
DISCONNECT CURRENT
```

Detaillierte Hinweise über den Gebrauch von SQL-Anweisungen bei einem bestimmten Werkzeug zur Anwendungsentwicklung oder einem **SQL-API** finden Sie im entsprechenden Handbuch.

Die Punkte im Beispiel zeigen an, daß Teile ausgelassen sind, die an dieser Stelle nicht von Bedeutung sind.

# Nützliche Online-Dateien

Zusätzlich zu den anderen INFORMIX-Handbüchern, ergänzen die folgenden Online-Dateien dieses Handbuch. Die Online-Dateien befinden sich im Dateiverzeichnis **$INFORMIXDIR/release**:

| | |
|---|---|
| **Documentation Notes** | beschreiben Funktionen und Eigenschaften, die nicht in den Handbüchern enthalten sind oder die sich seit Veröffentlichung geändert haben. Die Datei mit den Documentation Notes zu diesem Handbuch enthält, heißt **SQLRDOC_6.0** |
| **Release Notes** | beschreiben Unterschiede gegenüber früheren Versionen der INFORMIX-Produkte und wie sich diese Unterschiede auf die aktuellen Produkte auswirken können. Die Datei mit den Release Notes für die aktuelle Version 6.0 der INFORMIX-Server-Produkte enthält, heißt **SERVERS_6.0** |
| **Machine Notes** | beschreiben spezielle Maßnahmen, die für die Konfiguration und den Betrieb der INFORMIX-Produkte erforderlich sind. Machine Notes gibt es zu jedem beschriebenen Produkt. Die Datei mit den Machine Notes zu **INFORMIX-OnLine Dynamic Server** heißt **ONLINE_6.0** |

Schauen Sie sich diese Dateien gründlich an, denn sie enthalten wichtige Informationen zu Anwendungs- und Performance-Aspekten.

Weiterhin gibt es für eine Reihe von INFORMIX-Produkten Online Hilfedateien, die Sie durch die jeweilige Menüoption führen. Unabhängig davon, mit welchem INFORMIX-Produkt Sie arbeiten, können Sie einfach durch Drücken von CTRL-W die Hilfefunktion aufrufen.

Die PostScript-Fehlermeldungen werden in mehreren Dateien ausgeliefert, deren Namen das Format **errmsg1.ps, errmsg2.ps** usw. haben. Diese Dateien befinden sich im Dateiverzeichnis **$INFORMIXDIR/msg**.

# ASCII- und PostScript-Fehlermeldungs-Dateien

Alle INFORMIX-Produkte verwenden ASCII-Dateien, die die Fehlermeldungen und die jeweiligen Korrekturmaßnahmen enthalten. Auf diese Fehlermeldungsdateien können Sie mit Prozeduren zugreifen, die die Fehlermel-

dungen am Bildschirm anzeigen (finderr) oder formatiert ausdrucken (rofferr). Eine Erläuterung dieser Prozeduren finden Sie im Handbuch *Fehlermeldungen*.

Das Zusatzprodukt **Informix Meldungen und Korrekturen** verwendet PostScript-Dateien, die die Fehlermeldungen und die jeweiligen Korrekturmaßnahmen enthalten. Wenn Sie dieses Produkt installiert haben, können Sie die PostScript-Dateien auf einem PostScript-Drucker ausdrucken lassen.

# Die Beispieldatenbank

Das Dienstprogramm **DB-Access**, das mit Ihrem Datenbankserver mitgeliefert wurde, enthält eine Beispieldatenbank mit dem Namen **stores6**. Diese Datenbank enthält Daten über einen fiktiven Sportartikelgroßhändler. Ebenfalls enthalten sind Beispielkommandodateien, die eine Beispielanwendung erzeugen.

Die meisten Beispiele in diesem Handbuch basieren auf der Beispieldatenbank **stores6**. In Anhang A des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen* finden Sie diese Datenbank und ihren Inhalt eingehend beschrieben.

Zum Installieren der Beispieldatenbank benötigen Sie das Skript **dbaccessdemo6**, das im Dateiverzeichnis **$INFORMIXDIR/bin** abgelegt ist. Der Datenbankname, den Sie beim Aufruf eingeben, ist der Name der Beispieldatenbank. Wenn Sie keinen Datenbanknamen eingeben, ist der Name standardmäßig **stores6**. Beachten Sie bei der Vergabe des Datenbanknamens die folgenden Regeln:

- Datenbanknamen dürfen unter **INFORMIX-SE** bis zu 10 Zeichen lang sein, unter **INFORMIX-OnLine Dynamic Server** dürfen Datenbanknamen bis zu 18 Zeichen lang sein.
- Das erste Zeichen muß ein Buchstabe sein oder ein Unterstrich (_).
- Die restlichen Zeichen dürfen Buchstaben, Ziffern und Unterstriche (_) sein.
- **DB-Access** unterscheidet nicht zwischen Groß- und Kleinbuchstaben.
- Der Name einer Datenbank muß eindeutig sein.

Wenn Sie **dbaccessdemo6** ausführen, dann erzeugen Sie die Datenbank und sind deshalb Eigentümer und auch Administrator (DBA) dieser Datenbank.

Wenn Sie den **INFORMIX-SE** Datenbank-Server installiert haben, dann sind die Dateien geschützt, die die Beispieldatenbank erzeugen . Sie können keine Änderungen an der Original-Datenbank vornehmen.

Sie können das Skript **dbaccessdemo6** immer wieder dann aufrufen, wenn Sie mit der unveränderten Beispieldatenbank arbeiten wollen. Ist die Beispieldatenbank volständig erzeugt, werden Sie gefragt, ob Sie die Beispielkommandodateien in das aktuelle Dateiverzeichnis kopieren wollen. Geben Sie "N" ein, wenn Sie bereits Änderungen an den Beispielkommandodateien vorgenommen haben und nicht möchten, daß diese mit den Originalversionen überschrieben werden. Geben Sie "Y" ein, wenn Sie die veränderten Beispielkommandodateien überschreiben wollen.

# Die Beispieldatenbank erzeugen

Auf folgende Weise können Sie die Beispieldatenbank erzeugen und mit Daten füllen:

1. Geben Sie mit der Umgebungsvariablen INFORMIXDIR den Namen desjenigen Dateiverzeichnisses bekannt, in dem Ihre Informix-Produkte installiert sind. Geben Sie mit der Umgebungsvariable INFORMIXSERVER den Namen des verwendeten Servers bekannt. Dieser Name muß in der Datei **$INFORMIXDIR/etc/sqlhosts** existieren. (Zur vollständigen Beschreibung der Umgebungsvariablen siehe Kapitel 4 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen.*) Nähere Informationen über **sqlhosts**, finden Sie in den Handbüchern *INFORMIX-OnLine Dynamic Server Administrator's Guide* und *INFORMIX-SE Administrator's Guide.*
2. Erzeugen Sie ein neues Dateiverzeichnis für die SQL-Anweisungsdateien:

   ```
   mkdir dirname
   ```
3. Machen Sie das neue Dateiverzeichnis zum aktuellen Dateiverzeichnis indem Sie folgendes Kommando eingeben:

   ```
   cd dirname
   ```
4. Erzeugen Sie die Beispieldatenbank und kopieren Sie die Beispiel-Anweisungsdateien:

   ```
   dbaccessdemo6 dbname
   ```

   Wenn Sie eine Datenbank mit Protokollierung erzeugen wollen, geben Sie ein:

   ```
   dbaccessdemo6 -log dbname
   ```

   Unter **INFORMIX-OnLine Dynamic Server** werden die Daten zur Datenbank standardmäßig im Root-Dbspace abgelegt. Sie können aber auch

der Datenbank explizit einen Dbspace zuweisen. Mit dem folgenden Kommando weisen Sie einer Datenbank einen bestimmten Dbspace zu:

```
dbaccessdemo6 dbspacename
```

Unter **INFORMIX-SE** wird im aktuellen Dateiverzeichnis ein Unterverzeichnis angelegt mit dem Namen *dbname*.**dbs**. In diesem Unterverzeichnis werden die mit der Beispieldatenbank verknüpften Datendaten (.dat) ebenso abgelegt wie die Indexdateien (.idx). (Wenn Sie den Namen eines Dbspace angeben, so wird dieser ignoriert.)

Um die Kommandodateien verwenden zu können, die in Ihr Dateiverzeichnis kopiert wurden, müssen Sie die UNIX-Rechte zum Lesen und Ausführen für jedes Dateiverzeichnis haben, das im Pfadnamen enthalten ist, und zwar im Pfadnamen desjenigen Dateiverzeichnisses, von dem aus Sie die Prozedur **dbaccessdemo6** aufgerufen haben.

5. Mit dem UNIX-Kommando **chmod** können Sie einem anderen Benutzer Zugriffsrechte auf die Kommandodateien in Ihrem Dateiverzeichnis erteilen.
6. Mit den Anweisungen GRANT und REVOKE erteilen Sie anderen Personen SQL-Berechtigungen zum Zugriff auf die Daten. Die Anweisungen GRANT und REVOKE sind in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* beschrieben.

# Neue Funktionen der Informix-Produkte, Version 6.0

Dieser Abschnitt listet die wichtigsten neuen Funktionen auf, die in Version 6.0 der INFORMIX-Produkte enthalten sind, die SQL nutzen. Eine vollständige Liste aller neuen Funktionen zur Version 6.0 finden Sie in der Datei SERVERS_6.0, die die Release Notes enthält.

- Native Language Support (NLS)
- Aufgrund des Native Language Support (NLS) können Sie INFORMIX-Produkte der Version 6.0 an verschiedene europäische Kultur- und Sprachräume anpassen, ohne Ihre Anwendungen ändern zu müssen.
- Sind die entsprechenden Umgebungsvariablen so gesetzt, daß NLS aktiviert ist und einen Sprachraum bezeichnen, so kann INFORMIX die Sortiereihenfolge für Zeichenketten richtig einstellen, die landessprachliche Sonderzeichen enthalten. Auch Währungssymbole, Dezimalzeichen

sowie Datums- und Zeitangaben werden in der Form angenommen, die der eingestellte Sprachraum erfordert.

Sie können

- Datenbankinformation in jeder auf Ihrem Rechner verfügbaren Sprache erstellen oder abfragen, indem Sie die Werte einiger weniger Umgebungsvariablen ändern,
- bei der Vergabe von Namen für benutzer-definierte Objekte wie Datenbanken, Tabellen, Spalten, Views, Satzzeigern und Dateien auf landessprachliche Zeichensätze zurückgreifen,
- landessprachliche Sonderzeichen in den neuen Datentypen NCHAR bzw. NVARCHAR statt der bisherigen Datentypen CHAR bzw. VARCHAR speichern.

Darüber hinaus können Sie landessprachliche Fehlermeldungen und Warnungen ausgeben lassen, indem Sie eine oder mehrere Sprachunterstützung(en)l mit Ihrem INFORMIX-Produkt installieren.

- Erweiterte Datenbank-Verbindungen

  Mit den neuen Anweisungen CONNECT TO, DISCONNECT sowie SET CONNECTION können Sie Verbindungs-orientierte Verknüpfungen schaffen zwischen Client- und Serverprozessen - nicht nur in einer vernetzten Umgebung, sondern auch in einer nicht vernetzten Umgebung. Diese Anweisungen entsprechen X/Open- und ANSI/ISO-Normen. Sie können Anwendungen mit eingebetteter SQL mit diesen Anweisungen einheitlicher und portierbarer programmieren, gleichgültig ob die Daten auf einem lokalen oder einem fernen Rechner vorliegen.

- Kaskadisches Löschen

  Als Erweiterung der referentiellen Integrität unterstützt der Datenbankserver **INFORMIX-OnLine Dynamic Server** kaskadisches Löschen. In früheren Versionen wurde jeder Versuch, einen Satz der Elterntabelle zu löschen, abgewiesen, wenn in einer mit ihr verknüpften Kindtabelle noch Sätze existierten. Nun ist es möglich, die Option ON DELETE CASCADE in den Anweisungen CREATE TABLE und ALTER TABLE bzw. in **DB-Access**-Menüs anzugeben und so mit dem Löschen der Elterntabelle auch das Löschen der Kindtabelle(n) zu erlauben.

- Erweiterte CREATE INDEX-Anweisung

  Die Klausel FILLFACTOR erlaubt es Ihnen, mit der Anweisung CREATE INDEX kompakte Indizes zu erzeugen, oder Indizes, die Raum lassen für die Aufnahme neuer Indizes zu einem späteren Zeitpunkt.

- Neue Arithmetische Funktionen

  Es stehen Ihnen nun weitere trigonometrische und algebraische Funktionen in SQL-Anweisungen zur Verfügung: ABS, MOD, POW, ROOT, SQRT, COS, SIN, TAN, ACOS, ASIN, ATAN, ATAN2, EXP, LOGN, and LOG10.

- Verbesserte Fehlerbehandlung

  Zum Abfragen von Diagnoseinformationen über die Bearbeitung von SQL-Anweisungen steht Ihnen die neue Anweisung GET DIAGNOSTICS zur Verfügung. Diese Anweisung ist kompatibel zu X/Open- und ANSI/ISO-Bestimmungen und stellt die Standardmethode zum Abfangen und Behandeln von Fehlermeldungen dar. Dazu wurde die Fehlervariable SQLSTATE neu in den SQL-Umfang aufgenommen.

- Neue DBINFO-Funktion

  Die Funktion DBINFO erlaubt es Ihnen, die folgende Informationen abzufragen:

  - Den Dbspace-Namen für eine angegebene Tabelle
  - Den zuletzt in eine Tabelle eingetragenen SERIAL-Wert
  - Die Anzahl der von einer der folgenden Anweisungen bearbeiteten Datensätze: SELECT, INSERT, DELETE, UPDATE und EXECUTE PROCEDURE.

- Neue Umgebungsvariablen

  Die folgenden INFORMIX-Umgebungsvariablen und X/Open-Kategorien sind neu. Eine ausführliche Beschreibung diese Umgebungsvariablen finden Sie in Kapitel 4 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen*:

  - ARC_DEFAULT
  - ARC_KEYPAD
  - COLLCHAR
  - DBAPICODE
  - DBNLS
  - DBSPACETEMP
  - DBUPSPACE
  - ENVIGNORE
  - INFORMIXC
  - INFORMIXSERVER
  - INFORMIXSHMBASE
  - INFORMIXSTACKSIZE

- LANG
- LC_COLLATE
- LC_CTYPE
- LC_MONETARY
- LC_NUMERIC
- LC_TIME

- Neue Datenverteilungs-Funktionalität

  Mit der Anweisung UPDATE STATISTICS können Sie für jede Tabelle Datenverteilungen vornehmen, die zur Optimierung der Suchstrategie bei SELECT-Anweisungen verwendet werden. Diese neue Funktionalität betrifft die Entwicklung von Anwendungen auf dreierlei Arten:

  - Die Syntax der UPDATE STATISTICS-Anweisung wurde erweitert.
  - Mit der neuen Umgebungsvariable DBUPSPACE legen Sie den oberen Grenzwert für den beim Sortieren der Spalten verwendeten Speicherplatz fest.
  - Mit dem Dienstprogramm **dbschema** können Sie die Datenverteilungsinformation ausgeben lassen.

- Einführung einer Konfigurationsdatei für Umgebungsvariablen

  In den folgenden Konfigurationsdateien können Sie Voreinstellungen für Umgebungsvariablen definieren:

  - $INFORMIXDIR/etc/informix.rc
  - ~/.informix

  Diese optionalen Dateien erlauben es Ihnen, Umgebungsvariablen in einer Datei zu setzen, ähnlich wie Sie dies bereits in den Dateien **.login** bzw. **.profile** bereits tun können. Dies erspart Ihnen das Setzen der Umgebungsvariablen zu Beginn jeder Sitzung.

Kapitel

# 1

# Einführung in Datenbanken und SQL

# Kapitelüberblick

In diesem Handbuch erfahren Sie, wie Sie INFORMIX-Software zum Einsatz von Datenbanken nutzen. Der praktische Einstieg in Datenbanken beginnt mit der Beschreibung der SELECT-Anweisung in Kapitel 2 "Einfache SELECT-Anweisungen". Wenn Sie bereits einiges über Datenbanken wissen, können Sie sofort dort weiterlesen.

Im vorliegenden Kapitel werden grundlegende Konzepte von Datenbanken erläutert sowie einige Begriffe, die im gesamten Handbuch verwendet werden. Folgende Themen werden angesprochen:

- Worin unterscheidet sich eine Datenbank von einer Ansammlung von Dateien?
- Welche Begriffe werden verwendet, um die wichtigsten Bestandteile einer Datenbank zu beschreiben?
- Welche Sprache wird verwendet, um eine Datenbank zu erzeugen, abzufragen und zu ändern?
- Welches sind die wichtigsten Bestandteile der Software, die eine Datenbank verwaltet? Wie spielen diese Bestandteile zusammen?

# Datenbanken: Sinn und Zweck

Genau wie jede einfache Datei ist eine Datenbank zunächst eine *Ansammlung von Daten*. Zwei sehr wichtige Dinge aber unterscheiden Datenbanken von einfachen Dateien: Erstens enthält eine Datenbank nicht nur Daten, sondern auch einen Plan der Daten, oder besser ein *Datenmodell*. Zweitens kann eine Datenbank allgemein verfügbar sein und von vielen Benutzern zur gleichen Zeit genutzt werden.

## Das Datenmodell

Der hauptsächliche Unterschied zwischen Daten, die in einer Datenbank gehalten werden, und Daten, die in einer Datei gespeichert sind, liegt in der unterschiedlichen Organisation der Daten. Eine Datei ist physikalisch aufgebaut; einzelne Datenelemente folgen in einer bestimmten Reihenfolge aufeinander. Der Inhalt einer Datenbank dagegen beruht auf einem *Datenmodell.* Ein Datenmodell entspricht einem Plan, der die Datenelemente und die Beziehung der Datenelemente zueinander festlegt.

Eine Zahl kann z. B. sowohl in einer Datei als auch in einer Datenbank vorkommen. In einer Datei ist diese Zahl eben nur eine Zahl, die an einer bestimmten Stelle der Datei steht. In einer Datenbank jedoch wird der Zahl über das Datenmodell eine bestimmte Rolle zugewiesen. Die Zahl kann ein *Preis* sein, der zu einem *Produkt* gehört. Das Produkt kann ein *Posten* eines *Auftrags* sein, der von einem *Kunden* geordert wurde. Preise, Produkte, Positionen, Aufträge und Kunden entsprechen also Rollen, die durch das Datenmodell vorgegeben sind (Siehe Bild 1-1).

Mit der Erzeugung einer Datenbank wird auch das Datenmodell festgelegt. Anschließend werden die Datenelemente entsprechend dem Plan eingefügt, den das Modell vorgibt. Manchmal wird auch der Begriff *Schema* für *Datenmodell* verwendet.

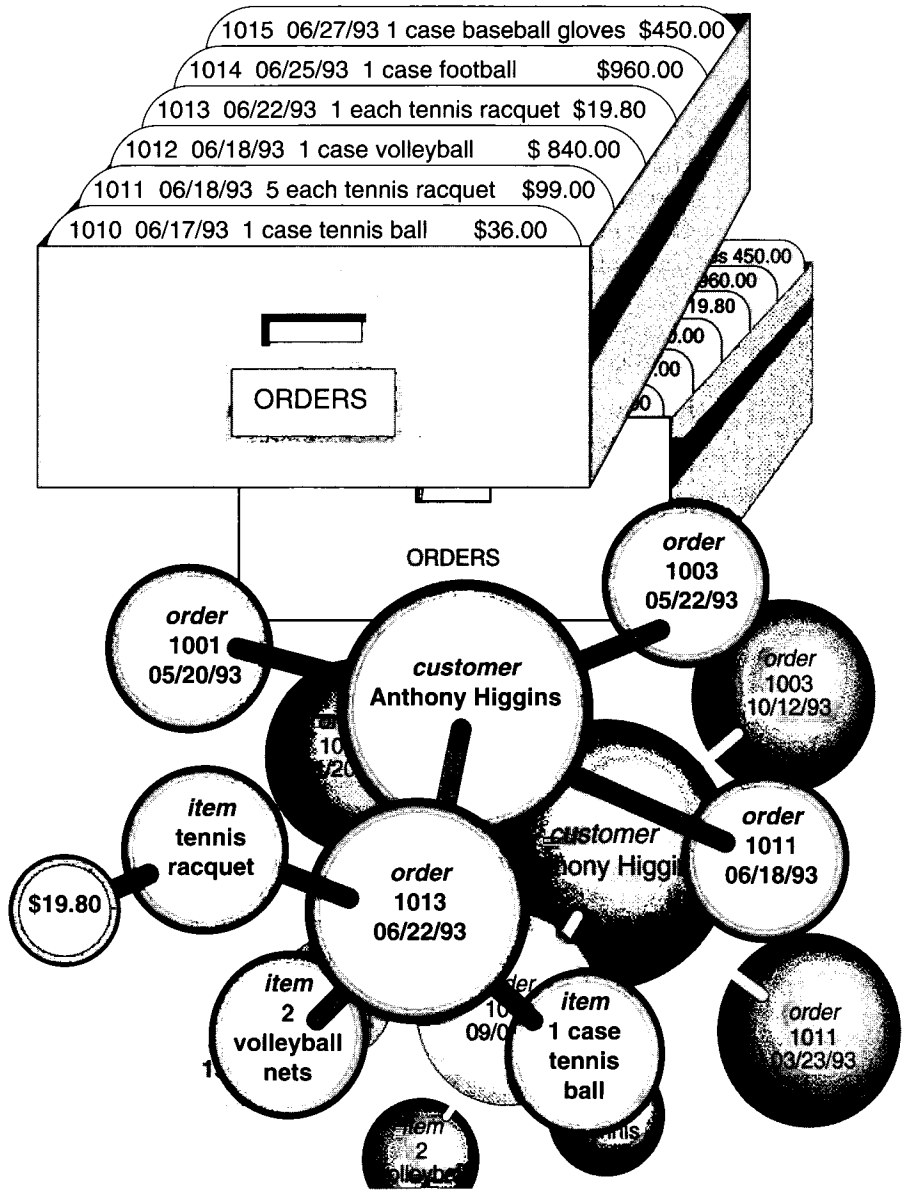



*Bild 1-1* *Darstellung eines Datenmodells*

## Datenspeicherung

Ein weiterer Unterschied zwischen einer Datenbank und einer Datei besteht darin, daß die Struktur einer Datenbank gemeinsam mit der Datenbank selbst gespeichert wird.

Eine Datei kann eine komplizierte innere Struktur haben, jedoch ist die Definition dieser Struktur nicht Bestandteil der Datei. Sie ist vielmehr in den Programmen enthalten, die die Datei erzeugen oder benutzen. Eine Dokumentdatei z. B., die von einem Textverarbeitungprogramm gespeichert wird, kann sehr detaillierte Strukturen enthalten, die das Dokument beschreiben. Jedoch nur das Textverarbeitungsprogramm selbst kann den Dateiinhalt entschlüsseln, weil die Struktur im Programm und nicht in der Datei definiert ist.

Ein Datenmodell ist jedoch in derjenigen Datenbank, die es beschreibt, auch selbst gespeichert. Das Modell verändert sich mit der Datenbank und es ist für jedes Programm verfügbar, das auf die Datenbank zugreift. Das Modell legt nicht nur die Namen der Datenelemente fest, sondern auch deren Datentyp, so daß ein Programm sich der Datenbank dahingehend anpassen kann. Ein Programm kann z. B. ermitteln, daß das Datenelement *Preis* eine 8-stellige Dezimalzahl mit 2 Nachkommastellen ist. Das Programm kann dann Speicher für eine Zahl diesen Typs anfordern. Kapitel 5 "SQL in Programmen" und Kapitel 6 "Programme zur Veränderung von Daten" zeigen, wie Programme mit Datenbanken zusammenarbeiten.

## Daten abfragen

Ein weiterer Unterschied zwischen einer Datenbank und einer Datei besteht in der Art der Abfrage. Sie können eine Datei sequentiell durchsuchen, indem Sie Satz für Satz oder Zeile für Zeile nach einem bestimmten Wert suchen. Eine solche Datenabfrage könnte lauten: "Welche Datensätze enthalten im fünften Feld eines jeden Datensatzes eine Zahl unter 20?" Bild 1-2 veranschaulicht diese Art der Suche.

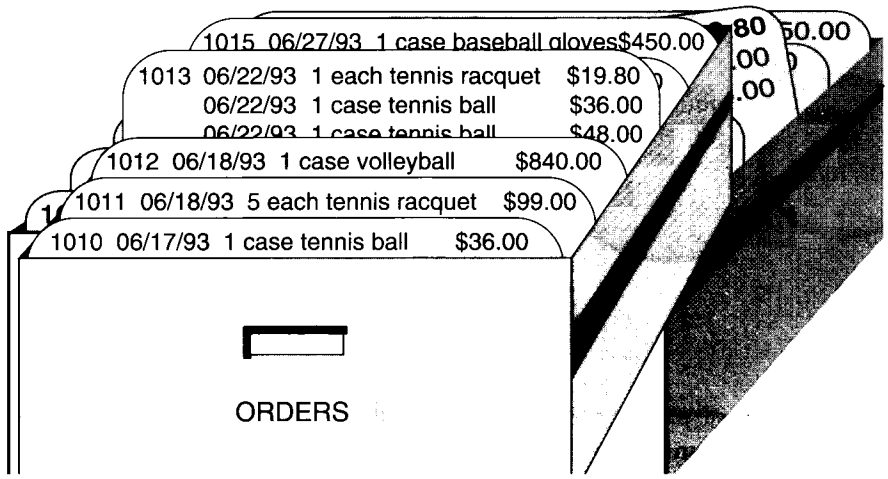



*Bild 1-2* ***Eine Datei sequentiell durchsuchen***

Im Gegensatz dazu verwenden Sie bei der Abfrage einer Datenbank die Begriffe, die über das Modell festgelegt wurden. An eine Datenbank können Sie Fragen stellen wie: "Welche *Aufträge* wurden von *Kunden* aus New Jersey für *Produkte* mit einem *Lieferdatum* im dritten Quartal erteilt, und zwar für Produkte, die von der Shimara Corporation hergestellt wurden?" Bild 1-3 verdeutlicht diese Art der Abfrage.

Anders gesagt, wenn Sie Daten abfragen, die in einer Datei gespeichert sind, müssen Sie Ihre Abfrage unter Berücksichtigung der physischen Datenanordnung formulieren. Wenn Sie eine Datenbank abfragen, können Sie die unbekannten Details der Datenspeicherung ignorieren und die Abfrage mit Begriffen stellen, die die reale Welt widerspiegeln – zumindest so weit, wie das Datenmodell selbst die reale Welt widerspiegelt.

In Kapitel 2 und 3 dieses Handbuchs wird die Sprache erörtert, die Sie für Abfragen verwenden. In den Kapiteln 8 bis 11 wird der Entwurf eines genauen, stabilen Datenmodells erläutert - eines Modells, das auch andere Benutzer zur Abfrage verwenden können.

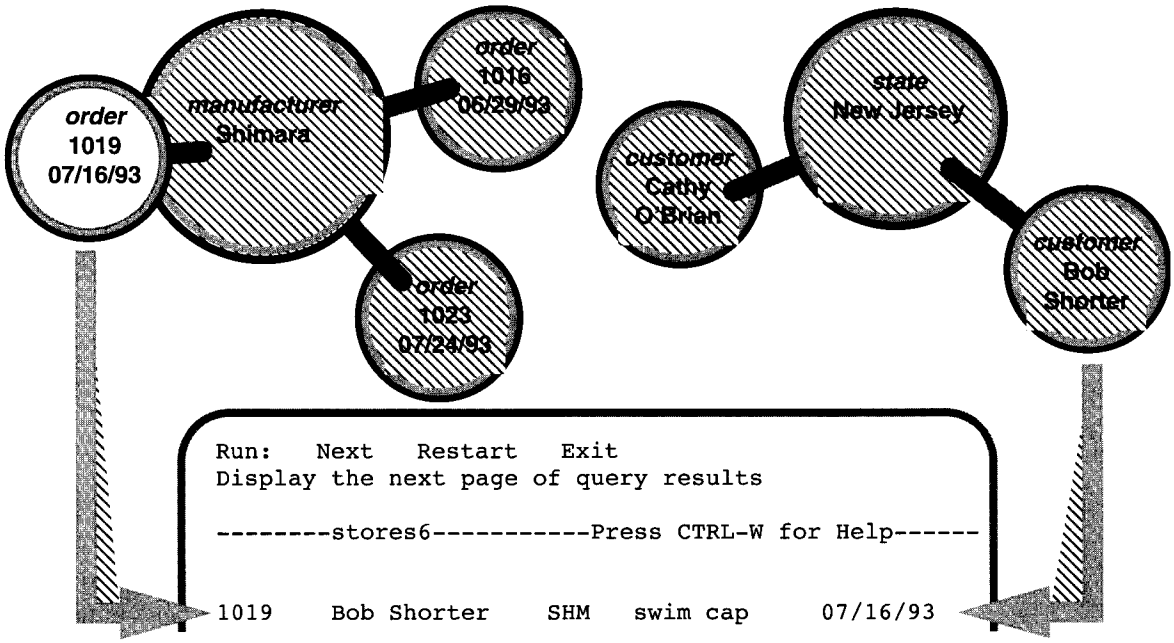



*Bild 1-3 Eine Datenbank abfragen*

## Datenänderungen

Ein Modell ermöglicht es auch, den Inhalt der Datenbank mit geringst möglicher Fehlerquote zu ändern. Sie können die Datenbank mit Kommandos abfragen wie "Suche alle *Artikel* der *Hersteller* Presta oder Schraeder, und erhöhe ihre Preise um 13%". Sie formulieren die Änderung in Begriffen, die die Bedeutung der Daten widerspiegeln. Sie müssen nicht zeitaufwendig und mühsam über die Eigenschaften der Datensatzfelder einer Datei nachdenken. Damit ist auch die Möglichkeit geringer, Fehler zu machen.

Die Anweisungen, mit denen Sie Daten ändern, finden Sie in Kapitel 5 "SQL in Programmen".

# Konkurrierender Zugriff und Datensicherheit

Eine Datenbank kann eine gemeinsame Informationsquelle für viele Benutzer sein. Mehrere Benutzer können eine Datenbank gleichzeitig abfragen und ändern. Der Datenbankserver (das Programm, das die Inhalte aller Datenbanken verwaltet) stellt sicher, daß die Abfragen und Änderungen nacheinander konfliktfrei ausgeführt werden.

Der gleichzeitige Zugriff verschiedener Benutzer auf eine Datenbank bringt große Vorteile. Es entstehen aber auch Probleme bezüglich Datensicherheit und Datenschutz.

Einige Datenbanken sind privat, d. h. sie werden für den eigenen Gebrauch erstellt. Andere Datenbanken enthalten vertrauliche Informationen, auf die nur eine ausgewählte Personengruppe zugreifen darf. Wieder andere Datenbanken erlauben allgemeinen Zugriff.

INFORMIX stellt Mittel zur Verfügung, mit denen der Zugriff auf eine Datenbank gesteuert werden kann. Bereits beim Entwurf einer Datenbank können Sie entscheiden, ob

- die Datenbank vollständig privat sein soll,
- ihr Inhalt allen oder nur bestimmten Benutzer zugänglich sein soll,
- die mögliche Auswahl der Daten für manche Benutzer eingeschränkt werden soll, so daß diese nur eine bestimmte Sicht (View) auf die Datenbank haben. (Es ist sogar möglich, unterschiedlichen Benutzergruppen auch jeweils unterschiedliche Datenausschnitte zu zeigen),
- bestimmten Benutzern die Auswahl, nicht aber die Änderung bestimmter Daten erlaubt sein soll,
- bestimmten Benutzern die Neueingabe, nicht aber die Änderung alter Daten erlaubt sein soll,
- bestimmten Benutzern die Änderung aller oder nur bestimmter Daten erlaubt sein soll,
- sichergestellt sein soll, daß hinzugefügte oder geänderte Daten dem Datenmodell entsprechen.

Nähere Informationen hierzu finden Sie in Kapitel 11 "Zugriff auf Datenbanken regeln".

# Zentralisierte Verwaltung

Datenbanken, die von vielen Personen benutzt werden, müssen besonders geschützt werden. Dies ruft zwei bedeutende Probleme hervor: Datensicherung und Pflege. Der Datenbankserver **INFORMIX-OnLine** erlaubt es Ihnen, diese Aufgaben zentralisiert durchzuführen.

Datenbanken müssen gegen Verlust oder Beschädigung abgesichert werden, z. B. gegen Fehler in Soft- und Hardware, Feuer- und Überschwemmungsgefahr und andere Naturkatastrophen. Der Verlust einer wichtigen Datenbank kann großen Schaden anrichten, der nicht nur die Kosten und Schwierigkeiten umfassen kann, die beim Rekonstruieren der Daten anfallen, sondern auch Umsatzverluste. Ein Plan zur regelmäßigen Sicherung kritischer Datenbanken kann diese möglichen Katastrophen vermeiden oder mildern helfen.

Eine große Datenbank, die von vielen Personen benutzt wird, muß gepflegt werden. Ein Datenbankverwalter muß die Systemresourcen überwachen, die Größe der Datenbank im Auge behalten, Engpäße vorhersehen und Erweiterungen vorausplanen. Benutzer werden Probleme in den Anwendungsprogrammen reklamieren; diese Probleme muß der Datenbankverwalter diagnostizieren und korrigieren. Wenn schnelle Antwortzeiten wichtig sind, muß jemand die Performance untersuchen und analysieren, um den Grund für lange Antwortzeiten herauszufinden.

## Datenbanken für einzelne Personen und Gruppen

Einige Datenbankserver wurden entworfen, um relativ kleine Datenbanken zu verwalten, die von einzelnen Personen oder nur einer kleinen Benutzergruppe genutzt werden.

Diese Datenbankserver (z. B. **INFORMIX-SE** für das Betriebssystem UNIX) speichern Datenbanken in Dateien, die vom Betriebssystem des Rechners verwaltet werden. Um diese Datenbanken zu archivieren, können Sie für die Sicherung der Dateien dieselben Prozeduren verwenden, die Sie für Dateien benutzen; d. h. die Dateien auf ein anderes Speichermedium kopieren, wenn sie gerade nicht benutzt werden. Der einzige Unterschied zu anderen Dateien besteht darin, daß nach der Archivierung einer Datenbank die zugehörige Transaktionsprotokoll-Datei zurückgesetzt werden muß. Die Verwendung von Transaktionsprotokollen wird in Kapitel 7 "Programmieren für Mehrneutzer-Betrieb" erläutert. Kapitel 10 "Das Modell tunen" geht näher auf die Archivierung ein.

Performance-Probleme treten häufig auf, weil bestimmte Abfragen zuviel Zeit benötigen. Kapitel 13 "Datenbankserver-Abfragen optimieren" erklärt, unter welchen Umständen eine Abfrage mehr oder auch weniger Zeit benö-

tigt. Nachdem Sie alle Funktionen der SELECT-Anweisung und die Alternativen zur Gestaltung einer Abfrage in Kapitel 2 "Einfache SELECT-Anweisungen" und Kapitel 3 "Komplexe SELECT-Anweisungen" kennengelernt haben, können Sie die Hinweise in Kapitel 13 nutzen, um die Performance der Abfragen zu verbessern.

## Datenbanken mit besonders wichtigen Daten

Der Datenbankserver **INFORMIX-OnLine** wurde entworfen, um große Datenbanken zu verwalten, die gleichzeitig höchste Ansprüche stellen an Zuverlässigkeit, Verfügbarkeit und Performance. Zwar unterstützt er auch Datenbanken für Einzelarbeitsplätze und Gruppen sehr gut, besonders geeignet ist er jedoch zur Verwaltung von Datenbanken, die absolut unentbehrlich sind für die Aufrechterhaltung des Betriebes.

**INFORMIX-OnLine** ermöglicht es dem Datenbankverwalter, Sicherungen während der Datenbanknutzung durchzuführen. Es können auch Differenzsicherungen durchgeführt werden (nur die veränderten Daten werden gesichert); dies ist immer dann wichtig, wenn eine Komplettsicherung viele Bänder und damit viel Zeit benötigen würde.

**INFORMIX-OnLine** verfügt über ein interaktives Überwachungsprogramm, mit dem der Datenbankverwalter die Aktivitäten im Datenbankserver überwachen kann, um festzustellen, wann Engpäße entstehen. Es verfügt auch über Dienstprogramme zur Analyse des Plattenspeichers. Außerdem stellt **INFORMIX-OnLine** die Systemtabellen **sysmaster** zur Verfügung, die Informationen über den gesamten Datenbankserver **INFORMIX-OnLine** beinhalten (der Datenbankserver betreut unter Umständen zahlreiche Datenbanken). Nähere Informationen zu den Systemtabellen **sysmaster** finden Sie im Handbuch *INFORMIX-OnLine Dynamic Server Administrator's Guide.*

Kapitel 10 "Das Modell tunen" gibt einen Überblick über die Methoden von **INFORMIX-OnLine** zur Plattenspeicherung. Vollständige Informationen über den Gebrauch und die Verwaltung von **INFORMIX-OnLine** finden Sie jedoch im Handbuch *INFORMIX-OnLine, Administratorhandbuch.*

# Wichtige Datenbank-Begriffe

Bevor Sie mit dem nächsten Kapitel beginnen, sollten Sie mit zwei Begriffskategorien vertraut sein, von denen die eine die Datenbank und das Datenmodell beschreibt, die andere Programme, die die Datenbank verwalten.

## Das relationale Modell

Informix-Datenbanken sind *relationale* Datenbanken. Dies bedeutet , daß das Datenmodell, durch das eine Datenbank gebildet wird, auf einem relationalen mathematischen Modell basiert, das von E.F. Codd entwickelt wurde. Praktisch ausgedrückt bedeutet dies, daß alle Daten in Form von *Tabellen* dargestellt sind, die aus *Zeilen* und *Spalten* bestehen.

### Tabellen

Eine Datenbank ist eine Sammlung von Daten, die in einer oder mehreren Tabellen eingeordnet sind. Eine Tabelle ist ein Gebilde von Daten*elementen*, das in Zeilen und Spalten untergliedert ist. Mit jedem INFORMIX-Produkt wird eine Beispieldatenbank mitgeliefert. Bild 1-4 zeigt ausschnittsweise eine Tabelle dieser Beispieldatenbank.

**Tabelle stock**

| stock_num | manu_code | description | unit_price | unit | unit_descr |
|---|---|---|---|---|---|
| 1 | HRO | baseball gloves | 250.00 | case | 10 gloves/case |
| 1 | HSK | baseball gloves | 800.00 | case | 10 gloves/case |
| 1 | SMT | baseball gloves | 450.00 | case | 10 gloves/case |
| 2 | HRO | baseball | 126.00 | case | 24/case |
| 3 | HSK | baseball bat | 240.00 | case | 12/case |
| 4 | HSK | football | 960.00 | case | 24/case |
| 4 | HRO | football | 480.00 | case | 24/case |
| 5 | NRG | tennis racquet | 28.00 | each | each |
| ... | .... | ..... | ... | ... | ..... |
| 313 | ANZ | swim cap | 60.00 | box | 12/box |

*Bild 1-4* ***Die Tabelle stock der bei allen Informix-Produkten mitgelieferten Beispieldatenbank***

Eine Tabelle repräsentiert all das, was über ein Objekt (Entity) bekannt ist, d. h. einen bestimmten Gegenstand, den die Datenbank beschreibt. Die Beispieltabelle **stock** repräsentiert all das, was von den Artikeln bekannt ist, die in einem Sportartikel-Laden vorrätig sind. Andere Tabellen der Beispiel-Datenbank repräsentieren Objekte wie **customer** (Kunden) und **orders** (Aufträge).

Eine Datenbank kann man sich als eine Ansammlung von Tabellen vorstellen. Eine Datenbank zu erzeugen bedeutet daher, eine Reihe von Tabellen zu erzeugen. Das Recht, Tabellen zu ändern oder abzufragen, kann von Tabelle zu Tabelle unterschiedlich vergeben werden, so daß einige Benutzer nur bestimmte Tabellen einsehen und verändern können und andere Tabellen nicht.

## Spalten

Jede Spalte einer Tabelle steht für ein charakteristisches Merkmal, das auf den Gegenstand zutrifft, um den es in der Tabelle geht. So hat die Tabelle **stock** Spalten für die folgenden Merkmale von Verkaufsposten: Lagernummern (stock numbers), Herstellercode (manufacturer codes), Beschreibungen (descriptions), Preise (prices) und Maßeinheiten (units of measure).

## Sätze

Jeder Satz einer Tabelle steht für genau eine *Ausprägung* , d. h. für ein konkretes Beispiel des Tabellengegenstandes. So steht jeder Satz der Tabelle **stock** für genau einen Artikel, der vom Sportartikel-Laden verkauft wird.

## Tabellen, Sätze und Spalten

Mit dem relationalen Datenmodell können sehr einfach Daten zusammengestellt werden, die die äußere Welt widerspiegeln, und zwar über die folgenden einfachen Beziehungen:

| | |
|---|---|
| Tabelle = Objekt | Eine Tabelle repräsentiert all das, was in der Datenbank über einen bestimmten Gegenstand (in unserem Beispiel *Sportartikel*) bekannt ist. |
| Spalte = Attribut | Eine Spalte repräsentiert ein Merkmal (in unserem Beispiel *Produktbeschreibung* oder *Preis*) eines bestimmten Gegenstandes. |
| Satz = Ausprägung | Ein Satz repräsentiert genau eine Ausprägung eines Tabellengegenstandes (beispielsweise einen bestimmten Tennisschläger). |

Es gibt einige Regeln darüber, wie man Objekte und Attribute auswählt. Dies ist aber nur dann von Bedeutung, wenn Sie eine Datenbank neu entwerfen. (Das Entwerfen von Datenbanken wird in den Kapiteln 8 bis 11 dieses Handbuchs behandelt). Bei einer vorhandenen Datenbank ist das Datenmodell bereits vorgegeben. Um die Datenbank benutzen zu können, müssen Sie nur die Namen der Tabellen und Spalten kennen und wissen, wie diese mit der realen Welt übereinstimmen.

## Operationen mit Tabellen

Da eine Datenbank letztlich eine Ansammlung von Tabellen ist, sind Datenbankoperationen Operationen mit Tabellen. Das relationale Modell unterstützt drei grundlegende Operationen; zwei dieser Operationen sind in Bild 1-5 dargestellt. (Die Kapitel 2 "Einfache SELECT-Anweisungen" und Kapitel 3 "Komplexe SELECT-Anweisungen" erläutern eingehend und mit vielen Beispielen alle drei Operationen.)

Aus einer Tabelle *selektieren* bedeutet, bestimmte Sätze auszuwählen und andere nicht. Eine mögliche Selektion aus der Tabelle **stock** könnte lauten "Selektiere alle Sätze, bei denen der Hersteller (manufacturer code) HRO ist und der Preis pro Einheit (unit_price) zwischen 100.00 und 200.00 liegt."

Aus einer Tabelle *projizieren* bedeutet, bestimmte Spalten auszuwählen und andere nicht. Eine mögliche Projektion der Tabelle **stock** könnte lauten: "Zeige nur die Spalten **stock_num**, **unit_descr** und **unit_price**."

Eine Tabelle enthält Daten nur über je ein Objekt; wenn Sie Daten über mehrere Objekte auswählen wollen, müssen Sie die entsprechenden Tabellen *verbinden* (*join*). Es gibt viele verschiedene Arten, Tabellen zu verbinden.Die Join-Operation wird in Kapitel 3 "Komplexe SELECT-Anweisungen" behandelt.

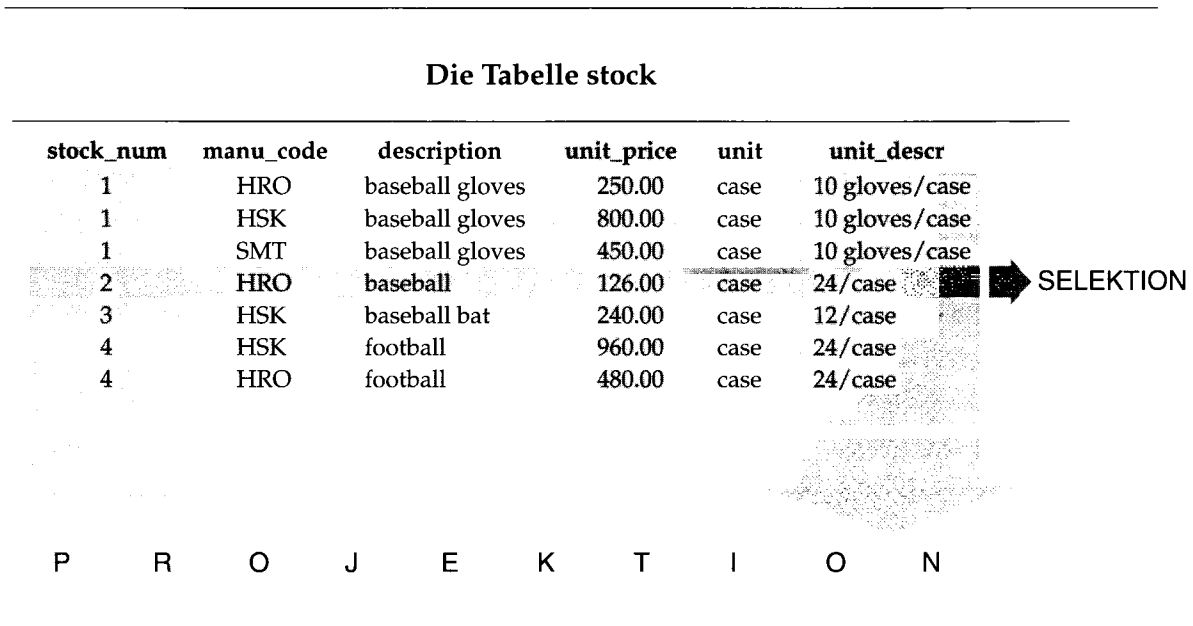



*Bild 1-5 Veranschaulichung von Selektion und Projektion*

# Strukturierte Abfragesprache

Die Entwicklung von Software hat bislang noch nicht den Stand erreicht, bei dem Sie eine Datenbank wörtlich fragen können "Welche Aufträge (orders) sind von Kunden (customers) aus New Jersey erteilt worden, mit Lieferdatum (ship_date) im zweiten Quartal?". Sie müssen Abfragen immer noch in einer eingeschränkten Syntax formulieren, die von der Software analysiert werden kann. Die obengenannte Frage können Sie in folgender Notation an die Beispieldatenbank stellen:

```
SELECT * FROM customer, orders
       WHERE customer.customer_num = orders.customer_num
              AND customer.state = NJ''
              AND orders.ship_date
              BETWEEN DATE(''7/1/93'') AND DATE(''7/30/93'')
```

Diese Abfrage ist ein Beispiel für die strukturierte Abfragesprache SQL. SQL verwenden Sie zur Ausführung aller Datenbankoperationen. Sie besteht aus Anweisungen, die alle mit einem oder zwei Schlüsselwörtern beginnen. Die Schlüsselwörter geben bestimmte Funktionen an. SQL verfügt über 73 Anweisungen, von ALLOCATE DESCRIPTOR bis WHENEVER.

Die SQL-Anweisungen sind vollständig im Handbuch *SQL-Sprachbeschreibung, Syntax* beschrieben. Die meisten Anweisungen werden nur selten benutzt, z. B. beim Einrichten oder Verbessern der Datenbank. In der Regel werden drei oder vier Anweisungen verwendet.

Nur die Anweisung SELECT wird fast ständig verwendet. Es ist die einzige Anweisung, mit der Sie Daten aus der Datenbank abfragen können. Es ist aber auch die komplizierteste Anweisung. Ein Großteil der folgenden zwei Kapitel wird darauf verwendet, die vielen Anwendungsmöglichkeiten dieser Anweisung zu erläutern.

## Standard-SQL

SQL und das relationale Modell wurde Anfang bis Mitte der 70er Jahre entwickelt. Die einzelnen SQL-Implementationen unterscheiden sich leicht voneinander, teils aus Performance- oder Wettbewerbsgründen, teils um bestimmte Funktionen in Hard- und Software zu nutzen. Um sicherzustellen, daß die Unterschiede gering bleiben, wurde in den frühen 80er Jahren ein Standardisierungskomitee gegründet.

Das Komitee X3H2, das vom American National Standards Institute (ANSI), unterstützt wird, gab 1986 den SQL1 Standard heraus. Dieser Standard definiert den Kernumfang der SQL-Funktionen und die Syntax von Anweisungen wie z. B. SELECT.

## Informix-SQL und ANSI-SQL

Die von Informix-Produkten unterstützte SQL-Version ist weitgehend kompatibel mit Standard-SQL. Gegenüber dem Standard enthält diese Version jedoch *Erweiterungen*, d. h. zusätzliche Optionen und Funktionen für bestimmte Anweisungen. Die meisten Unterschiede treten in den weniger alltäglichen Anweisungen auf. Wenige Unterschiede gibt es z. B. in der SELECT-Anweisung, die 90% des SQL-Bedarfs einer typischen Anwendung ausmacht.

Die Erweiterungen sind jedoch vorhanden, und dies schafft einen Konflikt. Tausende von INFORMIX-Kunden haben SQL in der INFORMIX-Variante in Programme und gespeicherte Abfragen eingebettet. Sie verlassen sich darauf, daß INFORMIX die Sprache unverändert läßt. Andere Kunden verlangen die Möglichkeit, Datenbanken in exakter Übereinstimmung mit dem ANSI-Standard verwenden zu können. Sie verlassen sich darauf, daß INFORMIX bei den Änderungen der Sprache konform zum Standard bleibt.

INFORMIX löste den Konflikt mit folgendem Kompromiß:

- Die INFORMIX-Version von SQL, mit seinen Erweiterungen zum Standard, bleibt verfügbar.
- Jeder INFORMIX-Sprachprozessor ist in der Lage, Anwendungen auf die Verwendung des ANSI-Standards zu prüfen und jedesmal Warnungen auszugeben, wenn INFORMIX-Erweiterungen benutzt werden.

Diese Lösung ist angemessen, aber sie macht die SQL-Dokumentation komplizierter. Wo immer ein Unterschied zwischen INFORMIX- und ANSI-SQL besteht, beschreibt das Handbuch *SQL-Sprachbeschreibung, Syntax* beide Versionen. Da Sie wahrscheinlich nur eine Version verwenden werden, können Sie die nicht benötigte Version ignorieren.

## ANSI-konforme Datenbanken

Indem Sie bei der Erstellung einer Datenbank die Schlüsselworte MODE ANSI verwenden, können Sie eine Datenbank ANSI-konform entwerfen. In einer solchen Datenbank kommen bestimmte Eigenschaften des ANSI-Standards zur Wirkung. Beispielsweise finden alle Aktionen, die Daten automatisch verändern, im Rahmen einer Transaktion statt. Dies bedeutet, daß die Änderungen entweder vollständig oder überhaupt nicht durchgeführt werden. Das Handbuch *SQL-Sprachbeschreibung, Syntax* weist an den entsprechenden Stellen auf abweichendes Verhalten ANSI-konformer Datenbanken hin.

## NLS-Datenbanken

Die Version 6.0 der Informix Server-Produkte bieten Native Language Support (NLS) an. Damit haben Sie die Möglichkeit, in zahlreichen europäischen und lateinamerikanischen Sprachumgebungen zu arbeiten und Ihre Datenbank an nationale Erfordernisse (*Locale*) anzupassen (beispielsweise in Bezug auf Währungs- und Datumsformate, Verwendung nationaler Zeichensätze in Daten und Feldbezeichnungen, abweichende Sortierfolgen etc.). Die NLS-Funktionalität aktivieren Sie, indem Sie die DBNLS-Umgebungsvariable sowie einige andere Umgebungsvariablen setzen.

In Kapitel 4 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen* erfahren Sie, wie Sie diese Umgebungsvariablen setzen. Weitere Informationen zum Thema NLS-Datenbanken finden Sie in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen*.

# Die Datenbanksoftware

Sie können auf Ihre Datenbank über zweier Schichten komplexer Software zugreifen. Die obere Schicht, auch *Anwendung* genannt, sendet Anweisungen oder Abfragen an die untere Schicht, den *Datenbankserver*. Dieser liefert die gesuchte Information zurück. Mit Hilfe der SQL steuern Sie beide Schichten.

## Die Anwendungen

Eine *Datenbankanwendung*, oder einfach nur *Anwendung*, ist ein Programm, das auf eine Datenbank zugreift, indem es den Datenbankserver aufruft. Im einfachsten Fall sendet die Anwendung SQL-Anweisungen an den Datenbankserver, und dieser schickt Datensätze zur Anwendung zurück. Dann zeigt die Anwendung Ihnen die Daten an.

Alternativ hierzu können Sie die Anwendung anweisen, neue Daten in die Datenbank aufzunehmen. Die Anwendung betrachtet die neuen Daten als Teil einer SQL-Anweisung, mit dem ein Satz eingefügt werden soll, und übergibt diese Anweisung dem Datenbankserver zur Ausführung.

Es gibt unterschiedliche Arten von Anwendungen. Einige von ihnen erlauben interaktiven Zugriff auf eine Datenbank mit Hilfe von SQL, andere zeigen die gespeicherten Daten je nach Bedarf in anderer Form.

## Der Datenbankserver

Der *Datenbankserver* ist ein Programm, das die Inhalte der Datenbank auf der Festplatte verwaltet. Er weiß, wie die Tabellen, Sätze und Spalten aktuell im physikalischen Speicher organisiert sind, er interpretiert alle SQL-Kommandos und führt sie aus.

## Interaktives SQL

Um die Beispiele in diesem Handbuch auszuführen, und um selbst mit SQL und Datenbankgestaltung experimentieren zu können, brauchen Sie ein Programm, mit dem Sie SQL-Anweisungen interaktiv ausführen können. **DB-Access** und **INFORMIX-SQL** sind zwei dieser Programme. Sie helfen Ihnen dabei, SQL-Anweisungen zu erstellen. Anschließend leiten sie Ihre SQL-Anweisungen an den Datenbankserver zur Ausführung weiter und geben die Ergebnisse auf dem Bildschirm aus.

Alternativ hierzu können Sie auf einer Workstation oder einem PC das Tabellenkalkulationsprogramm **Wingz** verwenden. **Wingz** ist in der Lage, über eine optionale **DataLink**-Erweiterung mit einem Datenbankserver zu kommunizieren. Mit der **Wingz**-Sprache **HyperScript** können Sie SQL-Anweisungen schreiben, die Datensätze zur Anzeige im Spreadsheet liefern.

## Listen und Bildschirmmasken

Nachdem Sie die Abfrage so gestaltet haben, daß sie genau die gewünschte Datenmenge liefert, müssen Sie die Daten zur Anzeige in einer Liste oder auf dem Bildschirm formatieren. Der Listengenerator für **INFORMIX-SQL** ist **ACE**. Sie übergeben an den Listengenerator eine SELECT-Anweisung, die genau die gewünschten Daten liefert sowie Anweisungen zur Gestaltung des Layouts. **ACE** compiliert aus diesen Informationen ein Programm, das Sie immer ablaufen lassen können, wenn Sie diese Liste erzeugen wollen.

**PERFORM** ist ein **INFORMIX-SQL**-Modul zur Generierung interaktiver Bildschirmmasken. Sie erstellen ein Maskenprogramm, in dem Sie die Verbindungen zwischen Bildschirmfeldern und Tabellenspalten der Datenbank angeben. **PERFORM** compiliert aus diesen Informationen ein Programm, das Sie jederzeit ablaufen lassen können. Bei der Ausführung befragt das Maskenprogramm die Datenbank und gibt den gefundenen Datensatz bzw. die gefundenen Datensätze am Bildschirm aus. Die Ausgabe erfolgt gemäß Ihren Festlegungen. Der Benutzer kann eine Maske auch für eine beispielorientierte Abfrage verwenden (*query by example*). Er kann Suchmuster eingeben, die eine bestimmte Menge von Sätzen zurückliefern.

Weitere Informationen zu diesen Produkten entnehmen Sie bitte den entsprechenden Handbüchern.

# Allgemeines zur Programmierung

Programme können SQL-Anweisungen enthalten und den Datenaustausch mit dem Datenbankserver durchführen. Ein solches Programm entnimmt die Daten einer Datenbank und gibt sie auf beliebige Weise formatiert aus. Andere Programme entnehmen die Daten aus beliebigen Quellen in beliebigem Format, bereiten diese auf und fügen sie in eine Datenbank ein.

Die für diese Zwecke komfortabelste Programmiersprache ist **INFORMIX-4GL**, eine nichtprozedurale Sprache, die insbesondere für das Schreiben von Datenbankanwendungen entworfen wurde. Die Kommunikation mit einem INFORMIX-Datenbankserver ist aber auch mit Programmen möglich, die in C, COBOL, Ada und FORTRAN geschrieben wurden und „eingebettetes" SQL enthalten.

Um mit den Datenbankdaten und -objekten zu arbeiten, kann man auch Programme schreiben, die als Prozeduren gespeichert werden (stored procedures). Diese selbstgeschriebenen Prozeduren werden direkt in einer Tabelle gespeichert. Man kann eine gespeicherte Prozedur mit **DB-Access** ausführen oder mit einem **SQL-API**.

Kapitel 5 "SQL in Programmen" und Kapitel 6 "Programme zur Veränderung von Daten" geben einen Überblick über die Verwendung von SQL in Programmen.

## Anwendungen und Datenbankserver

Jedes Programm, das mit Daten einer Datenbank arbeitet, geht auf die gleiche Weise vor. Unabhängig davon, ob es sich um ein aus verschiedenen Einzelprogrammen bestehendes Gesamtprogramm wie z. B. **INFORMIX-SQL** handelt, um ein Listenprogramm, das von **ACE** compiliert wurde, oder um ein Benutzerprogramm, das unter Verwendung von **INFORMIX-4GL** oder einem **SQL-API** geschrieben wurde: Sie werden immer auf dieselben beiden Schichten stoßen:

1. Eine Anwendung, mit der der Benutzer interaktiv im Dialog steht. Diese bereitet die Daten auf, formatiert sie und stellt die SQL-Anweisungen zusammen.
2. Einen Datenbankserver, der die Datenbank verwaltet und SQL interpretiert.

Alle Anwendungen kommen beim Datenbankserver zusammen, jedoch nur der Datenbankserver verändert die Datenbank-Dateien auf der Platte.

# Zusammenfassung

Eine Datenbank enthält eine Sammlung zusammengehöriger Daten. Die Datenbank enthält jedoch nicht nur die Daten, sondern auch ein Datenmodell, das einzelne Datenelemente definiert und seine Verbindungen zu anderen Datenelementen und zur realen Welt festlegt.

Eine Datenbank kann von mehreren Benutzern, die parallel arbeiten, benutzt und verändert werden. Unterschiedlichen Benutzern können unterschiedliche Sichtweisen (Views) auf die Datenbank zugeordnet werden. Ihre Zugriffe auf die Inhalte der Datenbank können auf mehrere Arten eingeschränkt werden.

Eine Datenbank kann entscheidende Bedeutung für den Erfolg eines Betriebs haben und kann eine zentrale Verwaltung und Überwachung erfordern. Die Datenbankserver **INFORMIX-OnLine Dynamic Server<Default ¶ Fo>** und **INFORMIX-SE** unterstützen kleinere Datenbanken für den Einzelplatzbetrieb oder für Gruppen; für besonders umfangreiche Anwendungen erfüllt der **INFORMIX-OnLine Dynamic Server<Default ¶ Fo>**- Datenbankserver höchste Anforderungen.

Mit der strukturierten Abfragesprache SQL manipulieren und befragen Sie eine Datenbank. Wahrscheinlich werden Sie die Vorteile der Sprach-Erweiterungen gegenüber ANSI nutzen wollen, aber die INFORMIX-Werkzeuge ermöglichen auch die strikte Einhaltung des ANSI-Standards.

Zwei Softwareschichten fungieren als Bindeglieder zu den Datenbanken. Die untere Schicht ist immer ein Datenbankserver, der SQL-Anweisungen ausführt und die Daten auf der Platte und im Arbeitsspeicher verwaltet. Die obere Schicht ist die Anwendungsschicht.

Kapitel

# 2 Einfache SELECT-Anweisungen

# Kapitelüberblick

SELECT ist die wichtigste und komplizierteste SQL-Anweisung. Sie können sie auf folgende Arten verwenden:

- Alleinstehend, um Daten aus einer Datenbank abzufragen
- Als Bestandteil einer INSERT-Anweisung, um neue Sätze zu erstellen
- Als Bestandteil einer UPDATE-Anweisung, um Daten zu ändern.

Die SELECT-Anweisung ist die am häufigsten genutzte Art, Daten aus einer Datenbank abzufragen. Sie erschließt Ihnen die Möglichkeit, Daten in einem Programm, einer Liste, einer Bildschirmmaske oder einer Tabelle (wie bei einem Tabellenkalkulations-Programm) abzufragen.

Dieses Kapitel zeigt, wie Sie die SELECT-Anweisung verwenden, um Daten auf verschiedene Arten aus einer Datenbank abzufragen. Es erläutert, wie Sie Anweisungen gestalten, um Spalten oder Datensätze aus einer oder mehreren Tabellen auszuwählen, wie Ausdrücke und Funktionen in SELECT-Anweisungen eingeschlossen und wie verschiedene Join-Bedingungen zwischen Datenbank-Tabellen erzeugt werden.

In diesem Kapitel werden die grundlegenden Methoden vorgestellt, wie Sie Daten aus einer relationalen Datenbank abfragen. Komplexere SELECT-Anweisungen wie *Unterabfragen*, *Outer Joins* oder *Unions* werden in Kapitel 3 dieses Handbuchs vorgestellt. Eine ausführliche Beschreibung der SELECT-Anweisung finden Sie in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

Die meisten Beispiele in diesem Kapitel stammen aus den neun Tabellen der Beispieldatenbank **stores6**, die zusammen mit der Software Ihrer INFORMIX-Software installiert wird. Aus Platzgründen wird immer nur ein Teil des Ergebnisses gezeigt, das die jeweilige SELECT-Anweisung zurückliefert. Anhang A des Manuals *SQL-Sprachbeschreibung, Nachschlagen* informiert Sie ausführlich über Aufbau und Inhalt der Beispieldatenbank **stores6**. Aus Gründen der besseren Lesbarkeit werden in den Beispielen Schlüsselwörter in Großbuchstaben dargestellt. SQL macht jedoch keinen Unterschied zwischen Groß- und Kleinschreibung.

# Einführung in die SELECT-Anweisung

Die SELECT-Anweisung besteht aus Klauseln, die Ihnen einen Blick auf Daten der relationalen Datenbank gewähren. Diese Klauseln ermöglichen es Ihnen, Spalten oder Sätze aus einer oder mehreren Datenbanktabellen oder Views auszuwählen, eine oder mehrere Bedingungen anzugeben, Daten zusammenzufassen und zu ordnen sowie die ausgewählten Daten in einer temporären Tabelle zu speichern.

Die fünf Klauseln der SELECT-Anweisung,

die in diesem Kapitel beschrieben werden, müssen in folgender Reihenfolge eingegeben werden:

1. SELECT-Klausel
2. FROM-Klausel
3. WHERE-Klausel
4. ORDER BY-Klausel
5. INTO TEMP-Klausel

Nur die SELECT- und FROM-Klausel sind zwingend. Diese beiden Klauseln bilden die Basis jeder Datenbankabfrage, da sie die Tabellen und Spalten benennen, die abgerufen werden.

- Fügen Sie die WHERE-Klausel hinzu, um bestimmte Datensätze auszuwählen oder um eine *Join*-Bedingung anzugeben.
- Fügen Sie die ORDER BY-Klausel hinzu, um die Reihenfolge zu ändern, in der die Daten ausgegeben werden.
- Fügen Sie die INTO TEMP-Klausel hinzu, um das Ergebnis als eine Tabelle für weitere Abfragen zu sichern.

Zwei zusätzliche Klauseln der SELECT-Anweisung, GROUP BY und HAVING, ermöglichen es Ihnen, kompliziertere Datenabfragen auszuführen. Diese Anweisungen werden in Kapitel 3 "Komplexe SELECT-Anweisungen"vorgestellt. Eine weitere Klausel, INTO, wird verwendet, um Programm- oder Host-Variablen anzugeben. Diese Variablen nehmen Daten einer SELECT-Anweisung bei den Produkten **INFORMIX-4GL** und **INFORMIX-ESQL** auf. Die vollständige Syntax und Regeln für die Verwendung der SELECT-Anweisung ist im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* beschrieben.

# Einige grundlegende Konzepte

Die SELECT-Anweisung verändert keine Daten in der Datenbank, ganz im Gegensatz zu den Anweisungen INSERT, UPDATE und DELETE. Sie wird nur verwendet, um Daten abzufragen. Während nur je ein Benutzer zur gleichen Zeit Daten ändern kann, können beliebig viele Benutzer parallel Daten abfragen und auswählen. Die Anweisungen zur Datenänderung werden in Kapitel 4 "Datenmanipulation" erläutert. Die Anweisungen INSERT, UPDATE und DELETE sind im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* beschrieben.

In einer relationalen Datenbank ist eine Spalte ein Datenelement, das eine spezifische Information aufnimmt, die in jedem Datensatz (auch *kurz Satz* genannt) der Tabelle vorkommt. Ein Datensatz ist eine Gruppe von zusammengehörigen Informationen über einen bestimmten Gegenstand. Der Datensatz umfaßt alle Spalten der Tabelle.

Sie können sich einzelne Spalten und Sätze einer Tabelle ausgeben lassen; entweder aus einer Systemtabelle, einer Datei, die Informationen über die Datenbank enthält; oder mit Hilfe einer *View*, also einer virtuellen Tabelle, die nur ganz bestimmte, benutzerspezifische Daten aus einer oder mehreren real existierenden Tabellen anzeigt. Systemtabellen werden im Kapitel 2 des Handbuch *SQL-Sprachbeschreibung, Nachschlagen* besprochen. Views werden im Kapitel 1 des Manuals *SQL-Sprachbeschreibung, Syntax* vorgestellt.

## Berechtigungen

Bevor Sie Daten abfragen können, müssen Sie die CONNECT-Berechtigung für die Datenbank und die SELECT-Berechtigung für die Tabellen der Datenbank haben. Diese Berechtigungen werden normalerweise allen Benutzern erteilt. Erläuterungen zu Datenbank-Berechtigungen finden Sie in Kapitel 11 "Zugriff auf Datenbanken regeln" und in den Beschreibungen der Anweisungen GRANT und REVOKE in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

## Relationale Operationen

Eine *relationale Operation* ist eine Bearbeitung einer oder mehrerer Tabellen oder *Relationen*, bei der eine weitere Tabelle entsteht. Es gibt drei Arten relationaler Operationen: Selektion, Projektion und Join (Verknüpfung). Dieses Kapitel zeigt Beispiele für diese Operationen.

## Selektion und Projektion

In der relationalen Terminologie ist die *Selektion* definiert als Entnahme einer *horizontalen* Teilmenge von Datensätzen aus einer einzelnen Tabelle, die eine bestimmte Bedingung erfüllen. Diese Art der SELECT-Anweisung liefert einige der Datensätze und alle Spalten einer Tabelle. Die Selektion wird mit der WHERE-Klausel einer SELECT-Anweisung ausgeführt:

```
SELECT * FROM customer
    WHERE state = 'NJ'
```

*Abfrage 2-1*

Das Ergebnis dieser Abfrage enthält die gleiche Anzahl an Spalten wie die Tabelle **customer**, aber nur eine Teilmenge ihrer Datensätze. (Da die Daten der ausgewählten Spalten nicht in eine Zeile von **DB-Access** bzw. des interaktiven Bildschirmeditors passen, werden die Daten vertikal statt horizontal angezeigt.)

```
customer_num  119
fname         Bob
lname         Shorter
company       The Triathletes Club
address1      2405 Kings Highway
address2
city          Cherry Hill
state         NJ
zipcode       08002
phone         609-663-6079

customer_num  122
fname         Cathy
lname         O'Brian
company       The Sporting Life
address1      543 Nassau Street
address2
city          Princeton
state         NJ
zipcode       08540
phone         609-342-0054
```

*Ergebnis 2-1*

In der relationalen Terminologie ist die *Projektion* definiert als Entnahme einer *vertikalen* Teilmenge aus den Spalten einer einzelnen Tabelle, die eindeutige Datensätze enthält. Diese Art der SELECT-Anweisung liefert einige

der Spalten und alle Datensätze einer Tabelle. Die Projektion wird entsprechend der Auswahlliste ausgeführt, die in der SELECT-Klausel einer SELECT-Anweisung angegeben ist (siehe Abfrage 2-2):

```
SELECT UNIQUE city, state, zipcode
    FROM customer
```

*Abfrage 2-2*

Das Ergebnis dieser Abfrage enthält die gleiche Anzahl Datensätze wie die Tabelle **customer**, aber es *projiziert* nur eine Teilmenge ihrer Spalten:

```
city            state zipcode

Bartlesville    OK    74006
Blue Island     NY    60406
Brighton        MA    02135
Cherry Hill     NJ    08002
Denver          CO    80219
Jacksonville    FL    32256
Los Altos       CA    94022
Menlo Park      CA    94025
Mountain View   CA    94040
Mountain View   CA    94063
Oakland         CA    94609
Palo Alto       CA    94303
Palo Alto       CA    94304
Phoenix         AZ    85008
Phoenix         AZ    85016
Princeton       NJ    08540
Redwood City    CA    94026
Redwood City    CA    94062
Redwood City    CA    94063
San Francisco   CA    94117
Sunnyvale       CA    94085
Sunnyvale       CA    94086
Wilmington      DE    19898
```

*Ergebnis 2-2*

Die häufigste Art der SELECT-Anweisung verwendet sowohl die Selektion als auch die Projektion. Eine solche Abfrage liefert, wie unten gezeigt, einige der Datensätze und einige der Spalten einer Tabelle:

```
SELECT UNIQUE city, state, zipcode
    FROM customerkunde
    WHERE state = 'NJ'
```

*Abfrage 2-3*

Das Ergebnis dieser Abfrage enthält eine Teilmenge der Datensätze und eine Teilmenge der Spalten der Tabelle **customer**:

```
city            state zipcode

Cherry Hill     NJ    08002
Princeton       NJ    08540
```

*Ergebnis 2-3*

## Joining

Ein Join ist eine Verknüpfung von zwei oder mehreren Tabellen über eine oder mehrere gemeinsame Spalten, wobei eine neue Ergebnistabelle erstellt wird. Bild 2-1 zeigt einen Teilbereich der Tabellen **items** und **stock**, um das Konzept des Joins darzustellen.

```
SELECT unique item_num, order_num, stock_num, description
    FROM items, stock
    WHERE items.stock_num = stock.stock_num
```

Tabelle **items** (Beispiel)

| item_num | order_num | stock_num |
|---|---|---|
| 1 | 1001 | 1 |
| 1 | 1002 | 4 |
| 2 | 1002 | 3 |
| 3 | 1003 | 5 |
| 1 | 1005 | 5 |

Tabelle **stock** (Beispiel)

| stock_num | manu_code | description |
|---|---|---|
| 1 | HRO | baseball gloves |
| 1 | HSK | baseball gloves |
| 2 | HRO | baseball |
| 4 | HSK | football |
| 5 | NRG | tennis racquet |

| item_num | order_num | stock_num | description |
|---|---|---|---|
| 1 | 1001 | 1 | baseball gloves |
| 1 | 1002 | 4 | football |
| 3 | 1003 | 5 | tennis racquet |
| 1 | 1005 | 5 | tennis racquet |

*Bild 2-1 Beispiel eines Joins zweier Tabellen*

Die folgende SELECT-Anweisung verknüpft die Tabellen **customer** und **state**:

```
SELECT UNIQUE city, state, zipcode, sname
    FROM customer, state
    WHERE customer.state = state.code
```

*Abfrage 2-4*

Das Ergebnis dieser Abfrage setzt sich aus bestimmten Datensätzen und Spalten der beiden Tabellen **customer** und **state** zusammen:.

```
city            state zipcode sname

Bartlesville    OK    74006   Oklahoma
Blue Island     NY    60406   New York
Brighton        MA    02135   Massachusetts
Cherry Hill     NJ    08002   New Jersey
Denver          CO    80219   Colorado
Jacksonville    FL    32256   Florida
Los Altos       CA    94022   California
Menlo Park      CA    94025   California
Mountain View   CA    94040   California
Mountain View   CA    94063   California
Oakland         CA    94609   California
Palo Alto       CA    94303   California
Palo Alto       CA    94304   California
Phoenix         AZ    85008   Arizona
Phoenix         AZ    85016   Arizona
Princeton       NJ    08540   New Jersey
Redwood City    CA    94026   California
Redwood City    CA    94062   California
Redwood City    CA    94063   California
San Francisco   CA    94117   California
Sunnyvale       CA    94085   California
Sunnyvale       CA    94086   California
Wilmington      DE    19898   Delaware
```

*Ergebnis 2-4*

## Formen der SELECT-Anweisung

Obwohl die Syntax bei allen Informix-Produkten gleich bleibt, sind sowohl die Formulierung einer bestimmten SELECT-Anweisung als auch die Positionierung und Formatierung der Ergebnisausgabe abhängig von der jeweiligen Anwendung. Die Beispiele in diesem Kapitel und in Kapitel 3 "Komplexe SELECT-Anweisungen" zeigen die SELECT-Anweisungen und ihre Ausgaben in der Form, wie sie bei der interaktiven Abfragesprache von **DB-Access** oder **INFORMIX-SQL** eingegeben werden.

SELECT-Anweisungen können auch nicht-interaktiv in **ACE**-Listen ausgeführt werden; sie können auch in eine Sprache wie beispielsweise **INFORMIX-ESQL/C** eingebettet verwendet werden (wo sie wie ausführbarer Code behandelt werden), oder in **INFORMIX-4GL**-Programme eingebaut werden.

### Spezielle Datentypen

In den Beispielen dieses Kapitels wird der Datenbankserver **INFORMIX-On-Line** verwendet, der es ermöglicht, in Datenbankanwendungen die Datentypen VARCHAR, TEXT und BYTE zu benutzen. Diese Datentypen sind für Anwendungen, die unter **INFORMIX-SE** laufen, nicht verfügbar.

Kommt einer dieser drei Datentypen in einer SELECT-Anweisung vor und wird der interaktive Bildschirmeditor von **DB-Access** oder **INFORMIX-SQL** verwendet, so wird das Ergebnis der Abfrage unterschiedlich dargestellt.

- Wenn Sie eine Abfrage auf eine VARCHAR-Spalte ausführen, wird der ganze VARCHAR-Wert angezeigt, und zwar genauso, wie CHARACTER-Werte angezeigt werden.
- Wenn Sie eine TEXT-Spalte auswählen, wird der Inhalt der TEXT-Spalte angezeigt und Sie können darin blättern.
- Wenn Sie eine BYTE-Spalte abfragen, werden statt des aktuellen Wertes die Worte `<BYTE value>` angezeigt.

Auf Unterschiede, die spezifisch für VARCHAR, TEXT und BYTE sind, wird an den entsprechenden Stellen in diesem Kapitel hingewiesen.

Wenn der Native Language Support (NLS) aktiviert ist, können Sie mit der SELECT-Anweisung nach NCHAR- und NVARCHAR-Spalten anstelle ihrer CHAR- und VARCHAR-Entsprechungen suchen lassen. Der Datentyp NCHAR und NVARCHAR erlaubt die Eingabe von landessprachlichen Sonderzeichen in Feldern vom Typ CHARACTER.

Nähere Informationen zu diesen und anderen Datentypen erhalten Sie in Kapitel 9 "Das Datenmodell implementieren" sowie in Kapitel 3 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen* .

# SELECT-Anweisungen auf einzelne Tabellen

Es gibt viele Arten, eine Datenbanktabelle abzufragen. Mit der SELECT-Anweisung können Sie folgende Aufgaben erledigen:

- Abrufen aller oder bestimmter Spalten
- Abrufen aller oder bestimmter Datensätze
- Verwenden der abgerufenen Daten in Berechnungen oder anderen Funktionen
- Sortieren der Daten auf unterschiedliche Weisen.

# Selektion aller Spalten und Sätze

Die einfachste SELECT-Anweisung enthält lediglich die beiden zwingenden Klauseln SELECT und FROM.

## Das Jokerzeichen * (Stern)

In der folgenden Anweisung sind alle Spalten der Tabelle **manufact** in der *Auswahlliste* angegeben. Eine Auswahlliste ist eine Liste der Spaltennamen oder Ausdrücke, die Sie aus der Tabelle projizieren wollen.

```
SELECT manu_code, manu_name, lead_time
    FROM manufact
```

*Abfrage 2-5a*

In der nächsten Anweisung wird das *Jokerzeichen* * (Stern) als Kurzschreibweise für die Auswahlliste verwendet. Der Stern steht für die Namen aller Spalten einer Tabelle. Sie können das Jokerzeichen * immer dann verwenden, wenn Sie alle Spalten in der in der Datenbank festgelegten Reihenfolge anzeigen lassen wollen.

```
SELECT * FROM manufact
```

*Abfrage 2-5b*

Jede der beiden obigen Anweisungen erzeugt dieselbe Anzeige; in beiden Fällen wird als Ergebnis eine Liste aller Spalten und Sätze der Tabelle **manufact** ausgegeben. Das Ergebnis wird in der Form angezeigt, wie es im interaktiven Editor von **DB-Access** oder **INFORMIX-SQL (ISED)** angezeigt werden würde.

```
manu_code manu_name       lead_time

 SMT      Smith              3
 ANZ      Anza               5
 NRG      Norge              7
 HSK      Husky              5
 HRO      Hero               4
 SHM      Shimara           30
 KAR      Karsten           21
 NKL      Nikolus            8
 PRC      ProCycle           9
```

*Ergebnis 2-5*

## Umstellen der Spalten

Das folgende Beispiel zeigt, wie Sie die Reihenfolge ändern können, in der die Spalten aufgelistet sind. Sie müssen hierfür deren Reihenfolge in der Auswahlliste ändern.

```
SELECT manu_name, manu_code, lead_time
    FROM manufact
```

*Abfrage 2-6*

Diese Auswahlliste enthält dieselben Spalten wie die vorherige; da aber die Spalten in einer anderen Reihenfolge angegeben wurden, ist auch die Anzeige anders.

```
manu_name       manu_code lead_time

 Smith           SMT          3
 Anza            ANZ          5
 Norge           NRG          7
 Husky           HSK          5
 Hero            HRO          4
 Shimara         SHM         30
 Karsten         KAR         21
 Nikolus         NKL          8
 ProCycle        PRC          9
```

*Ergebnis 2-6*

## Sortieren der Datensätze

Sie können Daten in einer bestimmten Reihenfolge sortieren, indem Sie Ihrer SELECT-Anweisung die ORDER BY-Klausel hinzufügen. Die Spalten, die Sie in der ORDER BY-Klausel angeben , müssen in der Auswahlliste explizit oder implizit enthalten sein.

In einer *expliziten* Auswahlliste sind alle Spaltennamen enthalten:

```
SELECT manu_code, manu_name, lead_time
    FROM manufact
    ORDER BY lead_time
```

*Abfrage 2-7a*

In einer *impliziten* Auswahlliste wird das Jokerzeichen Stern (*) verwendet:

```
SELECT * FROM manufact
    ORDER BY lead_time
```

*Abfrage 2-7b*

Jede dieser zwei Anweisungen erzeugt dieselbe Anzeige, nämlich eine Liste aller Spalten und Sätze der Tabelle **manufact**, sortiert nach der Spalte **lead_time**:

```
manu_code manu_name       lead_time

 SMT      Smith               3
 HRO      Hero                4
 HSK      Husky               5
 ANZ      Anza                5
 NRG      Norge               7
 NKL      Nikolus             8
 PRC      ProCycle            9
 KAR      Karsten            21
 SHM      Shimara            30
```

*Ergebnis 2-7*

## Aufsteigende Reihenfolge

Die abgerufenen Daten werden standardmäßig in *aufsteigender* Reihenfolge sortiert und angezeigt. Aufsteigende Reihenfolge bedeutet, daß beim Datentyp CHARACTER zunächst die Großbuchstaben A-Z, dann erst die Kleinbuchstaben a-z sortiert werden. Bei numerischen Datentypen wird vom kleinsten zum größten Wert sortiert. Die Sortierung der Datentypen DATE und DATETIME erfolgt vom frühesten zum spätesten Zeitpunkt und beim Datentyp INTERVAL von der kürzesten zur längsten Zeitspanne.

## Absteigende Reihenfolge

Bei der absteigenden Reihenfolge werden Buchstaben in der Reihenfolge z bis A sortiert, Ziffern in der Reihenfolge "hohe Zahl" bis "niedrige Zahl". Daten der Datentypen DATE und DATETIME werden in der Reihenfolge "ältestes Datum" bis "jüngstes Datum" sortiert. Bei INTERVAL-Werten ist die

Reihenfolge "längster Zeitraum" bis "kürzester Zeitraum". Im folgenden Beispiel werden die Ergebnisse der Abfrage in *absteigender* Reihenfolge ausgegeben. Diese Reihenfolge wird mit dem Schlüsselwort DESC festgelegt:

```
SELECT * FROM manufactmanufact
    ORDER BY lead_time DESC
```

*Abfrage 2-8*

Sie können jede Spalte (außer TEXT oder BYTE) in der ORDER BY-Klausel angeben. Der Datenbankserver sortiert die Daten entsprechend den Werten dieser Spalte.

```
manu_code manu_name       lead_time

 SHM       Shimara            30
 KAR       Karsten            21
 PRC       ProCycle            9
 NKL       Nikolus             8
 NRG       Norge               7
 HSK       Husky               5
 ANZ       Anza                5
 HRO       Hero                4
 SMT       Smith               3
```

*Ergebnis 2-8*

## Sortieren nach mehreren Spalten

Mit einer *geschachtelten Sortierung* können Sie auch nach zwei oder mehr Spalten sortieren. Die Standardeinstellung ist immer *aufsteigend.* Die Spalte hat Vorrang, die in der ORDER BY-Klausel zuerst angegeben ist.

Die folgenden zwei Beispiele und ihre Ergebnisse zeigen eine geschachtelte Sortierung. Die Reihenfolge, in der die ausgewählten Daten angezeigt werden, wird verändert, indem die Reihenfolge der beiden Spaltennamen in der ORDER BY-Klausel vertauscht wird:

```
SELECT * FROM stock
    ORDER BY manu_code, unit_price
```

*Abfrage 2-9*

In dieser Abfrage werden die Daten der Spalte **manu_code** in alphabetischer Reihenfolge angezeigt. Innerhalb der Sätze mit gleichem **manu_code** (z. B. ANZ, HRO) wird der **unit_price** in aufsteigender Reihenfolge aufgelistet:

```
stock_num manu_code description     unit_price unit unit_descr

        5 ANZ       tennis racquet      $19.80 each each
        9 ANZ       volleyball net      $20.00 each each
        6 ANZ       tennis ball         $48.00 case 24 cans/case
      313 ANZ       swim cap            $60.00 box  12/box
      201 ANZ       golf shoes          $75.00 each each
      310 ANZ       kick board          $84.00 case 12/case
      301 ANZ       running shoes       $95.00 each each
      304 ANZ       watch              $170.00 box  10/box
      110 ANZ       helmet             $244.00 case 4/case
      205 ANZ       3 golf balls       $312.00 case 24/case
        8 ANZ       volleyball         $840.00 case 24/case
      302 HRO       ice pack             $4.50 each each
      309 HRO       ear drops           $40.00 case 20/case
      .
      .
      .
      113 SHM       18-spd, assmbld    $685.90 each each
        5 SMT       tennis racquet      $25.00 each each
        6 SMT       tennis ball         $36.00 case 24 cans/case
        1 SMT       baseball gloves    $450.00 case 10 gloves/case
```

*Ergebnis 2-9*

Im folgenden Beispiel ist die Reihenfolge der Spalten der ORDER BY-Klausel vertauscht:

```
SELECT * FROM stock
    ORDER BY unit_price, manu_code
```

*Abfrage 2-10*

Hier werden die Daten in aufsteigender Reihenfolge des **unit_price** angezeigt. Haben zwei oder mehr Datensätze denselben **unit_price** (z. B. $20.00, $48.00, $312.00), ist das **manu_code** alphabetisch sortiert.

```
stock_num manu_code description     unit_price unit unit_descr

      302 HRO       ice pack            $4.50 each each
      302 KAR       ice pack            $5.00 each each
        5 ANZ       tennis racquet     $19.80 each each
        9 ANZ       volleyball net     $20.00 each each
      103 PRC       frnt derailleur    $20.00 each each
      106 PRC       bicycle stem       $23.00 each each
        5 SMT       tennis racquet     $25.00 each each
      .
      .
      .
      301 HRO       running shoes      $42.50 each each
      204 KAR       putter             $45.00 each each
      108 SHM       crankset           $45.00 each each
        6 ANZ       tennis ball        $48.00 case 24 cans/case
      305 HRO       first-aid kit      $48.00 case 4/case
      303 PRC       socks              $48.00 box  24 pairs/box
      311 SHM       water gloves       $48.00 box  4 pairs/box
      .
      .
      .
      110 HSK       helmet            $308.00 case 4/case
      205 ANZ       3 golf balls      $312.00 case 24/case
      205 HRO       3 golf balls      $312.00 case 24/case
      205 NKL       3 golf balls      $312.00 case 24/case
        1 SMT       baseball gloves   $450.00 case 10 gloves/case
        4 HRO       football          $480.00 case 24/case
      102 PRC       bicycle brakes    $480.00 case 4 sets/case
      111 SHM       10-spd, assmbld   $499.99 each each
      112 SHM       12-spd, assmbld   $549.00 each each
        7 HRO       basketball        $600.00 case 24/case
      203 NKL       irons/wedge       $670.00 case 2 sets/case
      113 SHM       18-spd, assmbld   $685.90 each each
        1 HSK       baseball gloves   $800.00 case 10 gloves/case
        8 ANZ       volleyball        $840.00 case 24/case
        4 HSK       football          $960.00 case 24/case
```

*Ergebnis 2-10*

Sowohl die Reihenfolge, in der die Spalten bei der ORDER BY-Klausel angegeben werden, ist wichtig, als auch die Position des Schlüsselwortes DESC. Obwohl jede der folgenden SELECT-Anweisungen in der ORDER BY-Klausel dieselben Bestandteile enthält, unterscheiden sich deren (nicht abgedruckte) Ergebnisse voneinander:

```
SELECT * FROM stock

ORDER BY manu_code, unit_price DESC

SELECT * FROM stock

ORDER BY unit_price, manu_code DESC

SELECT * FROM stock

ORDER BY manu_code DESC, unit_price

SELECT * FROM stock

ORDER BY unit_price DESC, manu_code
```

*Abfrage 2-11*

# Selektion spezifischer Spalten

Der vorherige Abschnitt hat gezeigt, wie man alle Daten einer Tabelle auswählen und sortieren kann. Sie werden jedoch in der Regel nur die Daten einer oder mehrerer ausgewählter Spalten benötigen. Auch in diesem Fall verwenden Sie die SELECT- und FROM-Klauseln, geben die Spalten und die Tabelle an und sortieren nach Bedarf die Daten mit der ORDER BY-Klausel in auf- oder absteigender Reihenfolge.

Mit der folgenden SELECT-Anweisung können Sie alle Kundennummern in der Tabelle **orders** suchen.

```
SELECT customer_num FROM orders
```

*Abfrage 2-12*

Aufgrund dieser Anweisung werden lediglich alle Daten aus der Spalte **customer_num** der Tabelle **orders** ausgewählt und die Kundennummern aller Aufträge, inklusive doppelter Werte, aufgelistet.

Die Ausgabe enthält eine Reihe doppelter Werte, da einige Kunden mehr als einen Auftrag erteilt haben. Manchmal sollen doppelte Werte in einer Projektion enthalten sein. Ein anderes Mal sind nur die verschiedenen Werte interessant und nicht, wie oft jeder Wert vorkommt.

```
customer_num

         101
         104
         104
         104
         104
         106
         106
         110
         110
         111
         112
         115
         116
         117
         117
         119
         120
         121
         122
         123
         124
         126
         127
```

*Ergebnis 2-12*

Sie können mit dem Schlüsselwort DISTINCT oder dessen Synonym UNIQUE, die Ausgabe doppelter Werte unterdrücken:

```
SELECT DISTINCT customer_num FROM orders

SELECT UNIQUE customer_num FROM orders
```

*Abfrage 2-13*

Beide Anweisungen bewirken, daß jede Kundennummer der Tabelle **orders** nur einmal angezeigt und dadurch eine leichter lesbare Liste (siehe Ergebnis 2-13) erstellt wird.

```
customer_num

         101
         104
         106
         110
         111
         112
         115
         116
         117
         119
         120
         121
         122
         123
         124
         126
         127
```

*Ergebnis 2-13*

Angenommen, Sie bearbeiten einen Kundenauftrag und suchen die Auftragsnummer DM354331. Mit der folgenden Anweisung lassen Sie sich alle Auftragsnummern der Tabelle **orders** anzeigen:

```
SELECT po_num FROM orders
```

*Abfrage 2-14*

Aufgrund dieser SELECT-Anweisung werden die Daten der Spalte **po_num** in der Tabelle **orders** abgerufen:

```
po_num

 B77836
 9270
 B77890
 8006
 2865
 Q13557
 278693
 LZ230
 4745
 429Q
 B77897
 278701
 B77930
 8052
 MA003
 PC6782
 DM354331
 S22942
 Z55709
 W2286
 C3288
 W9925
 KF2961
```

*Ergebnis 2-14*

Die Liste befindet sich jedoch in keiner brauchbaren Reihenfolge. Sie können die ORDER BY-Klausel hinzufügen, um die Daten der Spalte in aufsteigender Reihenfolge zu sortieren. Dies vereinfacht die Suche nach einer bestimmten **po_num**:

```
SELECT po_num FROM orders
    ORDER BY po_numk
```

*Abfrage 2-15*

```
po_num

 278693
 278701
 2865
 429Q
 4745
 8006
 8052
 9270
 B77836
 B77890
 B77897
 B77930
 C3288
 DM354331
 KF2961
 LZ230
 MA003
 PC6782
 Q13557
 S22942
 W2286
 W9925
 Z55709
```

*Ergebnis 2-15*

Um mehrere Spalten einer Tabelle auszuwählen, geben Sie diese in der Auswahlliste der SELECT-Klausel an. Die Reihenfolge (von links nach rechts), in der die Spalten ausgewählt werden, entspricht der Reihenfolge, in der die Spalten ausgegeben werden.

```
SELECT paid_date, ship_date, order_date,
       customer_num, order_num, po_num
    FROM orders
    ORDER BY paid_date, order_date, customer_num
```

*Abfrage 2-16*

Wie bereits oben im Abschnitt "Sortieren nach mehreren Spalten" gezeigt, können Sie mit der ORDER BY-Klausel Daten in auf- oder absteigender Reihenfolge sortieren und geschachtelte Sortierungen durchführen. Im folgenden Beispiel wird die aufsteigende Reihenfolge verwendet :

```
paid_date  ship_date  order_date customer_num   order_num po_num

           05/30/1993 05/22/1993          106        1004 8006
                      05/30/1993          112        1006 Q13557
           06/05/1993 05/31/1993          117        1007 278693
           06/29/1993 06/18/1993          117        1012 278701
           07/12/1993 06/29/1993          119        1016 PC6782
           07/13/1993 07/09/1993          120        1017 DM354331
06/03/1993 05/26/1993 05/21/1993          101        1002 9270
06/14/1993 05/23/1993 05/22/1993          104        1003 B77890
06/21/1993 06/09/1993 05/24/1993          116        1005 2865
07/10/1993 07/03/1993 06/25/1993          106        1014 8052
07/21/1993 07/06/1993 06/07/1993          110        1008 LZ230
07/22/1993 06/01/1993 05/20/1993          104        1001 B77836
07/31/1993 07/10/1993 06/22/1993          104        1013 B77930
08/06/1993 07/13/1993 07/10/1993          121        1018 S22942
08/06/1993 07/16/1993 07/11/1993          122        1019 Z55709
08/21/1993 06/21/1993 06/14/1993          111        1009 4745
08/22/1993 06/29/1993 06/17/1993          115        1010 429Q
08/22/1993 07/25/1993 07/23/1993          124        1021 C3288
08/22/1993 07/30/1993 07/24/1993          127        1023 KF2961
08/29/1993 07/03/1993 06/18/1993          104        1011 B77897
08/31/1993 07/16/1993 06/27/1993          110        1015 MA003
09/02/1993 07/30/1993 07/24/1993          126        1022 W9925
09/20/1993 07/16/1993 07/11/1993          123        1020 W2286
```

*Ergebnis 2-16*

Wenn Sie mehrere Spalten einer Tabelle auswählen und sortieren wollen, können Sie Zahlen in der ORDER BY-Klausel dazu verwenden, auf die entsprechende Position einer Spalte in der Klausel verweisen. In den beiden Anweisungen im folgenden Beispiel werden dieselben Daten abgefragt und angezeigt:

```
SELECT customer_num, order_num, po_num, order_date
    FROM orders
    ORDER BY 4, 1

SELECT customer_num, order_num, po_num, order_date
    FROM orders
    ORDER BY order_date, customer_num
```

*Abfrage 2-17*

Beide Anweisungen rufen dieselben Daten ab und zeigen sie in derselben Form an:

```
customer_num    order_num po_num      order_date

         104         1001 B77836      05/20/1993
         101         1002 9270        05/21/1993
         104         1003 B77890      05/22/1993
         106         1004 8006        05/22/1993
         116         1005 2865        05/24/1993
         112         1006 Q13557      05/30/1993
         117         1007 278693      05/31/1993
         110         1008 LZ230       06/07/1993
         111         1009 4745        06/14/1993
         115         1010 429Q        06/17/1993
         104         1011 B77897      06/18/1993
         117         1012 278701      06/18/1993
         104         1013 B77930      06/22/1993
         106         1014 8052        06/25/1993
         110         1015 MA003       06/27/1993
         119         1016 PC6782      06/29/1993
         120         1017 DM354331    07/09/1993
         121         1018 S22942      07/10/1993
         122         1019 Z55709      07/11/1993
         123         1020 W2286       07/11/1993
         124         1021 C3288       07/23/1993
         126         1022 W9925       07/24/1993
         127         1023 KF2961      07/24/1993
```

*Ergebnis 2-17*

Auch wenn Sie den Spaltennamen Zahlen zugewiesen haben, können Sie der ORDER BY-Klausel das Schlüsselwort DESC hinzufügen:

```
SELECT customer_num, order_num, po_num, order_date
    FROM orders
    ORDER BY 4 DESC, 1
```

*Abfrage 2-18*

In diesem Fall sind die Daten zunächst sortiert in absteigender Reihenfolge nach **order_date** und innerhalb des Datums in aufsteigender Reihenfolge nach **customer_num** .

## Sortierung und Native Language Support (NLS)

Mit der DBNLS-Umgebungsvariable aktivieren Sie den Native Language Support (NLS) . Anschließend können Sie eine spezifische Sprachumgebung einschalten, indem Sie die LANG-Umgebungsvariable setzen. Mit der X/Open-Kategorie LC_COLLATE können Sie die Sortierreihenfolge weiter beeinflussen.

Die folgende SELECT-Anweisung mit ORDER BY-Klausel bezieht sich auf eine Spalte, die Daten vom Typ NCHAR enthält (NCHAR ist das NLS-Äquivalent von CHAR). Die verwendete NLS-Tabelle **abonnés** wird separat mit der Beispieldatenbank ausgeliefert. Die gezeigte NLS-Anweisung wird unter dem Dateinamen **sel_nls1.sql** zusammen mit NLS-fähigem **DB-Access** ausgeliefert.

```
SELECT numéro,nom,prénom
    FROM abonnés
    ORDER BY nom;
```

*Abfrage 2-19*

Wie das Ergebnis dieser Abfrage sortiert wird, hängt davon ab, ob NLS aktiviert ist, von den Einstellungen der Umgebungsvariablen LANG und LC_COLLATE, sowie davon, wie das jeweilige System mit Groß- und Kleinschreibung umgeht. Zwei mögliche Ergebnistabellen werden hier gezeigt:

| numéro | nom | prénom |
|---|---|---|
| 13612 | Azevedo | Edouardo Freire |
| 13606 | Dupré | Michèle Françoise |
| 13607 | Hammer | Gerhard |
| 13602 | Hämmerle | Greta |
| 13604 | LaForêt | Jean-Noël |
| 13610 | LeMaître | Héloïse |
| 13613 | Llanero | Gloria Dolores |
| 13603 | Montaña | José Antonio |
| 13611 | Oatfield | Emily |
| 13609 | Tiramisù | Paolo Alfredo |
| 13600 | da Sousa | João Lourenço Antunes |
| 13615 | di Girolamo | Giuseppe |
| 13601 | Ålesund | Sverre |
| 13608 | Étaix | Émile |
| 13605 | Ötker | Hans-Jürgen |
| 13614 | Øverst | Per-Anders |

*Ergebnis 2-19a*

Diese Ergebnistabelle entspricht der Standard-Sortierung nach ASCII, die Großbuchstaben generell vor Kleinbuchstaben einordnet und Einträge, die mit großgeschriebenen landessprachlichen Sonderzeichen beginnen (Ålesund, Étaix, Ötker, and Øverst), generell ans Ende der Liste setzt.

| numéro | nom | prénom |
|---|---|---|
| 13601 | Ålesund | Sverre |
| 13612 | Azevedo | Edouardo Freire |
| 13600 | da Sousa | João Lourenço Antunes |
| 13615 | di Girolamo | Giuseppe |
| 13606 | Dupré | Michèle Françoise |
| 13608 | Étaix | Émile |
| 13607 | Hammer | Gerhard |
| 13602 | Hämmerle | Greta |
| 13604 | LaForêt | Jean-Noël |
| 13610 | LeMaître | Héloïse |
| 13613 | Llanero | Gloria Dolores |
| 13603 | Montaña | José Antonio |
| 13611 | Oatfield | Emily |
| 13605 | Ötker | Hans-Jürgen |
| 13614 | Øverst | Per-Anders |
| 13609 | Tiramisù | Paolo Alfredo |

*Ergebnis 2-19b*

Das obige Ergebnis wurde mit der identischen SELECT-Anweisung erzielt. Dank der entsprechenden Einstellung der NLS-Umgebungsvariablen werden Namen, die mit landessprachlichen Sonderzeichen beginnen (Ålesund, Étaix, Ötker, and Øverst), anders eingeordnet als nach der standardmäßigen Sortierung nach ASCII. Die Namen sind jetzt korrekt in der Liste einsortiert und stehen nicht mehr einfach an ihrem Ende. Auch die Unterscheidung zwischen Groß- und Kleinschreibung ist jetzt entfallen.

## Selektion von Zeichenkettenausschnitten

Sie können auch nur einen Teil eines Wertes auswählen, der in einer CHARACTER-Spalte steht. Dafür geben Sie in der Auswahlliste den gewünschten Zeichenkettenausschnitt an. Angenommen, Ihre Marketing-Abteilung plant ein Mailing an Ihre Kunden und möchte sich eine ungefähre Vorstellung über deren geographische Verteilung anhand der Postleitzahlen (zipcode) verschaffen. Eine entsprechende Abfrage könnte wie folgt aussehen:

```
SELECT zipcode[1,3], customer_num
    FROM customer
    ORDER BY zipcode
```

*Abfrage 2-20*

In dieser Abfrage werden von der Spalte **zipcode** (die den Staat identifiziert) nur die ersten drei Zeichen als Zeichenkettenausschnitt sowie die komplette Spalte **customer_num** ausgewählt. Die Ausgabe erfolgt in aufsteigender Reihenfolge der Postleitzahlen.

```
zipcode customer_num

021              125
080              119
085              122
198              121
322              123
604              127
740              124
802              126
850              128
850              120
940              105
940              112
940              113
940              115
940              104
940              116
940              110
940              114
940              106
940              108
940              117
940              111
940              101
940              109
941              102
943              103
943              107
946              118
```

*Ergebnis 2-20*

# Verwendung der WHERE-Klausel

Manchmal wollen Sie nur die Aufträge sehen, die von einem bestimmten Kunden oder Außendienstmitarbeiter erteilt wurden. In diesem Fall fügen Sie der SELECT-Anweisung die WHERE-Klausel hinzu, um nur diese Datensätze abzurufen.

Die WHERE-Klausel dient dazu, eine *Vergleichs-* oder *Join-Bedingung* zu stellen. Dieser Abschnitt demonstriert nur die erste Verwendung. Join-Bedingungen werden im nächsten Kapitel beschrieben.

Die Datensätze, die die Ergebnismenge einer SELECT-Anweisung bilden, werden als *aktuelle Liste* dieser Anweisung bezeichnet. Die Ergebnismenge einer *Einzel*-SELECT-Anweisung besteht aus einem Satz. Bei **INFORMIX-4GL** oder einem **SQL-API** erfordert die Abfrage mehrerer Datensätze die Verwendung eines *Cursors*. Weitere Informationen hierzu finden Sie in Kapitel 5 "SQL in Programmen" und Kapitel 6 "Programme zur Veränderung von Daten".

# Eine Vergleichsbedingung erstellen

Die WHERE-Klausel einer SELECT-Anweisung gibt genau die Datensätze an, die Sie sehen wollen. Eine Vergleichsbedingung verwendet bestimmte *Schlüsselwörter* und *Operatoren*, um Suchkriterien zu definieren.

Mit einem der Schlüsselwörter BETWEEN, IN, LIKE oder MATCHES können Sie z. B. Ausdrücke auf Gleichheit prüfen oder mit den Schlüsselwörtern IS NULL auf Nullwerte prüfen. Das Schlüsselwort NOT können Sie bei jedem dieser Schlüsselwörter verwenden, um die Umkehrung der Bedingung anzugeben.

In Bild 2-2 sind die *relationalen Operatoren* aufgelistet, die Sie anstelle der Schlüsselwörter in der WHERE-Klausel verwenden können, um auf Gleichheit hin zu prüfen.

| Operator | Bedeutung |
|---|---|
| = | gleich |
| != oder <> | ungleich |
| > | größer |
| >= | größer gleich |
| < | kleiner |
| <= | kleiner gleich |

*Bild 2-2* ***Relationale Operatoren, die auf Gleichheit hin prüfen***

Der Vergleich von CHAR-Ausdrücken erfolgt nach der ASCII -Vergleichsreihenfolge, d. h. Kleinbuchstaben kommen nach Großbuchstaben und Buchstaben nach den Ziffern. Eine Darstellung des ASCII-Zeichensatzes finden Sie im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*. Bei DATE- und DATETIME-Ausdrücken bedeutet "größer als" einen *späteren Zeitpunkt* und bei INTERVAL eine *längere Zeitdauer*. TEXT oder BYTE-Spalten können Sie nicht in Zeichenkettenausdrücken verwenden, außer, wenn Sie auf NULL-Werte abprüfen.

Sie verwenden die vorgestellten Schlüsselwörter und Operatoren in der WHERE-Klausel, um Abfragen mit Vergleichsbedingungen zu erstellen, die folgendes leisten:

Sie verwenden die vorgestellten Schlüsselwörter und Operatoren in der WHERE-Klausel, um Abfragen mit Vergleichsbedingungen zu erstellen, die folgendes leisten:

- Werte einschließen
- Werte ausschließen
- einen Wertebereich ermitteln
- eine Teilmenge der Werte ermitteln
- NULL-Werte erkennen

Ferner können Sie diese Schlüsselwörter und Operatoren dazu verwenden, Abfragen mit Vergleichsbedingungen zu erstellen, die eine Suche nach Text flexibel durchführen. Hierfür können Sie die folgenden Kriterien verwenden:

- genauen Textvergleich
- Jokerzeichen, die einzelne Zeichen ersetzen
- eingeschränkte Jokerzeichen, die einzelne Zeichen ersetzen
- Jokerzeichen, die einen Text von beliebiger Länge ersetzen
- Subskription (nach Feldausschnitten suchen)

Die Beispiele im folgenden Abschnitt stellen diese Abfragearten dar.

## Datensätze in die Abfrage einschließen

Verwenden Sie den Operator = (*ist gleich*) wie unten gezeigt in der WHERE-Klausel , um bestimmte Datensätze in Ihre Abfrage einzuschließen.

```
SELECT customer_num, call_code, call_dtime, res_dtime
    FROM cust_calls
    WHERE user_id = 'maryj'
```

*Abfrage 2-21*

Diese SELECT-Anweisung liefert die folgenden Datensätze zurück:

```
customer_num call_code call_dtime       res_dtime

         106 D         1993-06-12 08:20 1993-06-12 08:25
         121 O         1993-07-10 14:05 1993-07-10 14:06
         127 I         1993-07-31 14:30
```

*Ergebnis 2-21*

## Datensätze aus der Abfrage ausschließen

Verwenden Sie die Operatoren != oder <> (*ungleich*) in der WHERE-Klausel, um Datensätze aus Ihrer Abfrage auszuschließen.

In den folgenden Beispielen wird angenommen , daß Sie aus einer ANSI-konformen Datenbank auswählen. Es wird deshalb in den Anweisungen der *Eigentümer-* oder login-Name des Erstellers der Tabelle **customer** angegeben. Diese Angabe ist nicht erforderlich, wenn der Ersteller der Tabelle der aktuelle Benutzer ist oder wenn die Datenbank nicht ANSI-konform ist. Dennoch ist diese Angabe in beiden Fällen nicht falsch. Eine vollständige Erläuterung der *Angabe des Eigentümers* finden Sie in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* .

```
SELECT customer_num, company, city, state
    FROM odin.customer
    WHERE state != 'CA'

SELECT customer_num, company, city, state
    FROM odin.customer
    WHERE state <> 'CA'
```

*Abfrage 2-22*

Jede der beiden Anweisungen in diesem Beispiel schließen Datensätze dadurch aus, daß der Wert in der Spalte **state** nicht gleich CA sein darf. Diese Spalte gehört zur Tabelle **customer**, deren Eigentümer **odin** ist.

```
customer_num  company              city            state

         119 The Triathletes Club Cherry Hill     NJ
         120 Century Pro Shop     Phoenix         AZ
         121 City Sports          Wilmington      DE
         122 The Sporting Life    Princeton       NJ
         123 Bay Sports           Jacksonville    FL
         124 Putnum's Putters     Bartlesville    OK
         125 Total Fitness Sports Brighton        MA
         126 Neelie's Discount Sp Denver          CO
         127 Big Blue Bike Shop   Blue Island     NY
         128 Phoenix College      Phoenix         AZ
```

*Ergebnis 2-22*

## Bereiche von Datensätzen angeben

Im folgenden Beispiel werden zwei Möglichkeiten gezeigt, Bereiche von Datensätzen in der WHERE-Klausel anzugeben:

```
SELECT catalog_num, stock_num, manu_code, cat_advert
    FROM catalog
    WHERE catalog_num BETWEEN 10005 AND 10008

SELECT catalog_num, stock_num, manu_code, cat_advert
    FROM catalog
    WHERE catalog_num >= 10005 AND catalog_num <= 10008
```

*Abfrage 2-23*

Jede der beiden Anweisungen gibt den Bereich der **catalog_num** von 10005 bis 10008 inklusive der Eckwerte an. Die erste Anweisung verwendet Schlüsselwörter, die zweite relationale Operatoren. Folgende Datensätze werden dabei ausgegeben:

```
catalog_num  10005
stock_num    3
manu_code    HSK
cat_advert   High-Technology Design Expands the Sweet Spot

catalog_num  10006
stock_num    3
manu_code    SHM
cat_advert   Durable Aluminum for High School and Collegia
             te Athletes

catalog_num  10007
stock_num    4
manu_code    HSK
cat_advert   Quality Pigskin with Joe Namath Signature

catalog_num  10008
stock_num    4
manu_code    HRO
cat_advert   Highest Quality Football for High School
              and Collegiate Competitions
```

*Ergebnis 2-23*

Beachten Sie: Obwohl die Tabelle **catalog** eine Spalte des Datentyps BYTE enthält, ist diese Spalte nicht in der SELECT-Anweisung enthalten. Dies ist darin begründet, daß in der Ausgabe nach dem Spaltennamen nur die Worte <BYTE value> angezeigt werden würden. Die Werte TEXT und BYTE können Sie sich nur anzeigen lassen, wenn Sie in **INFORMIX-SQL**- oder **INFORMIX-4GL**-Masken das Attribut PROGRAM verwenden oder wenn Sie ein Programm in **4GL** bzw. einem **SQL-API** schreiben , um diese Werte anzeigen zu lassen.

## Bestimmte Datensatz-Bereiche ausschließen

```
SELECT fname, lname, company, city, state
    FROM customer
    WHERE zipcode NOT BETWEEN '94000' AND '94999'
    ORDER BY state
```

*Abfrage 2-24*

Durch die Schlüsselwörter NOT BETWEEN wird in diesem Beispiel eine Bedingung gestellt, die all diejenigen Datensätze ausschließt, deren Wert in der Spalte **zipcode** im Bereich von `94000` bis `94999` liegt:

| fname | lname | company | city | state |
|---|---|---|---|---|
| Fred | Jewell | Century* Pro Shop | Phoenix | AZ |
| Frank | Lessor | Phoenix University | Phoenix | AZ |
| Eileen | Neelie | Neelie's Discount Sp | Denver | CO |
| Jason | Wallack | City Sports | Wilmington | DE |
| Marvin | Hanlon | Bay Sports | Jacksonville | FL |
| James | Henry | Total Fitness Sports | Brighton | MA |
| Bob | Shorter | The Triathletes Club | Cherry Hill | NJ |
| Cathy | O'Brian | The Sporting Life | Princeton | NJ |
| Kim | Satifer | Big Blue Bike Shop | Blue Island | NY |
| Chris | Putnum | Putnum's Putters | Bartlesville | OK |

*Ergebnis 2-24*

### Verwendung der WHERE-Klausel, um eine Teilmenge der Werte zu ermitteln

Auch im folgenden Beispiel wird davon ausgegangen, daß eine ANSI-konforme Datenbank verwendet wird. Hier wird der Name der Eigentümerin in Anführungszeichen gesetzt, um die *automatische* Umwandlung des Namens **Aleta** in Großbuchstaben zu *vermeiden*.

```
SELECT lname, city, state, phone
    FROM 'Aleta'.customer
    WHERE state = 'AZ' OR state = 'NJ'
    ORDER BY lname

SELECT lname, city, state, phone
    FROM 'Aleta'.customer
    WHERE state IN ('AZ', 'NJ')
    ORDER BY lname
```

*Abfrage 2-25*

Jede dieser beiden Anweisungen ruft eine Teilmenge der Datensätze aus der Tabelle **Aleta.customer** ab, und zwar die Sätze, bei denen in der Spalte **state** AZ oder NJ eingetragen ist:

```
lname           city            state phone

Jewell          Phoenix         AZ    602-265-8754
Lessor          Phoenix         AZ    602-533-1817
O'Brian         Princeton       NJ    609-342-0054
Shorter         Cherry Hill     NJ    609-663-6079
```

*Ergebnis 2-25*

Beachten Sie: Das Schlüsselwort IN können Sie bei TEXT- oder BYTE-Spalten nicht verwenden.

Im nächsten Beispiel für eine ANSI-konforme Datenbank ist der Tabelleneigentümer nicht in Anführungszeichen gesetzt. Während die beiden vorangegangenen Abfragen die Tabelle **Aleta.customer** durchsucht haben,

durchsucht diese SELECT-Anweisung die Tabelle **ALETA.customer** (also eine andere Tabelle). Eigentümer-Namen, die nicht in Anführungszeichen stehen, werden automatisch in Großbuchstaben umgewandelt.

```
SELECT lname, city, state, phone
    FROM Aleta.customer
    WHERE state NOT IN ('AZ', 'NJ')
    ORDER BY state
```

*Abfrage 2-26*

Durch die Verwendung der Schlüsselwörter NOT IN wurde die Teilmenge so verändert, daß diejenigen Datensätze ausgeschlossen werden, deren Wert in der Spalte **state** AZ oder NJ ist. Das Ergebnis ist nach der Spalte **state** sortiert.

| lname | city | state | phone |
|---|---|---|---|
| Pauli | Sunnyvale | CA | 408-789-8075 |
| Sadler | San Francisco | CA | 415-822-1289 |
| Currie | Palo Alto | CA | 415-328-4543 |
| Higgins | Redwood City | CA | 415-368-1100 |
| Vector | Los Altos | CA | 415-776-3249 |
| Watson | Mountain View | CA | 415-389-8789 |
| Ream | Palo Alto | CA | 415-356-9876 |
| Quinn | Redwood City | CA | 415-544-8729 |
| Miller | Sunnyvale | CA | 408-723-8789 |
| Jaeger | Redwood City | CA | 415-743-3611 |
| Keyes | Sunnyvale | CA | 408-277-7245 |
| Lawson | Los Altos | CA | 415-887-7235 |
| Beatty | Menlo Park | CA | 415-356-9982 |
| Albertson | Redwood City | CA | 415-886-6677 |
| Grant | Menlo Park | CA | 415-356-1123 |
| Parmelee | Mountain View | CA | 415-534-8822 |
| Sipes | Redwood City | CA | 415-245-4578 |
| Baxter | Oakland | CA | 415-655-0011 |
| Neelie | Denver | CO | 303-936-7731 |
| Wallack | Wilmington | DE | 302-366-7511 |
| Hanlon | Jacksonville | FL | 904-823-4239 |
| Henry | Brighton | MA | 617-232-4159 |
| Satifer | Blue Island | NY | 312-944-5691 |
| Putnum | Bartlesville | OK | 918-355-2074 |

*Ergebnis 2-26*

## NULL-Werte erkennen

Um auf NULL-Werte hin abzuprüfen, nutzen Sie die Optionen IS NULL oder IS NOT NULL. Ein NULL-Wert repräsentiert entweder einen Wert, der nicht bekannt oder nicht verfügbar ist, oder zeigt an, daß keine Daten vorhanden sind. Ein NULL-Wert ist *nicht gleichbedeutend* mit der Ziffer Null oder einem Leerzeichen.

```
SELECT order_num, customer_num, po_num, ship_date
    FROM orders
    WHERE paid_date IS NULL
    ORDER BY customer_num
```

*Abfrage 2-27*

Diese SELECT-Anweisung liefert alle Datensätze, die in der Spalte **paid_date** einen NULL-Wert enthalten.

| order_num | customer_num | po_num | ship_date |
|---|---|---|---|
| 1004 | 106 | 8006 | 05/30/1993 |
| 1006 | 112 | Q13557 | |
| 1007 | 117 | 278693 | 06/05/1993 |
| 1012 | 117 | 278701 | 06/29/1993 |
| 1016 | 119 | PC6782 | 07/12/1993 |
| 1017 | 120 | DM354331 | 07/13/1993 |

*Ergebnis 2-27*

## Zusammengesetzte Bedingungen

Mit den *logischen Operatoren* AND, OR und NOT können Sie zwei oder mehr Vergleichsbedingungen *(Boolsche Ausdrücke)* miteinander verknüpfen. Eine Vergleichsbedingung wird als `wahr` oder `falsch` bewertet oder als `unbekannt`, wenn ein NULL-Wert darin vorkommt. In einer Vergleichsbedingung können Sie TEXT- oder BYTE-Objekte nur dann verwenden, wenn Sie auf NULL-Werte hin abprüfen.

In der folgenden Abfrage verbindet der Operator AND zwei Vergleichsausdrücke in der WHERE-Klausel.

```
SELECT order_num, customer_num, po_num, ship_date
    FROM orders
    WHERE paid_date IS NULL
        AND ship_date IS NOT NULL
    ORDER BY customer_num
```

*Abfrage 2-28*

Diese SELECT-Anweisung liefert alle Datensätze, die in der Spalte **paid_date** einen NULL-Wert *und* in der Spalte **ship_date** keinen NULL-Wert enthalten.

```
order_num customer_num po_num     ship_date

     1004          106 8006       05/30/1993
     1007          117 278693     06/05/1993
     1012          117 278701     06/29/1993
     1016          119 PC6782     07/12/1993
```

*Ergebnis 2-28*

## Variable Textsuche

Bei CHARACTER-Feldern können Sie nach Zeichenkettenausschnitten suchen. Sie können die Schlüsselwörter LIKE und MATCHES verwenden, um Abfragen nach variablem Text durchzuführen. Fügen Sie das Schlüsselwort NOT hinzu, wenn die Bedingung umgekehrt werden soll. Das Schlüsselwort LIKE gehört zum ANSI-Standard, MATCHES dagegen ist eine Erweiterung von Informix.

Im Rahmen von LIKE oder MATCHES können Zeichenketten die folgenden in Bild 2-3 aufgelisteten Jokerzeichen enthalten.

| Symbol | Bedeutung |
|---|---|
| LIKE | |
| % | ersetzt kein, ein oder mehrere Zeichen |
| _ | ersetzt ein einziges Zeichen |
| \ | entwertet das folgende Sonderzeichen |
| MATCHES | |
| * | ersetzt kein, ein oder mehrere Zeichen |
| ? | ersetzt ein einziges Zeichen (Ausnahme: NULL) |
| [ ] | ersetzt ein einziges Zeichen oder einen Wertebereich |
| \ | entwertet das folgende Sonderzeichen |

*Bild 2-3* ***Jokerzeichen, die bei LIKE und MATCHES verwendet werden***

Bei LIKE oder MATCHES können keine TEXT- oder BYTE-Spalten angegeben werden.

## Strenger Textvergleich

Die folgenden Beispiele enthalten WHERE-Klauseln, die mit den Schlüsselwörtern LIKE oder MATCHES oder dem Vergleichsoperator (=) nach genau übereinstimmendem Text suchen. Im Gegensatz zu früheren Beispielen zeigen sie, wie man eine *externe* Tabelle in einer ANSI-konformen Datenbank abfragt.

Eine externe Tabelle ist eine Tabelle, die nicht zur aktuellen Datenbank gehört. Sie können nur auf diejenigen externen Tabellen zugreifen, die zu einer ANSI-konformen Datenbank gehören.

Während in diesem Kapitel bisher die Beispieldatenbank **stores6** verwendet wurde, ist im folgenden Beispiel in der FROM-Klausel die Tabelle **manatee** angegeben. Diese Tabelle wurde vom Benutzer **bubba** erstellt. Sie gehört zu der Datenbank **syzygy**, die ANSI-konform ist. Weitere Informationen darüber, wie externe Tabellen definiert werden, erhalten Sie im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

```
SELECT * FROM syzygy:bubba.manatee
    WHERE description = 'helmet'
    ORDER BY mfg_code

SELECT * FROM syzygy:bubba.manatee
    WHERE description LIKE 'helmet'
    ORDER BY mfg_code

SELECT * FROM syzygy:bubba.manatee
    WHERE description MATCHES 'helmet'
    ORDER BY mfg_code
```

*Abfrage 2-29*

Jede dieser SELECT-Anweisungen ruft all diejenigen Datensätze ab, die in der Spalte **description** das Wort `helmet` enthalten:

```
stock_no mfg_code  description     unit_price unit unit_type

     991 ANT       helmet            $222.00 case 4/case
     991 BKE       helmet            $269.00 case 4/case
     991 JSK       helmet            $311.00 each 4/case
     991 PRM       helmet            $234.00 case 4/case
     991 SHR       helmet            $245.00 case 4/case
```

*Ergebnis 2-29*

## Jokerzeichen, die einzelne Zeichen ersetzen

Die Anweisungen im folgenden Beispiel zeigen die Verwendung von Jokerzeichen, die in der WHERE-Klausel einzelne Zeichen ersetzen. Sie zeigen, wie eine externe Tabelle abgefragt werden kann. Die angegebene Tabelle **stock** befindet sich in der externen Datenbank **sloth**, die von einem separaten Datenbankserver mit dem Namen **meerkat** verwaltet wird.

Detaillierte Informationen über externe Tabellen, externe Datenbanken und Netze erhalten Sie in Kapitel 12 "Datenbankserver und Netzwerke" und im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

```
SELECT * FROM sloth@meerkat:stock
    WHERE manu_code LIKE '_R_'
        AND unit_price >= 100
    ORDER BY description, unit_price

SELECT * FROM sloth@meerkat:stock
    WHERE manu_code MATCHES '?R?'
        AND unit_price >= 100
    ORDER BY description, unit_price
```

*Abfrage 2-30*

Jede der SELECT-Anweisungen in diesem Beispiel rufen nur diejenigen Datensätze ab, bei denen der mittlere Buchstabe der Spalte **manu_code** ein R ist:

```
stock_num manu_code description     unit_price unit unit_descr

      205 HRO       3 golf balls       $312.00 case 24/case
        2 HRO       baseball           $126.00 case 24/case
        1 HRO       baseball gloves    $250.00 case 10 gloves/case
        7 HRO       basketball         $600.00 case 24/case
      102 PRC       bicycle brakes     $480.00 case 4 sets/case
      114 PRC       bicycle gloves     $120.00 case 10 pairs/case
        4 HRO       football           $480.00 case 24/case
      110 PRC       helmet             $236.00 case 4/case
      110 HRO       helmet             $260.00 case 4/case
      307 PRC       infant jogger      $250.00 each each
      306 PRC       tandem adapter     $160.00 each each
      308 PRC       twin jogger        $280.00 each each
      304 HRO       watch              $280.00 box  10/box
```

*Ergebnis 2-30*

Die Vergleichsmuster "_R_" (bei LIKE) oder "?R?" (bei MATCHES) bezeichnen, von links nach rechts, die folgenden Bestandteile:

- ein beliebiges Zeichen
- den Buchstaben R
- ein beliebiges Zeichen

## Die WHERE-Klausel mit Jokern für einzelne Zeichen einschränken

Die folgende Anweisung wählt nur diejenigen Datensätze aus, bei denen das manu_code mit einem Buchstaben zwischen A und H beginnt:

```
SELECT * FROM stock
    WHERE manu_code MATCHES '[A-H]*'

ORDER BY description, manu_code, unit_price
```

*Abfrage 2-31*

Die Anweisung liefert folgende Datensätze:

```
stock_num manu_code description     unit_price unit unit_descr

      205 ANZ       3 golf balls       $312.00 case 24/case
      205 HRO       3 golf balls       $312.00 case 24/case
        2 HRO       baseball           $126.00 case 24/case
        3 HSK       baseball bat       $240.00 case 12/case
        1 HRO       baseball gloves    $250.00 case 10 gloves/case
        1 HSK       baseball gloves    $800.00 case 10 gloves/case
        7 HRO       basketball         $600.00 case 24/case
      .
      .
      .
      110 ANZ       helmet             $244.00 case 4/case
      110 HRO       helmet             $260.00 case 4/case
      110 HSK       helmet             $308.00 case 4/case
      .
      .
      .
      301 ANZ       running shoes       $95.00 each each
      301 HRO       running shoes       $42.50 each each
      313 ANZ       swim cap            $60.00 box  12/box
        6 ANZ       tennis ball         $48.00 case 24 cans/case
        5 ANZ       tennis racquet      $19.80 each each
        8 ANZ       volleyball         $840.00 case 24/case
        9 ANZ       volleyball net      $20.00 each each
      304 ANZ       watch              $170.00 box  10/box
      304 HRO       watch              $280.00 box  10/box
```

*Ergebnis 2-31*

Die Prüfung "[A-H]" bezeichnet jeden Buchstaben zwischen A und H, jeweils inklusive. Für das Schlüsselwort LIKE gibt es kein gleichbedeutendes Jokerzeichen.

### Die WHERE-Klausel mit Jokerzeichen für variable Texte

In den Anweisungen dieses Beispiels wird am Ende einer Zeichenkette ein Jokerzeichen verwendet. Damit werden all die Datensätze abgerufen, bei denen die Werte in der Spalte **description** mit den Zeichen `bicycle` beginnen.

```
SELECT * FROM stock
    WHERE description LIKE 'bicycle%'
    ORDER BY description, manu_code

SELECT * FROM stock
    WHERE description MATCHES 'bicycle*'
    ORDER BY description, manu_code
```

*Abfrage 2-32*

Beide SELECT-Anweisungen liefern diese Datensätze:

```
stock_num manu_code description     unit_price unit unit_descr

      102 PRC       bicycle brakes     $480.00 case 4 sets/case
      102 SHM       bicycle brakes     $220.00 case 4 sets/case
      114 PRC       bicycle gloves     $120.00 case 10 pairs/case
      107 PRC       bicycle saddle      $70.00 pair pair
      106 PRC       bicycle stem        $23.00 each each
      101 PRC       bicycle tires       $88.00 box  4/box
      101 SHM       bicycle tires       $68.00 box  4/box
      105 PRC       bicycle wheels      $53.00 pair pair
      105 SHM       bicycle wheels      $80.00 pair pair
```

*Ergebnis 2-32*

Die Vergleichsmuster "`bicycle%`" oder "`bicycle*`" treffen auf die Zeichenkette `bicycle` zu, gefolgt von keinem, einem oder mehreren Zeichen. Das Vergleichsmuster paßt auf `bicycle stem`, wobei `stem` dem Jokerzeichen entspricht. Es paßt auch nur auf die Zeichen `bicycle`.

Die SELECT-Anweisung in der nächsten Abfrage schränkt durch die Angabe einer weiteren Vergleichsbedingung die Suche weiter ein. Diese Bedingung schließt Datensätze aus, deren **manu_code** gleich `PRC` ist.

```
SELECT * FROM stock
    WHERE description LIKE '%bicycle%'
        AND manu_code NOT LIKE 'PRC'
    ORDER BY description, manu_code
```

*Abfrage 2-33*

Die SELECT-Anweisung ruft nur folgende Datensätze ab:

```
stock_num manu_code description     unit_price unit unit_descr

      102 SHM       bicycle brakes     $220.00 case 4 sets/case
      101 SHM       bicycle tires       $68.00 box  4/box
      105 SHM       bicycle wheels      $80.00 pair pair
```

*Ergebnis 2-33*

Wenn Sie aus einer großen Tabelle selektieren und am Anfang eines Vergleichsmusters ein Jokerzeichen verwenden (z. B. "`%cycle`") , dauert die Ausführung der Abfrage meistens länger. Der Grund dafür ist, daß die Indizes nicht zur Wirkung kommen können und jeder einzelne Datensatz durchsucht werden muß.

## MATCHES und Native Language Support

Wie schon auf Seite 2-25 gesagt, können Sie mit den Umgebungsvariablen DBNLS, LANG und LC_COLLATE die Sortierfolge landessprachlicher Sonderzeichen beeinflussen.

In der folgenden Abfrage wird eine SELECT-Anweisung inklusive der Klauseln ORDER BY und WHERE verwendet, um Datensätze zu finden, deren Einträge in einem Feld des Typs NCHAR innerhalb eines bestimmten Bereiches liegen. Die NLS-Tabelle **abonnés** wird getrennt mit der Beispieldatenbank ausgeliefert. Die Abfrage selbst wird in der Datei **sel_nls2.sql** zusammen mit NLS-fähigem **DB-Access** ausgeliefert.

```
SELECT numéro,nom,prénom
    FROM abonnés
    WHERE nom MATCHES '[E-P]*'
    ORDER BY nom;
```

*Abfrage 2-34*

Im folgenden sehen Sie zwei Ergebnistabellen, die auf der identischen SELECT-Anweisung basieren; sie wurden allerdings mit jeweils verschiedenen Einträgen in den NLS-Umgebungsvariablen ausgeführt.

```
numéro      nom             prénom

13607       Hammer          Gerhard
13602       Hämmerle        Greta
13604       LaForêt         Jean-Noël
13610       LeMaître        Héloïse
13613       Llanero         Gloria Dolores
13603       Montaña         José Antonio
13611       Oatfield        Emily
```

*Ergebnis 2-34a*

In der ersten Ergebnistabelle sind die Datensätze von Étaix, Ötker und Øverst nicht enthalten, da nach der Standard-ASCII Sortierfolge die Anfangsbuchstaben dieser Namen nicht im Bereich E-P der Spalte **nom** liegen, der mit dem Schlüsselwort MATCHES eingegeben wurde.

```
numéro      nom             prénom

13608       Étaix           Émile
13607       Hammer          Gerhard
13602       Hämmerle        Greta
13604       LaForêt         Jean-Noël
13610       LeMaître        Héloïse
13613       Llanero         Gloria Dolores
13603       Montaña         José Antonio
13611       Oatfield        Emily
13605       Ötker           Hans-Jürgen
13614       Øverst          Per-Anders
```

*Ergebnis 2-34b*

In dieser Ergebnistabelle dagegen sind die Sätze mit den Namen Étaix, Ötker und Øverst in der Ergebnistabelle enthalten, da durch die entsprechenden Einstellungen in den NLS-Umgebungsvariablen die Anfangsbuchstaben der drei Namen im Bereich E-P der Spalte **nom** einsortiert werden.

## Sonderzeichen in Vergleichsmustern

Das Schlüsselwort ESCAPE wird im Zusammenhang mit LIKE und MATCHES verwendet, um ein Sonderzeichen vor einer falschen Interpretation als Jokerzeichen zu schützen.

```
SELECT * FROM cust_calls
    WHERE res_descr LIKE '%!%%' ESCAPE '!'
```

*Abfrage 2-35*

Das Schlüsselwort ESCAPE benennt ein *Entwertungszeichen* (in diesem Beispiel das Ausrufezeichen). Das Entwertungszeichen schützt das folgende Zeichen, so daß es als Teil der Daten und nicht als Sonderzeichen interpretiert wird. In diesem Beispiel bewirkt das Entwertungszeichen, daß das mittlere Prozentzeichen als Teil der Daten behandelt wird. Man kann in der Spalte **res_descr** auch nach einem Prozentzeichen suchen, indem man das Jokerzeichen % in Verbindung mit LIKE verwendet. Dies geschieht durch Verwendung des Schlüsselwortes ESCAPE. Die Abfrage liefert den folgenden Datensatz:

```
customer_num    116
call_dtime      1992-12-21 11:24
user_id         mannyn
call_code       I
call_descr      Second complaint from this customer! Received
                two cases right-handed outfielder gloves
                (1 HRO) instead of one case lefties.
res_dtime       1992-12-27 08:19
res_descr       Memo to shipping (Ava Brown) to send case of
                left-handed gloves, pick up wrong case; memo
                to billing requesting 5% discount to placate
                customer due to second offense and lateness
                of resolution because of holiday
```

*Ergebnis 2-35*

### Feldausschnitt in der WHERE-Klausel

In der WHERE-Klausel einer SELECT-Anweisung können Sie *Feldausschnitte* verwenden, um einen Zeichen- oder Zahlenbereich einer Spalte anzugeben:

```
SELECT catalog_num, stock_num, manu_code, cat_advert,
       cat_descr
    FROM catalog
    WHERE cat_advert[1,4] = 'High'
```

*Abfrage 2-36*

Der Ausschnitt `[1,4]` bewirkt, daß diejenigen Datensätze abgerufen werden, bei denen die ersten vier Buchstaben der Spalte **cat_advert** die Buchstabenkombination `High` sind.

```
 catalog_num  10004
 stock_num    2
 manu_code    HRO
 cat_advert   Highest Quality Ball Available, from
              Hand-Stitching to the Robinson Signature
 cat_descr
Jackie Robinson signature ball. Highest professional quality, used by
National League.

 catalog_num  10005
 stock_num    3
 manu_code    HSK
 cat_advert   High-Technology Design Expands the Sweet Spot
 cat_descr
Pro-style wood. Available in sizes: 31, 32, 33, 34, 35.

 catalog_num  10008
 stock_num    4
 manu_code    HRO
 cat_advert   Highest Quality Football for High School and
              Collegiate Competitions
 cat_descr
NFL-style, pigskin.


 catalog_num  10012
 stock_num    6
 manu_code    SMT
 cat_advert   High-Visibility Tennis, Day or Night
 cat_descr
Soft yellow color for easy visibility in sunlight or
artificial light.

 catalog_num  10043
 stock_num    202
 manu_code    KAR
 cat_advert   High-Quality Woods Appropriate for High School
              Competitions or Serious Amateurs
 cat_descr
Full set of woods designed for precision control and
power performance.

 catalog_num  10045
 stock_num    204
 manu_code    KAR
 cat_advert   High-Quality Beginning Set of Irons
              Appropriate for High School Competitions
 cat_descr
Ideally balanced for optimum control. Nylon covered shaft.

 catalog_num  10068
 stock_num    310
 manu_code    ANZ
 cat_advert   High-Quality Kickboard
 cat_descr
White. Standard size.
```

*Ergebnis 2-36*

# Ausdrücke und berechnete Werte

Spalten müssen nicht ausschließlich durch Angabe des Namens ausgewählt werden. Sie können mit der SELECT-Klausel einer SELECT-Anweisung auch Berechnungen mit Spaltenwerten durchführen und Daten anzeigen lassen, die aus dem Inhalt einer oder mehrerer Spalten *berechnet* wurden. Dies können Sie tun, indem Sie einen *Ausdruck* in der Auswahlliste angeben.

Ein Ausdruck besteht aus einem Spaltennamen, einer Konstanten, einer Zeichenkette in Anführungszeichen, einem Schlüsselwort oder einer Kombination dieser Bestandteile, jeweils verbunden durch Operatoren. Ein Ausdruck kann auch Hostvariablen (Programmdaten) enthalten, wenn die SELECT-Anweisung in ein Programm eingebettet ist.

Nähere Informationen zum Syntaxelement "Ausdruck" finden Sie in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

## Arithmetische Ausdrücke

Ein arithmetischer Ausdruck enthält mindestens einen der in Bild 2-4 aufgelisteten *arithmetischen Operatoren* und ergibt eine Zahl. Spalten der Datentypen BYTE und TEXT dürfen nicht in arithmetischen Ausdrücken verwendet werden.

| Operator | Operation |
|---|---|
| + | Addition |
| - | Subtraktion |
| * | Multiplikation |
| / | Division |

*Bild 2-4* ***Arithmetische Operatoren, die in arithmetischen Ausdrücken verwendet werden***

Mit Operationen dieser Art können Sie Ergebnisse von Berechnungen erhalten, ohne die Daten in einer Datenbank zu verändern. Wenn Sie die veränderten Daten in einer temporären Tabelle sichern wollen, fügen Sie die INTO TEMP-Klausel hinzu.Die temporäre Tabelle können Sie für weitere Abfragen, Berechnungen oder kurzfristig benötigte Listen verwenden.

In der folgenden Abfrage wird eine Verkaufssteuer in Höhe von 7% berechnet, indem der Wert der Spalte **unit_price** für Artikel, deren Wert über $400 liegt, mit dem Faktor 1,07 multipliziert wird. Die Datenbank selbst wird dabei nicht aktualisiert:

```
SELECT stock_num, description, unit, unit_descr,
       unit_price, unit_price * 1.07
    FROM stock
    WHERE unit_price >= 400
```

*Abfrage 2-37*

Wenn Sie mit **DB-Access** oder **INFORMIX-SQL** arbeiten, wird das Ergebnis in einer Spalte mit der Überschrift `expression` angezeigt.

```
stock_num description     unit unit_descr      unit_price   (expression)

        1 baseball gloves case 10 gloves/case     $800.00    $856.0000
        1 baseball gloves case 10 gloves/case     $450.00    $481.5000
        4 football        case 24/case           $960.00   $1027.2000
        4 football        case 24/case           $480.00    $513.6000
        7 basketball      case 24/case           $600.00    $642.0000
        8 volleyball      case 24/case           $840.00    $898.8000
      102 bicycle brakes  case 4 sets/case        $480.00    $513.6000
      111 10-spd, assmbld each each              $499.99    $534.9893
      112 12-spd, assmbld each each              $549.00    $587.4300
      113 18-spd, assmbld each each              $685.90    $733.9130
      203 irons/wedge     case 2 ets/case        $670.00    $716.9000
```

*Ergebnis 2-37*

Abfrage 2-38 berechnet bei Aufträgen, deren Bestellmenge (**quantity**) unter 5 liegt, einen Zuschlag von $6.50:.

```
SELECT item_num, order_num, quantity,
       total_price, total_price + 6.50
    FROM items
    WHERE quantity < 5
```

*Abfrage 2-38*

Wenn Sie mit **DB-Access** oder **INFORMIX-SQL** arbeiten, wird das Ergebnis in einer Spalte mit der Überschrift `expression` angezeigt.

| item_num | order_num | quantity | total_price | (expression) |
|---|---|---|---|---|
| 1 | 1001 | 1 | $250.00 | $256.50 |
| 1 | 1002 | 1 | $960.00 | $966.50 |
| 2 | 1002 | 1 | $240.00 | $246.50 |
| 1 | 1003 | 1 | $20.00 | $26.50 |
| 2 | 1003 | 1 | $840.00 | $846.50 |
| 1 | 1004 | 1 | $250.00 | $256.50 |
| 2 | 1004 | 1 | $126.00 | $132.50 |
| 3 | 1004 | 1 | $240.00 | $246.50 |
| 4 | 1004 | 1 | $800.00 | $806.50 |
| . | | | | |
| . | | | | |
| . | | | | |
| 1 | 1021 | 2 | $75.00 | $81.50 |
| 2 | 1021 | 3 | $225.00 | $231.50 |
| 3 | 1021 | 3 | $690.00 | $696.50 |
| 4 | 1021 | 2 | $624.00 | $630.50 |
| 1 | 1022 | 1 | $40.00 | $46.50 |
| 2 | 1022 | 2 | $96.00 | $102.50 |
| 3 | 1022 | 2 | $96.00 | $102.50 |
| 1 | 1023 | 2 | $40.00 | $46.50 |
| 2 | 1023 | 2 | $116.00 | $122.50 |
| 3 | 1023 | 1 | $80.00 | $86.50 |
| 4 | 1023 | 1 | $228.00 | $234.50 |
| 5 | 1023 | 1 | $170.00 | $176.50 |
| 6 | 1023 | 1 | $190.00 | $196.50 |

*Ergebnis 2-38*

Abfrage 2-39 berechnet die Zeitspanne von der Erteilung eines Kundenauftrags (**call_dtime**) bis zur Erledigung des Auftrags (**res_dtime**) in Tagen, Stunden und Minuten:

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime
    FROM cust_calls
    ORDER BY user_id
```

*Abfrage 2-39*

Wenn Sie mit **DB-Access** oder **INFORMIX-SQL** arbeiten, wird das Ergebnis in einer Spalte mit der Überschrift `expression` angezeigt.

```
customer_num user_id          call_code call_dtime          (expression)

         116 mannyn           I         1992-12-21 11:24       5 20:55
         116 mannyn           I         1992-11-28 13:34       0 03:13
         106 maryj            D         1993-06-12 08:20       0 00:05
         121 maryj            O         1993-07-10 14:05       0 00:01
         127 maryj            I         1993-07-31 14:30
         110 richc            L         1993-07-07 10:24       0 00:06
         119 richc            B         1993-07-01 15:00       0 17:21
```

*Ergebnis 2-39*

## Verwendung von Spaltenüberschriften

Sie können einer Spalte mit berechneten Daten auch eine *Spaltenüberschrift* zuweisen, die die Standard-Spaltenüberschrift `expression` ersetzt.. Auch in der nun folgenden Abfrage werden berechnete Werte dargestellt, aber die Spalten mit den berechneten Werten wird nun mit der aussagekräftigen Spaltenüberschrift `taxed` versehen.

```
SELECT stock_num, description, unit, unit_descr,
       unit_price, unit_price * 1.07 taxed
    FROM stock
    WHERE unit_price >= 400
```

*Abfrage 2-40*

In dieser Abfrage wurde dem Ausdruck, der das Ergebnis der arithmetischen Operation `unit_price * 1.07` anzeigt, in der Auswahlliste die Spaltenüberschrift `taxed` zugeordnet:

```
stock_num description     unit unit_descr      unit_price       taxed

        1 baseball gloves case 10 gloves/case     $800.00    $856.0000
        1 baseball gloves case 10 gloves/case     $450.00    $481.5000
        4 football        case 24/case            $960.00   $1027.2000
        4 football        case 24/case            $480.00    $513.6000
        7 basketball      case 24/case            $600.00    $642.0000
        8 volleyball      case 24/case            $840.00    $898.8000
      102 bicycle brakes  case 4 sets/case        $480.00    $513.6000
      111 10-spd, assmbld each each               $499.99    $534.9893
      112 12-spd, assmbld each each               $549.00    $587.4300
      113 18-spd, assmbld each each               $685.90    $733.9130
      203 irons/wedge     case 2 sets/case        $670.00    $716.9000
```

*Ergebnis 2-40*

In der folgenden SELECT-Anweisung wurde für die Spalte, die das Ergebnis der Berechnung `total_price + 6.50` anzeigt, die Überschrift `surcharge` festgelegt:

```
SELECT item_num, order_num, quantity,
       total_price, total_price + 6.50 surcharge
    FROM items
    WHERE quantity < 5
```

*Abfrage 2-41*

Bei der Ausgabe trägt die Spalte die Überschrift `surcharge`:

```
item_num   order_num quantity total_price    surcharge
      .
      .
      .
       2        1013        1      $36.00       $42.50
       3        1013        1      $48.00       $54.50
       4        1013        2      $40.00       $46.50
       1        1014        1     $960.00      $966.50
       2        1014        1     $480.00      $486.50
       1        1015        1     $450.00      $456.50
       1        1016        2     $136.00      $142.50
       2        1016        3      $90.00       $96.50
       3        1016        1     $308.00      $314.50
       4        1016        1     $120.00      $126.50
       1        1017        4     $150.00      $156.50
       2        1017        1     $230.00      $236.50
       .
       .
       .
```

*Ergebnis 2-41*

Die SELECT-Anweisung in der nächsten Abfrage weist der Spalte, die das Ergebnis der Subtraktion der DATETIME-Spalte **call_dtime** von der DATETIME-Spalte **res_dtime** anzeigt, die Überschrift `span` zu:

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime span
    FROM cust_calls
    ORDER BY  user_id
```

*Abfrage 2-42*

Bei der Ausgabe trägt die Spalte die Überschrift span:

```
customer_num user_id          call_code call_dtime              span

         116 mannyn           I         1992-12-21 11:24        5 20:55
         116 mannyn           I         1992-11-28 13:34        0 03:13
         106 maryj            D         1993-06-12 08:20        0 00:05
         121 maryj            O         1993-07-10 14:05        0 00:01
         127 maryj            I         1993-07-31 14:30
         110 richc            L         1993-07-07 10:24        0 00:06
         119 richc            B         1993-07-01 15:00        0 17:21
```

*Ergebnis 2-42*

## Sortierung nach berechneter Spalten

Wenn Sie die Klausel ORDER BY in einem Ausdruck verwenden wollen, können Sie hierfür entweder die Spaltenüberschrift angeben, die dem Ausdruck zugewiesen ist, oder eine Zahl (integer):

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime span
    FROM cust_calls
    ORDER BY span
```

*Abfrage 2-43*

Abfrage 2-43 ruft dieselben Datensätze aus der Tabelle **cust_calls** ab wie die vorhergehende Abfrage. Hier bewirkt jedoch die ORDER BY-Klausel, daß die Daten in aufsteigender Reihenfolge nach den berechneten Werten der Spalte span angezeigt werden:

```
customer_num user_id          call_code call_dtime              span

         127 maryj            I         1993-07-31 14:30
         121 maryj            O         1993-07-10 14:05        0 00:01
         106 maryj            D         1993-06-12 08:20        0 00:05
         110 richc            L         1993-07-07 10:24        0 00:06
         116 mannyn           I         1992-11-28 13:34        0 03:13
         119 richc            B         1993-07-01 15:00        0 17:21
         116 mannyn           I         1992-12-21 11:24        5 20:55
```

*Ergebnis 2-43*

Die nächste Anweisung verwendet eine Zahl, um das Ergebnis der Berechnung `res_dtime - call_dtime` in der ORDER BY -Klausel anzugeben. Sie ruft dieselben Datensätze ab wie die vorherige Abfrage:

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime span
    FROM cust_calls
    ORDER BY 5
```

*Abfrage 2-44*

# Funktionen in SELECT-Anweisungen

Zusätzlich zu Spaltennamen und Operatoren kann ein Ausdruck auch eine oder mehrere Funktionen enthalten.

Zu den unterstützten Ausdrücke gehören Mengen-, Funktionen- (inklusive der arithmetischen Funktionen), Konstanten- und Spaltenausdrücke. Eine ausführliche Beschreibung dieser Ausdrücke finden Sie in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

## Mengenfunktionen

Die Mengenfunktionen COUNT, AVG, MAX, MIN und SUM liefern Daten über Datensätze, nicht aber die Datensätze selbst. Diese Funktionen können nicht bei TEXT- oder BYTE-Spalten verwendet werden.

Mengenfunktionen werden häufig verwendet, um Werte von Datensatzgruppen einer Tabelle zusammenzufassen. Diese Verwendung wird in Kapitel 3 "Komplexe SELECT-Anweisungen" erläutert. Wenn Sie eine Mengenfunktion auf eine komplette Tabelle anwenden, enthält das Ergebnis nur einen einzigen Datensatz, in dem alle ausgewählten Datensätze berücksichtigt sind.

Die folgende Abfrage zählt die Datensätze der Tabelle **stock**:

```
SELECT COUNT(*)
    FROM stock
```

*Abfrage 2-45*

Als Ergebnis wird die Gesamtanzahl der Datensätze der Tabelle **stock** angezeigt:

```
        (count(*))

                73
```

*Ergebnis 2-45*

Die nächste Abfrage enthält die WHERE-Klausel, um bestimmte Datensätze der Tabelle **stock** zu zählen, in diesem Fall nur die Sätze, deren **manu_code** SHM lautet:

```
SELECT COUNT (*)
    FROM stock
    WHERE manu_code = 'SHM'
```

*Abfrage 2-46*

Als Ergebnis wird die ermittelte Anzahl der Datensätze angezeigt:

```
        (count(*))

                16
```

*Ergebnis 2-46*

Wenn Sie in einer SELECT-Anweisung das Schlüsselwort DISTINCT (oder dessen Synonym UNIQUE) und einen Spaltennamen angeben, können Sie die Anzahl der unterschiedlichen Hersteller-Codes der Tabelle **stock** ermitteln.

```
SELECT COUNT (DISTINCT manu_code)
    FROM stock
```

*Abfrage 2-47*

Als Ergebnis wird die ermittelte Anzahl der Datensätze angezeigt:

```
                  (count)

                 9
```

*Ergebnis 2-47*

Die folgende SELECT-Anweisung berechnet aus allen Datensätzen der Tabelle **stock** den Durchschnitt von **unit_price**.

```
SELECT AVG (unit_price)
    FROM stock
```

*Abfrage 2-48*

Als Ergebnis wird die ermittelte Zahl angezeigt:

```
          (avg)
           $197.14
```

*Ergebnis 2-48*

Die folgende SELECT-Anweisung berechnet den Durchschnitt von `unit_price` nur aus denjenigen Sätzen der Tabelle **stock**, deren **manu_code** `SHM` ist.

```
SELECT AVG (unit_price)
    FROM stock
    WHERE manu_code = 'SHM'
```

*Abfrage 2-49*

Als Ergebnis wird die ermittelte Zahl angezeigt:

```
          (avg)
          $204.93
```

*Ergebnis 2-49*

Die Mengenfunktionen in einer SELECT-Anweisung können auch kombiniert werden, wie Abfrage 2-50 zeigt:

```
SELECT MAX (ship_charge), MIN (ship_charge)
    FROM orders
```

*Abfrage 2-50*

Diese SELECT-Anweisung sucht in der Tabelle **orders** den größten und kleinsten Wert der Spalte **ship_charge** und zeigt beide Werte an.

```
     (max)      (min)
     $25.20      $5.00
```

*Ergebnis 2-50*

Sie können Ausdrücke in Funktionen verwenden, und Sie können für die Ergebnisse auch Spaltenüberschriften vergeben:

```
SELECT MAX (res_dtime - call_dtime) maximum,
    MIN (res_dtime - call_dtime) minimum,
    AVG (res_dtime - call_dtime) average
    FROM cust_calls
```

*Abfrage 2-51*

Abfrage 2-51 ermittelt die längste, kürzeste und durchschnittliche Zeitdauer (in Tagen, Stunden und Minuten), die zwischen Auftragserteilung und -ausführung liegt; sie zeigt diese Werte an und weist den berechneten Werten eine aussagekräftige Spaltenüberschrift zu:

```
maximum          minimum          average

5 20:55          0 00:01          1 02:56
```

*Ergebnis 2-51*

Die folgende SELECT-Anweisung berechnet das Gesamt-Ladegewicht (**ship_weight**) der Aufträge, die am 13. Juli 1993 verladen wurden:

```
SELECT SUM (ship_weight)
    FROM orders
    WHERE ship_date = '07/13/1993'
```

*Abfrage 2-52*

Die Anweisung zeigt auch das Ergebnis dieser Berechnung an:

```
(sum)
 1 row(s) retrieved
```

*Ergebnis 2-52*

## Zeitfunktionen

Die Zeitfunktionen DAY, MDY, MONTH, WEEKDAY, YEAR und DATE können sowohl in der SELECT-Klausel als auch in der WHERE-Klausel einer Abfrage verwendet werden. Die Funktionen liefern Werte, die den Ausdrücken oder Argumenten entsprechen, mit denen eine Funktion aufgerufen wurde. Die

Funktion CURRENT wird dazu verwendet, das aktuelle Datum und die Uhrzeit auszugeben. Die Funktion EXTEND stellt die Genauigkeit eines DATE- oder DATETIME-Wertes ein.

### Die Funktionen DAY und CURRENT

Die folgende SELECT-Anweisung liefert aus den Spalten **call_dtime** und **res_dtime** die Werte für "Tag im jeweiligen Monat":

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
    FROM cust_calls
```

*Abfrage 2-53*

Das Ergebnis der Berechnung wird in zwei Spalten mit dem Namen `expression` angezeigt:

```
customer_num (expression) (expression)

         106            12            12
         110             7             7
         119             1             2
         121            10            10
         127            31
         116            28            28
         116            21            27
```

*Ergebnis 2-53*

Die nächste SELECT-Anweisung verwendet die Funktionen DAY und CURRENT, um die Werte einer Spalte mit dem aktuellen Tag des Monats zu vergleichen:

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
    FROM cust_calls
    WHERE DAY (call_dtime) < DAY (CURRENT)
```

*Abfrage 2-54*

Die Anweisung wählt nur diejenigen Datensätze aus, bei denen der Wert kleiner ist als das aktuelle Datum.

```
customer_num (expression) (expression)

         106           12           12
         110            7            7
         119            1            2
         121           10           10
```

*Ergebnis 2-54*

```
SELECT customer_num, call_code, call_descr
    FROM cust_calls
    WHERE call_dtime < CURRENT YEAR TO DAY
```

*Abfrage 2-55*

Diese Abfrage zeigt eine weitere Verwendung der Funktion CURRENT. Sie wählen wiederum nur die Datensätze aus, bei denen der Tag vor dem aktuellen Tag liegt.

```
customer_num  106
call_code     D
 call_descr   Order was received, but two of the cans of ANZ tennis
              balls within the case were empty

customer_num  116
call_code     I
call_descr    Received plain white swim caps (313 ANZ) instead of
              navy with team logo (313 SHM)

customer_num  116
call_code     I
call_descr    Second complaint from this customer! Received
              two cases right-handed outfielder gloves (1 HRO) instead of
              one case lefties.
```

*Ergebnis 2-55*

### Die Funktion MONTH

Die folgende Abfrage verwendet die Funktion MONTH, um den Monat zu ermitteln, in dem ein Auftrag erteilt und erledigt wurde. Es wird jedoch nicht nach Jahren unterschieden:

```
SELECT customer_num,
    MONTH (call_dtime) call_month,
    MONTH (res_dtime) res_month
    FROM cust_calls
```

*Abfrage 2-56*

Für jede Ergebnisspalte wird eine Spaltenüberschrift vergeben:

| customer_num | call_month | res_month |
|---|---|---|
| 106 | 6 | 6 |
| 110 | 7 | 7 |
| 119 | 7 | 7 |
| 121 | 7 | 7 |
| 127 | 7 | |
| 116 | 11 | 11 |
| 116 | 12 | 12 |

*Ergebnis 2-56*

```
SELECT customer_num,
    MONTH (call_dtime) called,
    MONTH (res_dtime) resolved
    FROM cust_calls
    WHERE DAY (res_dtime) < DAY (CURRENT)
```

*Abfrage 2-57*

Diese SELECT-Anweisung verwendet neben der Funktion MONTH auch die Funktionen DAY und CURRENT, um zu zeigen, in welchem Monat ein Auftrag erteilt und erledigt wurde, falls DAY vor dem aktuellen Tag liegt.

```
customer_num       called     resolved

         106            6            6
         110            7            7
         119            7            7
         121            7            7
```

*Ergebnis 2-57*

## Die Funktion WEEKDAY

In Abfrage 2-58 wird die Funktion WEEKDAY verwendet, um aufzuzeigen, an welchem Wochentag Aufträge erteilt und erledigt wurden (0 bedeutet Sonntag, 1 bedeutet Montag, usw.).

```
SELECT customer_num,
    WEEKDAY (call_dtime) called,
    WEEKDAY (res_dtime) resolved
    FROM cust_calls
    ORDER BY resolved
```

*Abfrage 2-58*

Für jede Ergebnisspalte wird eine Spaltenüberschrift vergeben:

```
customer_num called resolved

         127      3
         110      0        0
         119      1        2
         121      3        3
         116      3        3
         106      3        3
         116      5        4
```

*Ergebnis 2-58*

```
SELECT COUNT(*)
    FROM cust_calls
    WHERE WEEKDAY (call_dtime) IN (0,6)
```

*Abfrage 2-59*

In Abfrage 2-59 wird mit den Funktionen COUNT und WEEKDAY die Anzahl der Aufträge ermittelt, die an Wochenenden erteilt wurden. Diese Art einer Selektion kann Ihnen eine Vorstellung über die Struktur der Kundenaufträge vermitteln.

```
            (count(*))
                 4
```

*Ergebnis 2-59*

```
SELECT customer_num, call_code,
    YEAR (call_dtime) call_year,
    YEAR (res_dtime) res_year
    FROM cust_calls
    WHERE YEAR (call_dtime) < YEAR (TODAY)
```

*Abfrage 2-60*

Abfrage 2-60 ruft Datensätze ab, bei denen **call_dtime** vor dem Beginn des aktuellen Jahres liegt.

```
customer_num call_code call_year res_year

         116 I                1992     1992
         116 I                1992     1992
```

*Ergebnis 2-60*

## Formatierung von DATETIME-Werten

In Abfrage 2-61 begrenzt die Funktion EXTEND zwei DATETIME-Werte dadurch, daß nur ein bestimmter Teil der Felder angezeigt wird.

```
SELECT customer_num,
    EXTEND (call_dtime, month to minute) call_time,
    EXTEND (res_dtime, month to minute) res_time
    FROM cust_calls
    ORDER BY res_time
```

*Abfrage 2-61*

Das Ergebnis liefert aus den Spalten **call_dtime** und **res_dtime** den Wertebereich Monat bis Minute und gibt einen Hinweis auf den Arbeitsaufwand.

```
customer_num call_time   res_time

         127 07-31 14:30
         106 06-12 08:20 06-12 08:25
         119 07-01 15:00 07-02 08:21
         110 07-07 10:24 07-07 10:30
         121 07-10 14:05 07-10 14:06
         116 11-28 13:34 11-28 16:47
         116 12-21 11:24 12-27 08:19
```

*Ergebnis 2-61*

## Die Funktion DATE

Abfrage 2-62 ruft nur die DATETIME-Werte ab, bei denen das Datum **call_dtime** größer ist als das Datum, das bei DATE angegeben ist.

```
SELECT customer_num, call_dtime, res_dtime
    FROM cust_calls
    WHERE call_dtime > DATE ('12/31/92')
```

*Abfrage 2-62*

Die Abfrage führt zu folgendem Ergebnis:

```
customer_num call_dtime       res_dtime

         106 1993-06-12 08:20 1993-06-12 08:25
         110 1993-07-07 10:24 1993-07-07 10:30
         119 1993-07-01 15:00 1993-07-02 08:21
         121 1993-07-10 14:05 1993-07-10 14:06
         127 1993-07-31 14:30
```

*Ergebnis 2-62*

In Abfrage 2-63 konvertiert die SELECT-Anweisung DATETIME-Werte in das DATE-Format. Die Werte, inklusive der Überschrift, werden nur angezeigt, wenn **call_dtime** größer oder gleich dem angegebenen Datum ist:

```
SELECT customer_num
    DATE (call_dtime) called,
    DATE (res_dtime) resolved
    FROM cust_calls
    WHERE call_dtime >= DATE ('1/1/93')
```

*Abfrage 2-63*

Die Abfrage führt zu folgendem Ergebnis:

```
  customer_num     called     resolved

           106 06/12/1993   06/12/1993
           110 07/07/1993   07/07/1993
           119 07/01/1993   07/02/1993
           121 07/10/1993   07/10/1993
           127 07/31/1993
```

*Ergebnis 2-63*

## Weitere Funktionen und Schlüsselwörter

Die Funktionen LENGTH, USER, CURRENT und TODAY können überall dort in einem SQL-Ausdruck benutzt werden, wo eine Konstante verwendet werden kann. Außerdem kann bei **INFORMIX-OnLine Dynamic Server** in einer SELECT-Anweisung das Schlüsselwort DBSERVERNAME angegeben werden, um den Namen des Datenbankservers der aktuelle Datenbank anzuzeigen.

Sie verwenden diese Funktionen und Schlüsselwörter, um einen Ausdruck auszuwählen, der nur Konstanten enthält oder einen Ausdruck, der Daten einer Spalte beinhaltet. Im ersten Fall ist das Ergebnis bei allen angegebenen Datensätzen gleich.

Weiterhin können Sie die Funktion HEX verwenden, um die hexadezimale Codierung eines Ausdrucks zu liefern; die Funktion ROUND, um gerundete Werte eines Ausdrucks zu liefern; die Funktion TRUNC, um Werte eines Ausdrucks abzuschneiden.

```
SELECT customer_num,
    LENGTH (fname) + LENGTH (lname) namelength
    FROM customer
    WHERE LENGTH (company) > 15
```

*Abfrage 2-64*

In Abfrage 2-64 berechnet die Funktion LENGTH die Anzahl der Zeichen, die in den Spalten **fname** und **lname** zusammen enthalten sind, und zwar für die Datensätze, bei denen der Wert der Spalte **company** länger als 15 Zeichen ist.

```
customer_num    namelength

         101            11
         105            13
         107            11
         112            14
         115            11
         118            10
         119            10
         120            10
         122            12
         124            11
         125            10
         126            12
         127            10
         128            11
```

*Ergebnis 2-64*

Wenn Sie mit dem interaktiven Editor von **DB-Access** oder **INFORMIX-SQL** arbeiten, ist diese Funktion wahrscheinlich nicht sehr sinnvoll. Die Funktion LENGTH kann jedoch wichtig sein, um die Länge der Zeichenketten für Programme und Listen zu ermitteln. LENGTH liefert die tatsächliche Länge einer CHARACTER oder VARCHAR-Zeichenkette und die vollständige Zeichenanzahl einer TEXT- oder BYTE-Zeichenkette.

Die Funktion USER kann Ihnen helfen, eine eingeschränkte View auf eine Tabelle zu definieren, der nur bestimmte Datensätze enthalten soll. Informationen darüber, wie eine View erstellt wird, erhalten Sie im Kapitel 11 "Zugriff auf Datenbanken regeln". Eine Beschreibung der Anweisungen GRANT und CREATE VIEW finden Sie im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

In der Abfrage 2-65a wird für die Tabelle **cust_calls** die Benutzerkennung des aktuellen Benutzers (Funktion USER) ermittelt:

```
SELECT USER from cust_calls
```

*Abfrage 2-65a*

In Abfrage 2-65b gibt den Namen der Benutzerkennung aus, der die Abfrage ausführt. Dies wird für jeden Datensatz der Tabelle wiederholt.

```
SELECT * FROM cust_calls
    WHERE user_id = USER
```

*Abfrage 2-65b*

Wenn der Name des momentanen Benutzers **richc** lautet, dann ruft Abfrage 2-65b nur die Datensätze aus der Tabelle **cust_calls** ab, die diesem Benutzer gehören.

```
customer_num  110
call_dtime    1993-07-07 10:24
user_id       richc
call_code     L
call_descr    Order placed one month ago (6/7) not received.
res_dtime     1993-07-07 10:30
res_descr     Checked with shipping (Ed Smith). Order sent
    yesterday- we were waiting for goods from ANZ. Next
    time will call with delay if necessary.

customer_num  119
call_dtime    1993-07-01 15:00
user_id       richc
call_code     B
call_descr    Bill does not reflect credit from previous order
res_dtime     1993-07-02 08:21
res_descr     Spoke with Jane Akant in Finance. She found the
    error and is sending new bill to customer
```

*Ergebnis 2-65*

```
SELECT * FROM orders
    WHERE order_date = TODAY
```

*Abfrage 2-66*

Wenn das heutige Systemdatum der 10. Juli 1993 ist, dann liefert die vorherige SELECT-Anweisung bei der Ausführung nur diesen einen Datensatz:

```
order_num       1018
order_date      07/10/1993
customer_num    121
ship_instruct   SW corner of Biltmore Mall
backlog         n
po_num          S22942
ship_date       07/13/1993
ship_weight     70.50
ship_charge     $20.00
paid_date       08/06/1993
```

*Ergebnis 2-66*

Bei **INFORMIX-OnLine Dynamic Server** können Sie in einer SELECT-Anweisung das Schlüsselwort DBSERVERNAME (oder sein Synonym SITENAME ) angeben, um den Namen des Datenbankservers zu ermitteln. Sie können DBSERVERNAME bei jeder Tabelle angeben. Dies gilt auch für eine Systemtabelle.

```
SELECT DBSERVERNAME server, tabid FROM systables
    WHERE tabid <= 4
```

*Abfrage 2-67*

In Abfrage 2-67 weisen Sie DBSERVERNAME die Überschrift **server** zu und selektieren die Spalte **tabid** aus der Systemtabelle **systables**. Diese Tabelle beschreibt Datenbank-Tabellen. **tabid** ist eine fortlaufende Zahl, die eine Tabelle eindeutig bezeichnet.

```
server          tabid

montague           1
montague           2
montague           3
montague           4
```

*Ergebnis 2-67*

Ohne die WHERE-Klausel, die die Werte in **tabid** begrenzt, würde der Name des Datenbankservers für jeden Datensatz der Tabelle **systables** wiederholt werden.

```
SELECT HEX (customer_num) hexnum, HEX (zipcode) hexzip,
    HEX (rowid) hexrow
    FROM CUSTOMER
```

*Abfrage 2-68*

```
hexnum     hexzip     hexrow

0x00000065 0x00016F86 0x00000001
0x00000066 0x00016FA5 0x00000002
0x00000067 0x0001705F 0x00000003
0x00000068 0x00016F4A 0x00000004
0x00000069 0x00016F46 0x00000005
0x0000006A 0x00016F6F 0x00000006
0x0000006B 0x00017060 0x00000007
0x0000006C 0x00016F6F 0x00000008
0x0000006D 0x00016F86 0x00000009
0x0000006E 0x00016F6E 0x0000000A
0x0000006F 0x00016F85 0x0000000B
0x00000070 0x00016F46 0x0000000C
0x00000071 0x00016F49 0x0000000D
0x00000072 0x00016F6E 0x0000000E
0x00000073 0x00016F49 0x0000000F
0x00000074 0x00016F58 0x00000010
0x00000075 0x00016F6F 0x00000011
0x00000076 0x00017191 0x00000012
0x00000077 0x00001F42 0x00000013
0x00000078 0x00014C18 0x00000014
0x00000079 0x00004DBA 0x00000015
0x0000007A 0x0000215C 0x00000016
0x0000007B 0x00007E00 0x00000017
0x0000007C 0x00012116 0x00000018
0x0000007D 0x00000857 0x00000019
0x0000007E 0x0001395B 0x0000001A
0x0000007F 0x0000EBF6 0x0000001B
0x00000080 0x00014C10 0x0000001C
```

*Ergebnis 2-68*

In Abfrage 2-68 liefert die Funktion HEX die drei angegebenen Spalten der Tabelle **customer** im hexadezimalen Format zurück.

# Gespeicherte Prozeduren in SELECT-Anweisungen

Sie haben Beispiele von SELECT-Anweisungen gesehen, in denen Spaltennamen, Operatoren und Funktionen verwendet wurden. Andere Ausdrücke beinhalten den Aufruf einer sogenannten *gespeicherten Prozedur* (Stored Procedures).

Gespeicherte Prozeduren bestehen aus SPL-Anweisungen (SPL = Stored Procedure Language) und herkömmlichen SQL-Anweisungen. Ausführliche Informationen zu diesem Thema finden Sie in Kapitel 14 "Gespeicherte Prozeduren".

Gespeicherte Prozeduren erweitern die Auswahl der Ihnen zur Verfügung stehenden Funktionen. Sie erlauben es Ihnen, jeden gefundenen Datensatz in einer Unterabfrage zu verwenden.

Angenommen, Sie wollen eine Liste aller Kunden erzeugen mit der Nummer und dem Nachnamen des Kunden, sowie der Anzahl der Bestellungen, die der jeweilige Kunde bisher gemacht hat. Die unten gezeigte SELECT-Anweisung zeigt eine Möglichkeit, an diese Information zu kommen. Die Tabelle **customer** beinhaltet zwar die Spalten **customer_num** und **lname**, aber keine Information über die Anzahl von Bestellungen, die jeder Kunde aufgegeben hat. Sie könnten allerdings eine Prozedur **get_orders** schreiben, die die Tabelle **orders** nach jeder Kundennummer durchgeht und die Anzahl der jeweiligen Bestellungen ausgibt:

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
        FROM customer
```

*Abfrage 2-69*

Das Ergebnis dieser gespeicherten Prozedurabfrage sehen Sie hier:

```
customer_num lname                n_orders

         101 Pauli                       1
         102 Sadler                      0
         103 Currie                      0
         104 Higgins                     4
         105 Vector                      0
         106 Watson                      2
         107 Ream                        0
         108 Quinn                       0
         109 Miller                      0
         110 Jaeger                      2
         111 Keyes                       1
         112 Lawson                      1
         113 Beatty                      0
         114 Albertson                   0
         115 Grant                       1
         116 Parmelee                    1
         117 Sipes                       2
         118 Baxter                      0
         119 Shorter                     1
         120 Jewell                      1
         121 Wallack                     1
         122 O'Brian                     1
         123 Hanlon                      1
         124 Putnum                      1
         125 Henry                       0
         126 Neelie                      1
         127 Satifer                     1
         128 Lessor                      0
```

*Ergebnis 2-69*

In gespeicherte Prozeduren können Sie Datenbankoperationen speichern, die Sie immer wieder durchführen müssen. So enthält die Bedingung in der folgenden Abfrage die Prozedur **conv_price**, mit der der Stückpreis eines Artikels in eine Fremdwährung umgerechnet wird und anfallende Zuschläge miteingerechnet werden.

```
SELECT stock_num, manu_code, description FROM stock
    WHERE conv_price(unit_price, ex_rate = 1.50, tarif = 50.00)
        < 1000
```

*Abfrage 2-70*

# SELECT-Anweisungen über mehrere Tabellen

Sie können Daten von zwei oder mehr Tabellen auswählen, indem Sie die Tabellen in der FROM-Klausel angeben. Fügen Sie die WHERE-Klausel hinzu, um damit im Rahmen einer *Join*-Bedingung die gleichartigen Spalten der jeweiligen Tabellen miteinander zu verknüpfen. Dies erzeugt eine temporäre, zusammengesetzte Tabelle, in der je zwei Sätze, die die gestellte Bedingung erfüllen, zu einem Satz zusammengesetzt werden.

Ein *einfacher Join* verknüpft Daten aus zwei oder mehr Tabellen, indem er eine Spalte der einen Tabelle in Beziehung setzt zur anderen Spalte der anderen Tabelle. Ein *zusammengesetzter Join* verknüpft zwei oder mehr Tabellen, indem er zwei oder mehr Spalten der einen Tabelle in Beziehung setzt zu zwei oder mehr Spalten der anderen Tabelle.

Um einen Join zu erzeugen, müssen Sie eine *Join-Bedingung* spezifizieren. Dabei muß mindestens eine Spalte pro Tabelle beteiligt sein. Weil die Spalten miteinander verglichen werden, müssen sie kompatible Datentypen haben. Wenn Sie große Tabellen miteinander verknüpfen, sollten die an der Join-Bedingung beteiligten Spalten indiziert sein, um die Performance zu verbessern.

Eine ausführliche Beschreibung von Datentypen finden Sie in Kapitel 3 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen*. Eine ausführliche Beschreibung über das Indizieren finden Sie in Kapitel 10 "Das Modell tunen" und Kapitel 13 "Datenbankserver-Abfragen optimieren".

## Ein kartesisches Produkt erzeugen

Wenn Sie eine Abfrage über mehrere Tabellen durchführen, ohne dabei eine explizite Join-Bedingung zur Verbindung der Tabellen anzugeben, dann erzeugen Sie damit ein *kartesisches Produkt* (Kreuzprodukt). Ein kartesisches Produkt besteht aus einer Kombination aller nur möglichen Datensätze der einzelnen Tabellen. Es liefert normalerweise ein sehr großes, unhandliches und bezüglich der Daten auch ungenaues Ergebnis.

Abfrage 2-71 wählt Daten aus zwei Tabellen aus und produziert ein kartesisches Produkt.

```
SELECT * FROM customer, state
```

*Abfrage 2-71*

Obwohl es nur 52 Sätze in der Tabelle **state** und 28 Sätze in der Tabelle **customer** gibt, liefert diese SELECT-Anweisung als Ergebnis einer Multiplikation der Datensätze der einen Tabelle mit den Datensätzen der anderen Tabelle, die enorme Menge von 1456 Sätzen zurück.

```
customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          AK
sname         Alaska

customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          HI
sname         Hawaii

customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          CA
sname         California
.
.
.
```

*Ergebnis 2-71*

Beachten Sie: Einige Daten bei den verknüpften Sätzen sind nicht korrekt. Wo zum Beispiel **city** und **state** der Tabelle **customer** eine Adresse in Kalifornien anzeigen, können **code** und **sname** der Tabelle **state** die Werte eines anderen Staates anzeigen.

# Erzeugen eines Joins

Um das kartesische Produkt zu verfeinern und einzuschränken und um sinnlose Datensätze auszuschließen, kann man eine SELECT-Anweisung um die WHERE-Klausel mit einer sinnvollen Join-Bedingung ergänzen.

Dieser Abschnitt erläutert *Equi-Joins*, *natürliche Joins* und *Mehr-Tabellen-Joins*. Komplexere Arten wie z. B. *Self-Joins* oder *Outer-Joins* sind in Kapitel 3 "Komplexe SELECT-Anweisungen" beschrieben.

## Equi-Join

Ein Equi-Join (Gleichsetzungs-Join) beruht auf der Gleichheit bzw. Übereinstimmung von Werten. Die Gleichheit wird im Rahmen der WHERE-Klausel durch ein Gleichheitszeichen (=) in der Bedingung angegeben.

```
SELECT * FROM manufact, stock

WHERE manufact.manu_code = stock.manu_code
```

*Abfrage 2-72*

Abfrage 2-72 verknüpft die Tabellen **manufact** und **stock** über die Spalte **manucode**. Der Zugriff erfolgt nur auf diejenigen Sätze, für die gilt, daß die Werte der beiden Spalten gleich sind.

```
manu_code     SMT
manu_name     Smith
lead_time        3
stock_num     1
manu_code     SMT
description   baseball gloves
unit_price    $450.00
unit          case
unit_descr    10 gloves/case

manu_code     SMT
manu_name     Smith
lead_time        3
stock_num     5
manu_code     SMT
description   tennis racquet
unit_price    $25.00
unit          each
unit_descr    each

manu_code     SMT
manu_name     Smith
lead_time        3
stock_num     6
manu_code     SMT
description   tennis ball
unit_price    $36.00
unit          case
unit_descr    24 cans/case

manu_code     ANZ
manu_name     Anza
lead_time        5
stock_num     5
manu_code     ANZ
description   tennis racquet
unit_price    $19.80
unit          each
unit_descr    each
.
.
.
```

*Ergebnis 2-72*

Beachten Sie: Bei diesem Equi-Join wird die Spalte **manu_code** zweimal ausgegeben, nämlich einmal als Teil der Tabelle **manufact** und einmal als Teil der Tabelle **stock**. Dies geschieht, weil bei der Auswahlliste alle Spalten zur Ausgabe angefordert wurden.

Man kann bei einem Equi-Join auch weitere Bedingungen hinzufügen, z. B. solche, die auf der Ungleichheit der Werte der verknüpften Spalten beruhen. Bei diesen Arten von Join wird im Rahmen der WHERE-Klausel ein Vergleichs-Operator angegeben, der nicht das Zeichen = ist.

Wenn Spalten in den verknüpften Tabellen den gleichen Namen haben, dann müssen den Spaltennamen die Namen der Tabellen und je ein Punkt vorangestellt werden, wie das folgende Beispiel zeigt:

```
SELECT order_num, order_date, ship_date, cust_calls.*
    FROM orders, customer_num
    WHERE call_dtime >= ship_date

AND cust_calls.customer_num = orders.customer_num
    ORDER BY customer_num
```

*Abfrage 2-73*

Abfrage 2-73 stellt die Verknüpfung über die Spalten **customer_num** her. Es werden nur diejenigen Sätze ausgewählt, für die gilt, daß der Wert von **call_dtime** der Tabelle **cust_calls** größer oder gleich dem Wert von **ship_date** der Tabelle **orders** ist. Die Anweisung liefert die folgenden Sätze zurück:

```
order_num       1004
order_date      05/22/1993
ship_date       05/30/1993
customer_num    106
call_dtime      1993-06-12 08:20
user_id         maryj
call_code       D
call_descr      Order received okay, but two of the cans of
                ANZ tennis balls within the case were empty
res_dtime       1993-06-12 08:25
res_descr       Authorized credit for two cans to customer,
                issued apology. Called ANZ buyer to report
                the qa problem.

order_num       1008
order_date      06/07/1993
ship_date       07/06/1993
customer_num    110
call_dtime      1993-07-07 10:24
user_id         richc
call_code       L
call_descr      Order placed one month ago (6/7) not received.
res_dtime       1993-07-07 10:30
res_descr       Checked with shipping (Ed Smith). Order out
                yesterday-was waiting for goods from ANZ.
                Next time will call with delay if necessary.

order_num       1023
order_date      07/24/1993
ship_date       07/30/1993
customer_num    127
call_dtime      1993-07-31 14:30
user_id         maryj
call_code       I
call_descr      Received Hero watches (item # 304) instead
                of ANZ watches
res_dtime
res_descr       Sent memo to shipping to send ANZ item 304
                to customer and pickup HRO watches. Should
                be done tomorrow, 8/1
```

*Ergebnis 2-73*

## Natürlicher Join

Ein natürlicher Join ist so aufgebaut, daß er keine redundanten Daten ausgibt. Dies zeigt das folgende Beispiel:

```
SELECT manu_name, lead_time, stock.*
    FROM manufact, stock

WHERE manufact.manu_code = stock.manu_code
```

*Abfrage 2-74*

Wie im Beispiel zum Equi-Join gezeigt, verknüpft auch Abfrage 2-73 die Tabellen **manufact** und **stock** über die Spalten **manu_code**. Aber weil die Auswahlliste präziser definiert wurde, erfolgte der Zugriff auf die Spalte **manu_code** nur einmal pro Satz:

```
manu_name     Smith
lead_time        3
stock_num     1
manu_code     SMT
description   baseball gloves
unit_price    $450.00
unit          case
unit_descr    10 gloves/case

manu_name     Smith
lead_time        3
stock_num     5
manu_code     SMT
description   tennis racquet
unit_price    $25.00
unit          each
unit_descr    each

manu_name     Smith
lead_time        3
stock_num     6
manu_code     SMT
description   tennis ball
unit_price    $36.00
unit          case
unit_descr    24 cans/case

manu_name     Anza
lead_time        5
stock_num     5
manu_code     ANZ
description   tennis racquet
unit_price    $19.80
unit          each
unit_descr    each
.
.
.
```

*Ergebnis 2-74*

Alle Joins sind *assoziativ*. Das bedeutet, daß die Reihenfolge der Join-Festlegungen in der WHERE-Klausel keinen Einfluß auf die Bedeutung des Joins hat.

```
SELECT catalog.*, description, unit_price, unit, unit_descr
    FROM catalog, stock
    WHERE catalog.stock_num = stock.stock_num
        AND catalog.manu_code = stock.manu_code
        AND catalog_num = 10017

SELECT catalog.*, description, unit_price, unit, unit_descr
    FROM catalog, stock
    WHERE catalog_num = 10017
        AND catalog.manu_code = stock.manu_code
        AND catalog.stock_num = stock.stock_num
```

*Abfrage 2-75*

Beide SELECT-Anweisungen in Abfrage 2-75 erzeugen den gleichen natürlichen Join und liefern auch den gleichen Datensatz zurück:

```
catalog_num  10017
stock_num    101
manu_code    PRC
cat_descr
Reinforced, hand-finished tubular. Polyurethane belted.
Effective against punctures. Mixed tread for super wear
and road grip.
cat_picture  <BYTE value>

cat_advert   Ultimate in Puncture Protection, Tires
             Designed for In-City Riding
description  bicycle tires
unit_price   $88.00
unit         box
unit_descr   4/box
```

*Ergebnis 2-75*

Beachten Sie: Hierbei wird auch eine TEXT-Spalte, **cat_descrt**, eine BYTE-Spalte, **cat_picture** und eine VARCHAR-Spalte, **cat_advert**, ausgegeben.

## Mehr-Tabellen-Join

Ein Mehr-Tabellen-Join verbindet mehr als zwei Tabellen über eine oder mehrere zueinander gehörige Spalten. Es kann sich dabei um einen Equi-Join oder auch um einen natürlichen Join handeln.

```
SELECT * FROM catalog, stock, manufact
    WHERE catalog.stock_num = stock.stock_num
        AND stock.manu_code = manufact.manu_code
        AND catalog_num = 10025
```

*Abfrage 2-76*

Abfrage 2-76 erzeugt einen Equi-Join über die Tabellen **catalog**, **stock** und **manufact**. Sie liefert die folgenden Sätze zurück:

```
catalog_num  10025
stock_num    106
manu_code    PRC
cat_descr
Hard anodized alloy with pearl finish; 6mm hex bolt hardware.
Available in lengths of 90-140mm in 10mm increments.
cat_picture  <BYTE value>

cat_advert   ProCycle Stem with Pearl Finish
stock_num    106
manu_code    PRC
description  bicycle stem
unit_price   $23.00
unit         each
unit_descr   each
manu_code    PRC
manu_name    ProCycle
lead_time       9
```

*Ergebnis 2-76*

Beachten Sie: **manu_code** wird dreimal wiederholt, und zwar je einmal pro Tabelle. **stock_num** wird zweimal wiederholt.

Wegen der beträchtlichen Redundanz im vorherigen Beispiel einer Mehr-Tabellen-Abfrage ist es empfehlenswert, die SELECT-Anweisung präziser zu formulieren. Hierfür gibt man die gewünschten Spalten in der Auswahlliste einzeln an.

```
SELECT catalog.*, description, unit_price, unit,
      unit_descr, manu_name, lead_time
    FROM catalog, stock, manufact
    WHERE catalog.stock_num = stock.stock_num
        AND stock.manu_code = manufact.manu_code
        AND catalog_num = 10025
```

*Abfrage 2-77*

Abfrage 2-77 wählt mit einem Jokerzeichen die komplette Tabelle aus, die die meisten Spalten hat. Von den beiden anderen Tabellen benennt sie in der Folge nur einzelne Spalten. Die SELECT-Anweisung produziert den folgen-

den natürlichen Join, der die gleichen Daten ausgibt wie das vorherige Beispiel, diesmal jedoch ohne Redundanzen:

```
catalog_num  10025
stock_num    106
manu_code    PRC
cat_descr
Hard anodized alloy with pearl finish. 6mm hex bolt hardware.
Available in lengths of 90-140mm in 10mm increments.
cat_picture  <BYTE value>

cat_advert   ProCycle Stem with Pearl Finish
description  bicycle stem
unit_price   $23.00
unit         each
unit_descr   each
manu_name    ProCycle
lead_time       9
```

*Ergebnis 2-77*

# Möglichkeiten zur Verkürzung von Abfragen

Sie können Alias-Namen (Alternativ-Namen) vergeben, die Klausel INTO TEMP verwenden und andere Namen für die Anzeige angeben und so den Weg durch Joins und Mehr-Tabellen-Abfragen beschleunigen und Ausgaben zur weiteren Verarbeitung produzieren.

## Verwendung von Alias-Namen

Sie können Abfragen über mehrere Tabellen dadurch kürzer und lesbarer machen, indem Sie für die Tabellen in der SELECT-Anweisung *Alias-Namen* vergeben. Ein Alias ist ein Wort, das in der FROM-Klausel unmittelbar auf den Namen der Tabelle folgt. Man kann ein Alias überall dort verwenden, wo sonst der Name der Tabelle angegeben würde, z. B. als Präfix vor den Spaltennamen in den anderen Klauseln.

```
SELECT s.stock_num, s.manu_code, s.description,
       s.unit_price, s.unit, c.catalog_num,
       c.cat_descr, c.cat_advert, m.lead_time

FROM stock s, catalog c, manufact m
    WHERE s.stock_num = c.stock_num
        AND s.manu_code = c.manu_code
        AND s.manu_code = m.manu_code
        AND s.manu_code IN ('HRO', 'HSK')
        AND s.stock_num BETWEEN 100 AND 301
    ORDER BY catalog_num
```

*Abfrage 2-78a*

Die assoziative Eigenart der SELECT-Anweisung erlaubt es, ein Alias zu verwenden, bevor er definiert wurde. In diesem Beispiel wurden die Alias-Namen **s** für die Tabelle **stock**, **c** für die Tabelle **customer** und **m** für die Tabelle **manufact** im Rahmen der FROM-Klausel vergeben und dann in der SELECT-Klausel sowie der WHERE-Klausel als Präfix vor den Spaltennamen verwendet.

Vergleichen Sie die Länge der vorausgegangenen SELECT-Anweisung mit der Länge der folgenden SELECT-Anweisung, bei der keine Alias-Namen verwendet wurden.

```
SELECT stock.stock_num, stock.manu_code, stock.description,
       stock.unit_price, stock.unit, catalog.catalog_num,
       catalog.cat_descr, catalog.cat_advert,
       manufact.lead_time
    FROM stock, catalog, manufact
    WHERE stock.stock_num = catalog.stock_num
        AND stock.manu_code = catalog.manu_code
        AND stock. = manufact.manu_code
        AND stock.manu_code IN ('HRO', 'HSK')
        AND stock.stock_num BETWEEN 100 AND 301
    ORDER BY catalog_num
```

*Abfrage 2-78b*

Beide SELECT-Anweisung sind gleichbedeutend und liefern die folgenden Sätze:

```
stock_num    110
manu_code    HRO
description  helmet
unit_price   $260.00
unit         case
catalog_num  10033
cat_descr
Newest ultralight helmet uses plastic shell. Largest ventilation
channels of any helmet on the market. 8.5 oz.
cat_advert   Lightweight Plastic Slatted with Vents Assures Cool
             Comfort Without Sacrificing Protection
lead_time       4

stock_num    110
manu_code    HSK
description  helmet
unit_price   $308.00
unit         each
catalog_num  10034
cat_descr
Aerodynamic (teardrop) helmet covered with anti-drag fabric.
Credited with shaving 2 seconds/mile from winner's time in
Tour de France time-trial. 7.5 oz.
cat_advert   Teardrop Design Endorsed by Yellow Jerseys,
             You Can Time the Difference
lead_time       5

stock_num    205
manu_code    HRO
description  3 golf balls
unit_price   $312.00
unit         each
catalog_num  10048
cat_descr
Combination fluorescent yellow and standard white.
cat_advert   HiFlier Golf Balls: Case Includes Fluorescent
             Yellow and Standard White
lead_time       4

stock_num    301
manu_code    HRO
description  running shoes
unit_price   $42.50
unit         each
catalog_num  10050
cat_descr
Engineered for serious training with exceptional stability.
Fabulous shock absorption. Great durability. Specify
mens/womens, size.
cat_advert   Pronators and Supinators Take Heart: A Serious
             Training Shoe For Runners Who Need Motion Control
lead_time       4
```

*Ergebnis 2-78*

Beachten Sie: Nach der TEXT-Spalte **cat_descr** oder der BYTE-Spalte **cat_picture** kann man nicht sortieren (ORDER BY).

Man kann Alias-Namen auch dazu verwenden, Abfragen über externe Tabellen von externen Datenbanken abzukürzen.

```
SELECT order_num, lname, fname, phone

FROM masterdb@central:customer c, sales@western:orders o
    WHERE c.customer_num = o.customer_num
        AND order_num <= 1010
```

*Abfrage 2-79*

Abfrage 2-79 verknüpft Spalten aus zwei Tabellen von unterschiedlichen Datenbanken und Datenbankservern. Die Datenbank ist nicht die aktuelle Datenbank und der Datenbankserver ist auch nicht der aktuelle Server. Den Langnamen *datenbank@dbservername:tabelle* konkret: **masterdb@central:customer** beziehungsweise **sales@western:orders** werden die Alias-Namen **c** und **o** zugewiesen. Dies macht es möglich, daß man die Ausdrücke in der WHERE-Klausel mit den Alias-Namen abkürzen kann. Die Anweisung liefert die folgenden Daten:

```
order_num lname           fname          phone

     1001 Higgins         Anthony        415-368-1100
     1002 Pauli           Ludwig         408-789-8075
     1003 Higgins         Anthony        415-368-1100
     1004 Watson          George         415-389-8789
     1005 Parmelee        Jean           415-534-8822
     1006 Lawson          Margaret       415-887-7235
     1007 Sipes           Arnold         415-245-4578
     1008 Jaeger          Roy            415-743-3611
     1009 Keyes           Frances        408-277-7245
     1010 Grant           Alfred         415-356-1123
```

*Ergebnis 2-79*

Nähere Informationen über externe Tabellen und Datenbanken finden Sie in Kapitel 12 "Datenbankserver und Netzwerke", sowie in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

Auch Synonyme können als Kurzverweise auf längere Namen von externen oder aktuellen Tabellen oder Views verwendet werden. Nähere Informationen über die Verwendung von Synonymen finden Sie in Kapitel 12 "Informix und Netzwerke" sowie in der Beschreibung der CREATE SYNONYM-Anweisung in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

## Die INTO TEMP-Klausel

Mit der INTO TEMP-Klausel der SELECT-Anweisung können Sie das Ergebnis einer Datenbankabfrage über mehrere Tabellen temporär in einer separaten Tabelle zwischenspeichern. Diese temporäre Tabelle kann abgefragt oder manipuliert werden, ohne daß dabei die Datenbank verändert wird. Temporäre Tabellen werden automatisch gelöscht, sobald eine SQL-Sitzung oder das Listen-Programm beendet ist.

```
SELECT DISTINCT stock_num, manu_name, description,
            unit_price, unit_price * 1.05 adj_price
    FROM stock, manufact

WHERE manufact.manu_code = stock.manu_code
    INTO TEMP stockman
```

*Abfrage 2-80*

Abfrage 2-80 erzeugt eine temporäre Tabelle mit dem Namen **stockman** und speichert darin das Ergebnis der Abfrage. Da alle Spalten in temporären Tabellen Namen haben müssen, wurde der Alias-Name **adj_price** verwendet.

| stock_num | manu_name | description | unit_price | adj_price |
|---|---|---|---|---|
| 1 | Hero | baseball gloves | $250.00 | $262.5000 |
| 1 | Husky | baseball gloves | $800.00 | $840.0000 |
| 1 | Smith | baseball gloves | $450.00 | $472.5000 |
| 2 | Hero | baseball | $126.00 | $132.3000 |
| 3 | Husky | baseball bat | $240.00 | $252.0000 |
| 4 | Hero | football | $480.00 | $504.0000 |
| 4 | Husky | football | $960.00 | $1008.0000 |
| . | | | | |
| . | | | | |
| . | | | | |
| 306 | Shimara | tandem adapter | $190.00 | $199.5000 |
| 307 | ProCycle | infant jogger | $250.00 | $262.5000 |
| 308 | ProCycle | twin jogger | $280.00 | $294.0000 |
| 309 | Hero | ear drops | $40.00 | $42.0000 |
| 309 | Shimara | ear drops | $40.00 | $42.0000 |
| 310 | Anza | kick board | $84.00 | $88.2000 |
| 310 | Shimara | kick board | $80.00 | $84.0000 |
| 311 | Shimara | water gloves | $48.00 | $50.4000 |
| 312 | Hero | racer goggles | $72.00 | $75.6000 |
| 312 | Shimara | racer goggles | $96.00 | $100.8000 |
| 313 | Anza | swim cap | $60.00 | $63.0000 |
| 313 | Shimara | swim cap | $72.00 | $75.6000 |

*Ergebnis 2-80*

Man kann diese Tabelle abfragen und auch mit anderen Tabellen verknüpfen. Das vermeidet mehrfache Sortierung und erlaubt schnelleres Bewegen durch die Datenbank. Kapitel 13 "Datenbankserver-Abfragen optimieren" behandelt temporäre Tabellen ausführlich.

# Zusammenfassung

Das vorliegende Kapitel führte beispielhaft in die Syntax grundlegender SELECT-Anweisungen ein, mit denen man eine relationale Datenbank abfragen kann und zeigt auch die Ergebnisse der Abfragen. Folgende Aktionen werden erläutert:

- Alle Spalten und Sätze einer Tabelle mit den Klauseln SELECT und FROM auswählen,
- bestimmte Spalten einer Tabelle mit den Klauseln SELECT und FROM auswählen,
- bestimmte Sätze einer Tabelle mit den Klauseln SELECT, FROM und WHERE auswählen,
- die Schlüsselwörter DISTINCT oder UNIQUE verwenden, um damit doppelt vorkommende Sätze auszuschließen,
- die ermittelten Daten mit der ORDER BY-Klausel und dem Schlüsselwort DESC sortieren,
- Daten auswählen und sortieren, die landesspezifische Sonderzeichen enthalten,
- die Schlüsselwörter BETWEEN, IN, MATCHES und LIKE und verschiedene relationale Operatoren in der WHERE-Klausel verwenden, mit denen man eine vergleichende Bedingung erzeugen kann,
- vergleichende Bedingungen erzeugen, durch die Werte ein- oder ausgeschlossen werden (über Schlüsselwörter, relationale Operatoren und Feldausschnitte), und durch die Untermengen ermittelt werden,
- beliebigen Text über Textvergleich suchen, unter Verwendung von einschränkenden bzw. nicht einschränkenden Jokerzeichen,
- die logischen Operatoren AND, OR und NOT verwenden, mit denen man Suchbedingungen oder Boolsche Ausdrücke in der WHERE-Klausel erweitern kann,
- das Schlüsselwort ESCAPE verwenden, um bestimmte Zeichen in einer Abfrage zu entwerten,
- die Schlüsselwörter IS NULL bzw. IS NOT NULL in der WHERE-Klausel verwenden, um nach NULL-Werten zu suchen,
- die arithmetischen Operatoren in der SELECT-Klausel verwenden, um damit Berechnungen über numerische Spalten durchzuführen und um die ermittelten Ergebnisse anzuzeigen,
- Zeichenketten- und Feldausschnitte verwenden, um damit Abfragen zu gestalten,

- Spaltenüberschriften für Spalten zuweisen, die durch Berechnung zustande gekommen sind (als Formatiermöglichkeit bei Listen),
- die Mengenfunktionen COUNT, AVG, MAX, MIN und SUM im Rahmen der SELECT-Klausel verwenden, um damit bestimmte Daten zu ermitteln und ausgeben zu lassen,
- die Zeitfunktionen DATE, DAY, MDY, MONTH, WEEKDAY, YEAR, CURRENT und EXTEND sowie TODAY, LENGTH und USER in einer SELECT-Anweisung verwenden,
- Gespeicherte Prozeduren in Ihren SELECT-Anweisungen verwenden.

Weiter hat Sie dieses Kapitel in einfache Join-Bedingungen eingeführt. Mit Join-Bedingungen können Sie Daten von zwei oder mehr Tabellen ermitteln und anzeigen lassen. Der Abschnitt "SELECT-Anweisungen über mehrere Tabellen" hat Sie darüber informiert, wie man

- ein kartesisches Produkt erzeugt,
- ein kartesisches Produkt dadurch einschränkt, indem man bei der Abfrage im Rahmen der WHERE-Klausel eine gültige Join-Bedingung hinzufügt,
- einen natürlichen Join und einen Equi-Join erzeugt,
- zwei oder mehr Tabellen über eine oder mehrere Spalten verknüpft,
- Alias-Namen als Möglichkeit zur Verkürzung von Abfragen über mehrere Tabellen verwendet,
- durch Verwendung der INTO TEMP-Klausel die ausgewählten Daten in eine separate, temporäre Tabelle weiterleitet, um so Berechnungen außerhalb der Datenbank durchführen zu können.

Das nächste Kapitel erläutert komplexere Abfragen und Unterabfragen, Self-Joins und Outer-Joins, die Klauseln GROUP BY und HAVING und die Tabellen-Operationen UNION, INTERSECTION und DIFFERENCE.

# Komplexe SELECT-Anweisungen

3

# Kapitelüberblick

In Kapitel 2 "Einfache SELECT-Anweisungen" wurden einige grundlegende Wege aufgezeigt, wie man mit der SELECT-Anweisung Daten aus einer Datenbank abrufen kann. Dieses Kapitel vermittelt weitere Kenntnisse über die SQL-Anweisung. Es zeigt, wie man komplexere Datenbankabfragen und Datenbearbeitungen gestalten kann.

Während sich das vorangegangene Kapitel auf fünf Klauseln aus der Syntax einer SELECT-Anweisung konzentriert hat, werden in diesem Kapitel zwei weitere Klauseln behandelt. Sie können die GROUP BY-Klausel zusammen mit Mengenfunktionen verwenden, um die Datensätze zu ordnen, die von der FROM-Klausel zurückgeliefert werden. Sie können die HAVING-Klausel angeben, um Bedingungen an die Werte zu stellen, die von der Klausel GROUP BY zurückgeliefert werden.

Das vorliegende Kapitel erweitert die frühere Erläuterung über Joins. Sie erfahren, wie Sie mit *Self-Joins* eine Tabelle mit sich selbst verknüpfen können. Das Kapitel beschreibt außerdem vier Arten eines *Outer Joins*, bei dem Sie das Schlüsselwort OUTER verwenden, um zwei oder mehr verknüpfte Tabellen unterschiedlich zu behandeln. Sie werden daneben in korrelierte und nichtkorrelierte Unterabfragen und deren operationale Schlüsselwörter eingeführt. Es wird Ihnen außerdem gezeigt, wie man Abfragen mit Hilfe des Operators UNION verbindet. Daneben werden eine Reihe von Operationen vorgestellt, die als Vereinigungsmenge, Schnittmenge und Unterschiedsmenge bekannt sind.

Die Beispiele in diesem Kapitel zeigen, wie Sie einige oder auch alle Klauseln der SELECT-Anweisung in Ihren Abfragen verwenden können. Die Klauseln müssen in folgender Reihenfolge angegeben werden:

1. SELECT-Klausel
2. FROM-Klausel
3. WHERE-Klausel
4. GROUP BY-Klausel
5. HAVING-Klausel
6. ORDER BY-Klausel
7. INTO TEMP-Klausel

Mit einer weiteren Klausel der SELECT-Anweisung, der INTO-Klausel, können Sie Programm- und Hostvariablen bei **INFORMIX-4GL** und bei **SQL-APIs** angeben. Diese Klausel wird in Kapitel 5 "SQL in Programmen" beschrieben sowie in der Dokumentation zum jeweiligen Produkt.

# Die Klauseln GROUP BY und HAVING

Die optionalen Klauseln GROUP BY und HAVING erweitern den Funktionsumfang einer SELECT-Anweisung. Sie können eine oder beide Klauseln in einer SELECT-Anweisung verwenden.

Die GROUP BY-Klausel verbindet einander ähnliche Datensätze dadurch, daß für jede *Gruppe* von Datensätzen ein Ergebnissatz gebildet wird. Die Datensätze einer Gruppe haben dieselben Werte in jeder Spalte, die in der Auswahlliste angegeben ist. Die HAVING-Klausel stellt Bedingungen an diese Gruppen, nachdem sie gebildet wurden. Die GROUP BY-Klausel kann ohne die HAVING-Klausel verwendet werden. Ebenso kann die HAVING-Klausel ohne die GROUP BY-Klausel verwendet werden.

## Die GROUP BY-Klausel

Die GROUP BY-Klausel teilt eine Tabelle in Gruppen ein. Diese Klausel wird sehr oft mit den Mengenfunktionen kombiniert. Die Mengenfunktionen liefern für jede dieser Gruppen gemeinsame Werte. Kapitel 2 "Einfache SELECT-Anweisungen" zeigt einige Beispiele für die Verwendung der Mengenfunktionen, die auf eine komplette Tabelle angewandt werden. Dieses Kapitel dagegen zeigt, wie Mengenfunktionen auf Gruppen von Datensätzen angewendet werden.

Verwendet man die GROUP BY-Klausel ohne eine Mengenfunktion, ist dies gleichbedeutend mit der Verwendung des Schlüsselworts DISTINCT (oder UNIQUE) in der SELECT-Klausel. Kapitel 2 "Einfache SELECT-Anweisungen" enthielt folgendes Beispiel:

```
SELECT DISTINCT customer_num FROM orders
```

*Abfrage 3-1a*

Diese Anweisung könnten Sie auch folgendermaßen angeben:

```
SELECT customer_num
    FROM orders
    GROUP BY customer_num
```

*Abfrage 3-1b*

Beide Anweisungen liefern folgende Datensätze zurück:

```
customer_num

         101
         104
         106
         110
         111
         112
         115
         116
         117
         119
         120
         121
         122
         123
         124
         126
         127
```

*Ergebnis 3-1*

Die GROUP BY-Klausel sammelt die Datensätze in Gruppen, so daß jeder Satz einer jeden Gruppe dieselbe Kundennummer hat. Da keine anderen Spalten ausgewählt wurden, ist das Ergebnis eine Liste der eindeutigen Kundennummern, die in der Spalte **customer_num** stehen.

Die Mächtigkeit der GROUP BY-Klausel wird noch deutlicher, wenn diese zusammen mit Mengenfunktionen verwendet wird.

```
SELECT order_num, COUNT (*) number, SUM (total_price) price
    FROM items
    GROUP BY order_num
```

*Abfrage 3-2*

In Abfrage 3-2 wird die Anzahl der jeweiligen Artikel und der Gesamtpreis der Artikel pro Auftrag ermittelt. Die GROUP BY-Klausel bewirkt, daß die Datensätze der Tabelle **items** in Gruppen zusammengefaßt werden. Jede Datensatzgruppe besteht aus Datensätzen, die in der Spalte **order_num** identische Werte haben, d. h. die Artikel eines jeden Auftrags sind zu einer Gruppe zusammengefaßt. Nachdem die Gruppen gebildet wurden, werden die Mengenfunktionen COUNT und SUM auf jede dieser Gruppen angewendet.

Diese Abfrage liefert für jede Gruppe einen Datensatz zurück. Außerdem vergibt diese Abfrage die folgenden Überschriften für die Spalten, die das Ergebnis der Ausdrücke COUNT und SUM enthalten:

| order_num | number | price |
|---|---|---|
| 1001 | 1 | $250.00 |
| 1002 | 2 | $1200.00 |
| 1003 | 3 | $959.00 |
| 1004 | 4 | $1416.00 |
| 1005 | 4 | $562.00 |
| 1006 | 5 | $448.00 |
| 1007 | 5 | $1696.00 |
| 1008 | 2 | $940.00 |
| . | | |
| . | | |
| . | | |
| 1015 | 1 | $450.00 |
| 1016 | 4 | $654.00 |
| 1017 | 3 | $584.00 |
| 1018 | 5 | $1131.00 |
| 1019 | 1 | $1499.97 |
| 1020 | 2 | $438.00 |
| 1021 | 4 | $1614.00 |
| 1022 | 3 | $232.00 |
| 1023 | 6 | $824.00 |

*Ergebnis 3-2*

Die SELECT-Anweisung gruppiert diejenigen Datensätze der Tabelle **items**, die über die gleiche Auftragsnummer verfügen. Für jede Gruppe wird die Anzahl der Datensätze (COUNT) und der Gesamtpreis ermittelt.

Beachten Sie: In der GROUP BY-Klausel kann keine TEXT- oder BYTE- Spalte angegeben werden. Um Daten *gruppieren* zu können, müssen Daten sortiert werden können. Bei TEXT- oder BYTE-Daten gibt es jedoch keine natürliche Sortierreihenfolge.

Im Gegensatz zur ORDER BY-Klausel sortiert die GROUP BY-Klausel keine Daten. Wenn Sie Daten in einer bestimmten Reihenfolge sortieren wollen, dann geben Sie *nach* der GROUP BY-Klausel die ORDER BY-Klausel an. Dasselbe tun Sie auch, wenn Sie nach den Argumenten einer Mengenfunktion sortieren wollen.

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
    FROM items
    GROUP BY order_num
    ORDER BY price
```

*Abfrage 3-3*

Abfrage 3-3 unterscheidet sich von der vorherigen lediglich durch die zusätzliche ORDER BY-Klausel, mit der die abgerufenen Datensätze in aufsteigender Reihenfolge des Preises sortiert werden:

```
order_num      number          price

     1010           2         $84.00
     1011           1         $99.00
     1013           4        $143.80
     1022           3        $232.00
     1001           1        $250.00
     1020           2        $438.00
     1006           5        $448.00
     1015           1        $450.00
     1009           1        $450.00
      .
      .
      .
     1018           5       $1131.00
     1002           2       $1200.00
     1004           4       $1416.00
     1014           2       $1440.00
     1019           1       $1499.97
     1021           4       $1614.00
     1007           5       $1696.00
```

*Ergebnis 3-3*

Wie bereits im vorherigen Kapitel gesagt wurde, können Sie in der ORDER BY-Klausel eine Zahl verwenden, mit der die Position der Spalte in der Auswahlliste angezeigt wird. Sie können auch in der GROUP BY-Klausel eine Zahl verwenden, um in der Gruppenliste die Position der Spaltennamen oder Spaltenüberschriften anzugeben.

Die folgende SELECT-Anweisung liefert dieselben Datensätze wie die vorherige Abfrage zurück:

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
    FROM items
    GROUP BY 1
    ORDER BY 3
```

*Abfrage 3-4*

Beachten Sie bei der Erstellung einer Abfrage, daß alle Spalten, die in der Auswahlliste der SELECT-Klausel angegeben, aber kein Argument einer Mengenfunktion sind, auch in der Gruppenliste der GROUP BY-Klausel angegeben werden müssen. Der Grund dafür ist, daß bei einer SELECT-Anwei-

sung mit der GROUP BY-Klausel pro Gruppe nur ein Satz zurückgeliefert werden kann. Die auf GROUP BY folgenden Spalten bestimmen pro Gruppe nur einen eindeutigen Wert; diesen Wert kann man ausgeben lassen. Eine Spalte, die nicht in der Gruppenliste enthalten ist, kann jedoch unterschiedliche Werte in den Datensätzen haben, die zu einer Gruppe gehören.

Sie können die GROUP BY-Klausel auch in einer SELECT-Anweisung verwenden, die Tabellen miteinander verknüpft.

```
SELECT o.order_num, SUM (i.total_price)
    FROM orders o, items i
    WHERE o.order_date > '01/01/92'
        AND o.customer_num = 110
        AND o.order_num = i.order_num
    GROUP BY o.order_num
```

*Abfrage 3-5*

Die Abfrage 3-5 verknüpft die Tabellen **orders** und **items**, weist den Tabellen Alias-Namen zu und liefert die folgenden zwei Datensätze zurück:

```
order_num            (sum)

     1008          $940.00
     1015          $450.00
```

*Ergebnis 3-5*

# Die HAVING-Klausel

Die HAVING-Klausel ergänzt normalerweise die GROUP BY-Klausel dadurch, daß eine oder mehrere Bedingungen an die Gruppen gestellt werden, nachdem die Gruppen gebildet wurden. Diese Klausel ist vergleichbar mit der WHERE-Klausel, mit der Sie einzelne Datensätze bestimmen. Bei der Verwendung der HAVING-Klausel liegt ein Vorteil darin, daß man die Mengenfunktionen in eine Suchbedingung einschließen kann. Bei der WHERE-Klausel dagegen können keine Mengenfunktionen in der Suchbedingung eingeschlossen werden.

Jede HAVING-Bedingung vergleicht eine Spalte oder einen Mengenausdruck einer Gruppe mit einem anderen Mengenausdruck oder mit einer Konstanten. Man kann eine HAVING-Klausel auch dazu verwenden, Bedingungen sowohl an Spaltenwerte als auch an Werte einer Mengenfunktion zu stellen, die in der Gruppenliste enthalten sind.

```
SELECT order_num, COUNT(*) number, AVG (total_price) average
    FROM items
    GROUP BY order_num
    HAVING COUNT(*) > 2
```

*Abfrage 3-6*

Abfrage 3-6 liefert den durchschnittlichen Gesamtpreis der Artikel für alle Aufträge. Dabei werden aber nur die Aufträge berücksichtigt, die mehr als zwei Posten aufweisen. Die HAVING-Klausel überprüft jede Gruppe sobald sie gebildet ist und wählt diejenigen Gruppen aus, die mehr als zwei Sätze umfassen.

| order_num | number | average |
|---|---|---|
| 1003 | 3 | $319.67 |
| 1004 | 4 | $354.00 |
| 1005 | 4 | $140.50 |
| 1006 | 5 | $89.60 |
| 1007 | 5 | $339.20 |
| 1013 | 4 | $35.95 |
| 1016 | 4 | $163.50 |
| 1017 | 3 | $194.67 |
| 1018 | 5 | $226.20 |
| 1021 | 4 | $403.50 |
| 1022 | 3 | $77.33 |
| 1023 | 6 | $137.33 |

*Ergebnis 3-6*

Wenn Sie die HAVING-Klausel ohne die GROUP BY-Klausel verwenden, dann wird die HAVING-Klausel auf alle Datensätze angewandt, die der Suchbedingung entsprechen. Anders gesagt, alle Sätze, die der Suchbedingung entsprechen, bilden eine einzige Gruppe.

```
SELECT AVG (total_price) average
    FROM items
    HAVING count(*) > 2
```

*Abfrage 3-7*

Abfrage 3-7, die eine veränderte Version des vorherigen Beispiels ist, liefert genau einen Datensatz zurück. Dieser Datensatz enthält den Durchschnitt aller Werte aus der Spalte **total_price**:

```
average

$270.97
```

*Ergebnis 3-7*

Wenn die zuletzt gezeigte Abfrage in der Auswahlliste die Spalte **order_num**, die kein Argument einer Mengenfunktion ist, enthalten hätte, dann hätten Sie eine GROUP BY-Klausel angeben und diese Spalte in die Gruppenliste aufnehmen müssen. Außerdem wäre, wenn die Bedingung in der
HAVING-Klausel nicht erfüllt worden wäre, in der Ausgabe nur die Spaltenüberschrift gezeigt worden. Eine Meldung hätte angezeigt, daß keine Datensätze gefunden wurden.

Im folgenden Beispiel sind alle sieben Klauseln der SELECT-Anweisung enthalten, die Sie bei der Informix-Version des interaktiven SQL verwenden können (die INTO-Klausel, die Programm- oder Hostvariablen benennt, ist nur in einem **INFORMIX-4GL**-Programm oder einem Programm mit eingebetteter Sprache verfügbar):

```
SELECT o.order_num, SUM (i.total_price) price,
      paid_date - order_date span
    FROM orders o, items i
    WHERE o.order_date > '01/01/92'
        AND o.customer_num > 110
        AND o.order_num = i.order_num
    GROUP BY 1, 3
    HAVING COUNT (*) < 5
    ORDER BY 3
    INTO TEMP temptab1
```

*Abfrage 3-8*

Abfrage 3-8 verknüpft die Tabellen **orders** und **items**; sie vergibt Spaltenüberschriften, Alias-Namen für Tabellen und sie verwendet Zahlen als Verweise auf die Spalten; sie gruppiert und sortiert die Daten; und sie schreibt das folgende Ergebnis in eine temporäre Tabelle:

| order_num | price | span |
|---|---|---|
| 1017 | $584.00 | |
| 1016 | $654.00 | |
| 1012 | $1040.00 | |
| 1019 | $1499.97 | 26 |
| 1005 | $562.00 | 28 |
| 1021 | $1614.00 | 30 |
| 1022 | $232.00 | 40 |
| 1010 | $84.00 | 66 |
| 1009 | $450.00 | 68 |
| 1020 | $438.00 | 71 |

*Ergebnis 3-8*

# Fortgeschrittene Joins erstellen

Im vorherigen Kapitel wurde gezeigt, wie in einer SELECT-Anweisung eine WHERE-Klausel eingegeben wird, um zwei oder mehr Tabellen über eine oder mehrere Spalten zu verknüpfen. Es zeigte natürliche Joins und Equi-Joins.

Dieses Kapitel erläutert die Verwendung von zwei komplexeren Arten eines Joins: Self-Joins und Outer Joins. Um Abfragen über mehrere Tabellen zu verkürzen, können Sie, wie bereits im Kapitel 2 "Einfache SELECT-Anweisungen" Joins beschrieben, für Tabellen Alias-Namen vergeben und Ausdrücken Spaltenüberschriften zuweisen. Außerdem können Sie mit der ORDER BY-Klausel Daten sortieren und das Ergebnis einer Abfrage in eine temporäre Tabelle schreiben.

## Self-Joins

Bei einem Join müssen nicht immer zwei verschiedene Tabellen beteiligt sein. Sie können eine Tabelle auch mit sich selbst verknüpfen, und somit einen *Self-Join* erstellen. Dies kann dann sinnvoll sein, wenn Sie Werte einer Spalte mit anderen Werten derselben Spalte vergleichen wollen.

Um einen Self-Join zu erzeugen, geben Sie den Tabellennamen in der FROM-Klausel zweimal an, weisen diesem jedoch in beiden Fällen einen anderen Alias-Namen zu. Um sich auf die Tabelle zu beziehen, verwenden Sie in der SELECT- und WHERE-Klausel diese Alias-Namen so, als ob es zwei verschie-

dene Tabellen wären. (Alias-Namen in einer SELECT-Anweisung werden in Kapitel 2 "Einfache SELECT-Anweisungen" gezeigt und im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* erläutert.)

In Self-Joins können Sie - ebenso wie in Joins zwischen verschiedenen Tabellen - arithmetische Ausdrücke verwenden. Sie können auf NULL-Werte hin abprüfen und nach einer bestimmten Spalte in auf- oder absteigender Reihenfolge sortieren (ORDER BY).

```
SELECT x.order_num, x.ship_weight, x.ship_date,
       y.order_num, y.ship_weight, y.ship_date
    FROM orders x, orders y
    WHERE x.ship_weight >= 5 * y.ship_weight
        AND x.ship_date IS NOT NULL
        AND y.ship_date IS NOT NULL
    ORDER BY x.ship_date
```

*Abfrage 3-9*

Diese SELECT-Anweisung sucht Auftragspaare, bei denen das Verladegewicht (**ship_weight**) um den Faktor fünf oder einen größeren Faktor voneinander abweicht und bei denen das Verladedatum (**ship_date**) nicht gleich NULL ist. Die Daten werden nach den Werten der Spalte **ship_date** sortiert:

| order_num | ship_weight | ship_date | order_num | ship_weight | ship_date |
|---|---|---|---|---|---|
| 1004 | 95.80 | 05/30/1993 | 1011 | 10.40 | 07/03/1993 |
| 1004 | 95.80 | 05/30/1993 | 1020 | 14.00 | 07/16/1993 |
| 1004 | 95.80 | 05/30/1993 | 1022 | 15.00 | 07/30/1993 |
| 1007 | 125.90 | 06/05/1993 | 1015 | 20.60 | 07/16/1993 |
| 1007 | 125.90 | 06/05/1993 | 1020 | 14.00 | 07/16/1993 |
| 1007 | 125.90 | 06/05/1993 | 1022 | 15.00 | 07/30/1993 |
| 1007 | 125.90 | 06/05/1993 | 1011 | 10.40 | 07/03/1993 |
| 1007 | 125.90 | 06/05/1993 | 1001 | 20.40 | 06/01/1993 |
| 1007 | 125.90 | 06/05/1993 | 1009 | 20.40 | 06/21/1993 |
| 1005 | 80.80 | 06/09/1993 | 1011 | 10.40 | 07/03/1993 |
| 1005 | 80.80 | 06/09/1993 | 1020 | 14.00 | 07/16/1993 |
| 1005 | 80.80 | 06/09/1993 | 1022 | 15.00 | 07/30/1993 |
| 1012 | 70.80 | 06/29/1993 | 1011 | 10.40 | 07/03/1993 |
| 1012 | 70.80 | 06/29/1993 | 1020 | 14.00 | 07/16/1993 |
| 1013 | 60.80 | 07/10/1993 | 1011 | 10.40 | 07/03/1993 |
| 1017 | 60.00 | 07/13/1993 | 1011 | 10.40 | 07/03/1993 |
| 1018 | 70.50 | 07/13/1993 | 1011 | 10.40 | 07/03/1993 |
| . | | | | | |
| . | | | | | |
| . | | | | | |

*Ergebnis 3-9*

Angenommen, Sie wollen das Ergebnis eines Self-Joins in eine temporäre Tabelle schreiben, dann würden Sie der SELECT-Anweisung selbstverständlich die INTO TEMP-Klausel hinzufügen. Jedoch müssen Sie zumindest einen Teil der Spaltennamen umbenennen, indem Sie diesen eine Spaltenüberschrift zuweisen. Dies ist erforderlich, da Sie in Wirklichkeit eine neue Tabelle anlegen. Anderenfalls erhalten Sie eine Fehlermeldung, die die doppelt vorkommenden Spaltennamen anzeigt und die temporäre Tabelle wird nicht erstellt.

```
SELECT x.order_num orders1, x.po_num purch1,
       x.ship_date ship1, y.order_num orders2,
       y.po_num purch2, y.ship_date ship2
    FROM orders x, orders y
    WHERE x.ship_weight >= 5 * y.ship_weight
        AND x.ship_date IS NOT NULL
        AND y.ship_date IS NOT NULL
    ORDER BY orders1,orders2
    INTO TEMP shipping
```

*Abfrage 3-10*

Diese SELECT-Anweisung (Abfrage 3-10) vergibt für alle Spalten, die aus der Tabelle **orders** ausgewählt wurden, Überschriften und schreibt diese in eine temporäre Tabelle mit dem Namen **shipping**. Wenn Sie mit `SELECT *` alle Spalten abrufen, dann erhalten Sie diese Datensätze:

| orders1 | purch1 | ship1 | orders2 | purch2 | ship2 |
|---|---|---|---|---|---|
| 1004 | 8006 | 05/30/1993 | 1011 | B77897 | 07/03/1993 |
| 1004 | 8006 | 05/30/1993 | 1020 | W2286 | 07/16/1993 |
| 1004 | 8006 | 05/30/1993 | 1022 | W9925 | 07/30/1993 |
| 1005 | 2865 | 06/09/1993 | 1011 | B77897 | 07/03/1993 |
| 1005 | 2865 | 06/09/1993 | 1020 | W2286 | 07/16/1993 |
| 1005 | 2865 | 06/09/1993 | 1022 | W9925 | 07/30/1993 |
| 1007 | 278693 | 06/05/1993 | 1001 | B77836 | 06/01/1993 |
| 1007 | 278693 | 06/05/1993 | 1009 | 4745 | 06/21/1993 |
| 1007 | 278693 | 06/05/1993 | 1011 | B77897 | 07/03/1993 |
| 1007 | 278693 | 06/05/1993 | 1015 | MA003 | 07/16/1993 |
| 1007 | 278693 | 06/05/1993 | 1020 | W2286 | 07/16/1993 |
| 1007 | 278693 | 06/05/1993 | 1022 | W9925 | 07/30/1993 |
| 1012 | 278701 | 06/29/1993 | 1011 | B77897 | 07/03/1993 |
| 1012 | 278701 | 06/29/1993 | 1020 | W2286 | 07/16/1993 |
| 1013 | B77930 | 07/10/1993 | 1011 | B77897 | 07/03/1993 |
| 1017 | DM354331 | 07/13/1993 | 1011 | B77897 | 07/03/1993 |
| 1018 | S22942 | 07/13/1993 | 1011 | B77897 | 07/03/1993 |
| 1018 | S22942 | 07/13/1993 | 1020 | W2286 | 07/16/1993 |
| 1019 | Z55709 | 07/16/1993 | 1011 | B77897 | 07/03/1993 |
| 1019 | Z55709 | 07/16/1993 | 1020 | W2286 | 07/16/1993 |
| 1019 | Z55709 | 07/16/1993 | 1022 | W9925 | 07/30/1993 |
| 1023 | KF2961 | 07/30/1993 | 1011 | B77897 | 07/03/1993 |

*Ergebnis 3-10*

Sie können eine Tabelle mit sich selbst mehr als einmal verknüpfen. Die maximale Anzahl an Self-Joins hängt von den Betriebsmitteln ab, die Ihnen zur Verfügung stehen.

```
SELECT s1.manu_code, s2.manu_code, s3.manu_code,
       s1.stock_num, s1.description
    FROM stock s1, stock s2, stock s3
    WHERE s1.stock_num = s2.stock_num
        AND s2.stock_num = s3.stock_num
        AND s1.manu_code < s2.manu_code
        AND s2.manu_code < s3.manu_code
    ORDER BY stock_num
```

*Abfrage 3-11*

Dieser Self-Join erzeugt eine Liste derjenigen Artikel aus der Tabelle **stock**, die von drei Herstellern geliefert werden. Die letzten zwei Bedingungen in der WHERE-Klausel bewirken, daß doppelt vorkommende Hersteller-Codes in den abgerufenen Datensätzen ausgeschlossen werden.

```
manu_code manu_code manu_code stock_num description

HRO       HSK       SMT             1 baseball gloves
ANZ       NRG       SMT             5 tennis racquet
ANZ       HRO       HSK           110 helmet
ANZ       HRO       PRC           110 helmet
ANZ       HRO       SHM           110 helmet
ANZ       HSK       PRC           110 helmet
ANZ       HSK       SHM           110 helmet
ANZ       PRC       SHM           110 helmet
HRO       HSK       PRC           110 helmet
HRO       HSK       SHM           110 helmet
HRO       PRC       SHM           110 helmet
HSK       PRC       SHM           110 helmet
ANZ       KAR       NKL           201 golf shoes
ANZ       HRO       NKL           205 3 golf balls
ANZ       HRO       KAR           301 running shoes
  .
  .
  .
HRO       PRC       SHM           301 running shoes
KAR       NKL       PRC           301 running shoes
KAR       NKL       SHM           301 running shoes
KAR       PRC       SHM           301 running shoes
NKL       PRC       SHM           301 running shoes
```

*Ergebnis 3-11*

Nehmen wir an, Sie wollen Datensätze aus der Tabelle **payroll** auswählen, um die Arbeitnehmer zu ermitteln, die mehr als ihr jeweiliger Abteilungsleiter verdienen. Dafür erstellen Sie den folgenden Self-Join:

```
SELECT emp.employee_num, emp.gross_pay, emp.level,
       emp.dept_num, mgr.employee_num, mgr.gross_pay,
       mgr.dept_num, mgr.level
    FROM payroll emp, payroll mgr
    WHERE emp.gross_pay > mgr.gross_pay
        AND emp.level < mgr.level
        AND emp.dept_num = mgr.dept_num
    ORDER BY 4
```

*Abfrage 3-12A*

Im folgenden Beispiel wird eine *korrelierte Unterabfrage* verwendet, um die Artikel mit den 10 höchsten Preisen abzurufen und diese sortiert aufzulisten:

```
SELECT order_num, total_price
    FROM items a
    WHERE 10 >
        (SELECT COUNT (*)
            FROM items b
            WHERE b.total_price < a.total_price)
    ORDER BY total_price
```

*Abfrage 3-12B*

Die folgenden 10 Datensätze werden zurückgeliefert:

```
order_num total_price

     1018      $15.00
     1013      $19.80
     1003      $20.00
     1005      $36.00
     1006      $36.00
     1013      $36.00
     1010      $36.00
     1013      $40.00
     1022      $40.00
     1023      $40.00
```

*Ergebnis 3-12*

Sie können eine ähnliche Abfrage erstellen, um die 10 Arbeitnehmer der Firma zu ermitteln, die am längsten zum Betrieb gehören.

Korrelierte und nicht korrelierte Unterabfragen sind in diesem Kapitel weiter unten beschrieben.

## Verwendung von Rowid-Werten

Sie können in einem Self-Join die unsichtbare Spalte *Rowid* (*Row Id* = Satznummer) verwenden, um doppelte Werte in einer Tabelle ausfindig zu machen. Im folgenden Beispiel ist die Bedingung `x.rowid != y.rowid` gleichbedeutend mit der Aussage "Satz `x` ist nicht derselbe Datensatz wie Satz `y`".

```
SELECT x.rowid, x.customer_num
    FROM cust_calls x, cust_calls y
    WHERE x.customer_num = y.customer_num
        AND x.rowid != y.rowid
```

*Abfrage 3-13*

Diese SELECT-Anweisung (Abfrage 3-13) wählt Daten aus der Tabelle **cust_calls** zweimal aus. Dabei werden der Tabelle die Alias-Namen **x** und **y** zugewiesen. Die Anweisung sucht nach doppelten Werten in der Spalte **customer_num** und nach den zugehörigen Rowids. Die folgenden beiden Sätze werden gefunden:

```
      rowid customer_num

        515          116
        769          116
```

*Ergebnis 3-13*

Die letzte Bedingung der vorherigen SELECT-Anweisung können Sie entweder so angeben

```
AND x.rowid != y.rowid
```

oder auf diese Weise

```
AND NOT x.rowid = y.rowid
```

Eine andere Möglichkeit, doppelte Werte mit Hilfe einer korrelierten Unterabfrage ausfindig zu machen, zeigt Abfrage 3-14:

```
SELECT x.customer_num, x.call_dtime
    FROM cust_calls x
    WHERE 1 <
        (SELECT COUNT (*) FROM cust_calls y
            WHERE x.customer_num = y.customer_num)
```

*Abfrage 3-14*

Abfrage 3-14 ermittelt aus der Spalte **customer_num** dieselben zwei doppelten Werte wie die vorhergehende Abfrage. Folgende Datensätze werden zurückgeliefert:

```
customer_num call_dtime

         116 1992-11-28 13:34
         116 1992-12-21 11:24
```

*Ergebnis 3-14*

Wie bereits beim Self-Join gezeigt, können Sie mit der Rowid die interne Nummer ermitteln, die zu einem Datensatz in der Datenbanktabelle gehört. In Wirklichkeit ist die Rowid ein unsichtbare Spalte, die in jeder Tabelle enthalten ist. Daß die Werte der Rowids aufeinander folgen, hat keine spezielle Bedeutung und kann auch anders sein, abhängig von der physikalischen Position der Daten im Chunk. Daher kann die Satznummer, die Sie erhalten, sich von der im Beispiel abgedruckten unterscheiden. In Kapitel 13 "Datenbankserver-Abfragen optimieren" werden Performance-Probleme und der Wert der Rowid erörtert. Besonders detailliert wird im Handbuch *INFORMIX-OnLine Administrator's Guide* auf die Rowid eingegangen.

```
SELECT rowid, * FROM manufact
```

*Abfrage 3-15*

Abfrage 3-15 verwendet in der SELECT-Klausel die Rowid und das Jokerzeichen *, um alle Spalten der Tabelle **manufact** und deren zugehörige Rowid abzurufen.

```
      rowid manu_code manu_name       lead_time

        257 SMT       Smith             3
        258 ANZ       Anza              5
        259 NRG       Norge             7
        260 HSK       Husky             5
        261 HRO       Hero              4
        262 SHM       Shimara          30
        263 KAR       Karsten          21
        264 NKL       Nikolus           8
        265 PRC       ProCycle          9
```

*Ergebnis 3-15*

Sie können die Rowid auch verwenden,wenn Sie eine bestimmte Spalte auswählen.

```
SELECT rowid, manu_code FROM manufact
```

*Abfrage 3-16*

Die SELECT-Anweisung liefert das folgende Ergebnis:

```
      rowid manu_code

        258 ANZ
        261 HRO
        260 HSK
        263 KAR
        264 NKL
        259 NRG
        265 PRC
        262 SHM
        257 SMT
```

*Ergebnis 3-16*

Sie können die Rowid auch in einer WHERE-Klausel verwenden, um Datensätze über ihre internen Satznummern abzurufen. Diese Methode ist praktisch, wenn es in der Tabelle keine andere eindeutige Spalte gibt. Als Rowid wird die Rowid des vorigen Beispiels verwendet:

```
SELECT * FROM manufact WHERE rowid = 263
```

*Abfrage 3-17*

Diese SELECT-Anweisung liefert nur einen Datensatz zurück:

```
manu_code manu_name       lead_time

KAR       Karsten           21
```

*Ergebnis 3-17*

## Die Funktion USER

Um weitere Informationen über eine Tabelle zu erhalten, kann man die Rowid mit der Funktion USER verbinden.

```
SELECT USER username, rowid FROM cust_calls
```

*Abfrage 3-18*

Abfrage 3-18 weist der Spalte USER die Überschrift username zu und liefert folgende Daten über die Tabelle **cust_calls**:

```
username            rowid

zenda                 257
zenda                 258
zenda                 259
zenda                 513
zenda                 514
zenda                 515
zenda                 769
```

*Ergebnis 3-18*

Wenn Sie die Rowid auswählen, können Sie die Funktion USER auch in der WHERE-Klausel verwenden.

```
SELECT rowid FROM cust_calls WHERE user_id = USER
```

*Abfrage 3-19*

Abfrage 3-19 liefert nur die Rowid derjenigen Datensätze zurück, die von dem Benutzer erfaßt wurden, der die Anweisung erteilt hat. Wenn z. B. der Benutzer **richc** diese SELECT-Anweisung erteilt, dann würde die Ausgabe folgendermaßen aussehen:

```
      rowid

        258
        259
```

*Ergebnis 3-19*

## Die Funktion DBSERVERNAME

Um herauszufinden, auf welchem Datenbankserver die aktuelle Datenbank liegt, können Sie unter **INFORMIX-OnLine** einer Abfrage das Schlüsselwort DBSERVERNAME (oder des Synonyms SITENAME) hinzufügen.

```
SELECT DBSERVERNAME server, tabid, rowid, USER username
    FROM systables
    WHERE tabid >= 105 OR rowid <= 260
    ORDER BY rowid
```

*Abfrage 3-20*

Abfrage 3-20 ermittelt den Servernamen und den Benutzernamen sowie die Rowid und die *Tabid*. Die Tabid ist eine fortlaufende Zahl, die Systemtabellen eindeutig bezeichnet. Die Anweisung weist den Ausdrücken DBSERVERNAME und USER Spaltenüberschriften zu und liefert diese 10 Sätze aus der Systemtabelle **systables** zurück:

```
server          tabid       rowid username

manatee             1         257 zenda
manatee             2         258 zenda
manatee             3         259 zenda
manatee             4         260 zenda
manatee           105         274 zenda
manatee           106        1025 zenda
manatee           107        1026 zenda
manatee           108        1027 zenda
manatee           109        1028 zenda
manatee           110        1029 zenda
```

*Ergebnis 3-20*

Beachten Sie: Sie sollten eine Rowid nie in einer permanenten Tabelle speichern oder die Rowid als Fremdschlüssel verwenden, da sich die Rowid verändern kann. Wenn z. B. eine Tabelle gelöscht wurde und anschließend mit externen Daten wieder aufgefüllt wird, werden alle Rowids von denen der gelöschten Tabelle abweichen.

Eine Beschreibung der Funktionen USER und DBSERVERNAME finden Sie in Kapitel 2 "Einfache SELECT-Anweisungen".

# Outer Joins

In Kapitel 2 "Einfache SELECT-Anweisungen" wurde gezeigt, wie einige einfache Joins erstellt und verwendet werden. Während ein einfacher Join zwei oder mehrere verknüpfte Tabellen ebenbürtig behandelt, behandelt ein *Outer Join* zwei oder mehrere verknüpfte Tabellen *asymmetrisch*. Ein Outer Join macht eine der Tabellen zur *dominanten* Tabelle über die anderen, *untergeordneten* Tabellen.

Es gibt vier Grundarten des Outer Joins:

- Ein einfacher Outer Join mit zwei Tabellen
- Ein einfacher Outer Join zu einer dritten Tabelle
- Ein Outer Join von einem einfachen Join zu einer dritten Tabelle
- Ein Outer Join von einem Outer Join zu einer dritten Tabelle

Diese vier Arten des Outer Joins werden in diesem Abschnitt dargestellt. Eine umfassendere Darstellung der Syntax, Verwendung und Logik des Outer Joins erhalten Sie im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

Bei einem *einfachen Join* enthält das Ergebnis nur die verbundenen Datensätze aus den Tabellen, die die Join-Bedingung erfüllen. *Datensätze, die die Join-Bedingungen nicht erfüllen, werden zurückgewiesen.*

Bei einem *Outer Join* enthält das Ergebnis nur die verbundenen Datensätze, die die Join-Bedingungen erfüllen. *Datensätze der dominanten Tabelle, die sonst zurückgewiesen werden würden, werden beibehalten, und dies selbst dann, wenn kein passender Satz in der untergeordneten Tabelle gefunden wurde.* Den Datensätzen der dominanten Tabelle, für die es keinen passenden Datensatz in der untergeordneten Tabelle gibt, wird vor Ausgabe der ausgewählten Spalte ein Datensatz mit NULL-Werten zugewiesen.

Ein Outer Join stellt an die untergeordnete Tabelle Bedingungen; und zwar entsprechend der Reihenfolge, in der die Join-Bedingung auf die Sätze der dominanten Tabelle angewendet wird. Die Bedingungen werden in der WHERE-Klausel gesetzt.

Ein Outer Join muß eine SELECT-Klausel, eine FROM-Klausel und eine WHERE-Klausel enthalten. Einen einfachen Join wandeln Sie in einen Outer Join um, indem Sie in der FROM-Klausel das Schlüsselwort OUTER direkt vor den Namen der untergeordneten Tabelle setzen. Sie können das Schlüsselwort OUTER mehr als einmal in Ihrer Abfrage angeben.

Bevor Sie Outer Joins verwenden, sollten Sie prüfen, ob Sie nicht auch mit einem oder mehreren einfachen Joins arbeiten können. Wenn Sie keine zusätzlichen Daten aus anderen Tabellen benötigen, erhalten Sie mit einem einfachen Join meistens dasselbe Ergebnis.

Um die Beispiele in diesem Abschnitt kurz zu halten, werden für die Tabellen Alias-Namen verwendet. Alias-Namen für Tabellen wurden im vorhergehenden Kapitel erläutert.

## Einfacher Join

Im folgenden sehen Sie ein Beispiel eines einfachen Joins mit den Tabellen **customer** und **cust_calls**, das bereits in Kapitel 2 "Einfache SELECT-Anweisungen" verwendet wurde:

```
SELECT c.customer_num, c.lname, c.company,
       c.phone, u.call_dtime, u.call_descr
    FROM customer c, cust_calls u
    WHERE c.customer_num = u.customer_num
```

*Abfrage 3-21*

Abfrage 3-21 liefert nur die Datensätze zurück, bei denen der Kunde einen Auftrag beim Kundendienst erteilt hat:

```
customer_num  106
lname         Watson
company       Watson & Son
phone         415-389-8789
call_dtime    1993-06-12 08:20
call_descr    Order was received, but two of the cans of
              ANZ tennis balls within the case were empty

customer_num  110
lname         Jaeger
company       AA Athletics
phone         415-743-3611
call_dtime    1993-07-07 10:24
call_descr    Order placed one month ago (6/7) not received.

customer_num  119
lname         Shorter
company       The Triathletes Club
phone         609-663-6079
call_dtime    1993-07-01 15:00
call_descr    Bill does not reflect credit from previous order

customer_num  121
lname         Wallack
company       City Sports
phone         302-366-7511
call_dtime    1993-07-10 14:05
call_descr    Customer likes our merchandise. Requests that we
              stock more types of infant joggers. Will call back
              to place order.

customer_num  127
lname         Satifer
company       Big Blue Bike Shop
phone         312-944-5691
call_dtime    1993-07-31 14:30
call_descr    Received Hero watches (item # 304) instead of
              ANZ watches

customer_num  116
lname         Parmelee
company       Olympic City
phone         415-534-8822
call_dtime    1992-11-28 13:34
call_descr    Received plain white swim caps (313 ANZ) instead
              of navy with team logo (313 SHM)

customer_num  116
lname         Parmelee
company       Olympic City
phone         415-534-8822
call_dtime    1992-12-21 11:24
call_descr    Second complaint from this customer! Received
              two cases right-handed outfielder gloves (1 HRO)
              instead of one case lefties.
```

*Ergebnis 3-21*

### Einfacher Outer Join mit zwei Tabellen

In diesem Beispiel werden dieselbe Auswahlliste, Tabellen und Vergleichsbedingungen wie im vorherigen Beispiel verwendet. Es wird hier jedoch ein einfacher Outer Join erstellt:

```
SELECT c.customer_num, c.lname, c.company,
       c.phone, u.call_dtime, u.call_descr
    FROM customer c, OUTER cust_calls u
    WHERE c.customer_num = u.customer_num
```

*Abfrage 3-22*

Das Schlüsselwort OUTER vor dem Tabellennamen **cust_calls** macht diese zur untergeordneten Tabelle. Ein Outer Join bewirkt, daß die Daten aller Kunden zurückgeliefert werden. Dabei ist es egal, ob sie einen Auftrag beim Kundendienst erteilt haben oder nicht. Aus der dominanten Tabelle **customer** werden alle Datensätze abgerufen. Den entsprechenden Datensätzen aus der untergeordneten Tabelle **cust_calls** werden NULL-Werte zugewiesen.

```
customer_num  101
lname         Pauli
company       All Sports Supplies
phone         408-789-8075
call_dtime
call_descr

customer_num  102
lname         Sadler
company       Sports Spot
phone         415-822-1289
call_dtime
call_descr

customer_num  103
lname         Currie
company       Phil's Sports
phone         415-328-4543
call_dtime
call_descr

customer_num  104
lname         Higgins
company       Play Ball!
phone         415-368-1100
call_dtime
call_descr

customer_num  105
lname         Vector
company       Los Altos Sports
phone         415-776-3249
call_dtime
call_descr

customer_num  106
lname         Watson
company       Watson & Son
phone         415-389-8789
call_dtime    1991-06-12 08:20
call_descr    Order was received, but two of the cans of
              ANZ tennis balls within the case were empty

customer_num  107
lname         Ream
company       Athletic Supplies
phone         415-356-9876
call_dtime
call_descr

customer_num  108
lname         Quinn
company       Quinn's Sports
phone         415-544-8729
call_dtime
call_descr

.
.
.
```

*Ergebnis 3-22*

## Outer Join von einem einfachen Join zu einer dritten Tabelle

Das folgende Beispiel zeigt einen Outer Join, der aus einem einfachen Join zu einer dritten Tabelle resultiert. Diese zweite Art eines Outer Joins ist auch als *einfacher geschachtelter Join* bekannt:

```
SELECT c.customer_num, c.lname, o.order_num,
       i.stock_num, i.manu_code, i.quantity
    FROM customer c, OUTER (orders o, items i)
    WHERE c.customer_num = o.customer_num
        AND o.order_num = i.order_num
        AND manu_code IN ('KAR', 'SHM')
    ORDER BY lname
```

*Abfrage 3-23*

Abfrage 3-23 führt zuerst einen einfachen Join mit den Tabellen **orders** und **items** durch und ruft die Daten aller Aufträge ab, bei denen der Hersteller-Code des Artikels KAR oder SHM ist. Anschließend führt die Anweisung einen Outer Join durch, um diese Daten mit Daten der dominanten Tabelle **customer** zu verbinden.

Die optionale ORDER BY-Klausel ordnet die Daten folgendermaßen:

```
customer_num lname            order_num stock_num manu_code quantity

         114 Albertson
         118 Baxter
         113 Beatty
         103 Currie
         115 Grant
         123 Hanlon                1020       301 KAR             4
         123 Hanlon                1020       204 KAR             2
         125 Henry
         104 Higgins
         110 Jaeger
         120 Jewell                1017       202 KAR             1
         120 Jewell                1017       301 SHM             2
         111 Keyes
         112 Lawson
         128 Lessor
         109 Miller
         126 Neelie
         122 O'Brian               1019       111 SHM             3
         116 Parmelee
         101 Pauli
         124 Putnum                1021       202 KAR             3
         108 Quinn
         107 Ream
         102 Sadler
         127 Satifer               1023       306 SHM             1
         127 Satifer               1023       105 SHM             1
         127 Satifer               1023       110 SHM             1
         119 Shorter               1016       101 SHM             2
         117 Sipes
         105 Vector
         121 Wallack               1018       302 KAR             3
         106 Watson
```

*Ergebnis 3-23*

## Outer Join von einem Outer Join zu einer dritten Tabelle

Abfrage 3-24 erstellt einen Outer Join, der aus einem Outer Join zu einer dritten Tabelle resultiert. Diese dritte Art ist auch als *geschachtelter Outer Join* bekannt:

```
SELECT c.customer_num, lname, o.order_num,
       stock_num, manu_code, quantity
    FROM customer c, OUTER (orders o, OUTER items i)
    WHERE c.customer_num = o.customer_num
        AND o.order_num = i.order_num
        AND manu_code IN ('KAR', 'SHM')
    ORDER BY lname
```

*Abfrage 3-24*

Die Abfrage führt zuerst einen Outer Join der Tabellen **orders** und **items** durch. Hierbei werden alle Daten aller Aufträge abgerufen, bei denen der Hersteller-Code des Artikels KAR oder SHM ist. Anschließend führt die Anweisung einen Outer Join durch, der diese Daten mit den Daten der dominanten Tabelle **customer** verbindet. Diese Abfrage behält die Auftragsnummern, die die vorherige Anweisung ausgeschlossen hat. Die Datensätze der Aufträge werden zurückgeliefert, bei denen kein Artikel einen der beiden Hersteller-Codes enthält. Die optionale ORDER BY-Klausel sortiert die Daten.

```
customer_num lname            order_num stock_num manu_code quantity

         114 Albertson
         118 Baxter
         113 Beatty
         103 Currie
         115 Grant                 1010
         123 Hanlon                1020        204 KAR             2
         123 Hanlon                1020        301 KAR             4
         125 Henry
         104 Higgins               1011
         104 Higgins               1001
         104 Higgins               1013
         104 Higgins               1003
         110 Jaeger                1008
         110 Jaeger                1015
         120 Jewell                1017        301 SHM             2
         120 Jewell                1017        202 KAR             1
         111 Keyes                 1009
         112 Lawson                1006
         128 Lessor
         109 Miller
         126 Neelie                1022
         122 O'Brian               1019        111 SHM             3
         116 Parmelee              1005
         101 Pauli                 1002
         124 Putnum                1021        202 KAR             3
         108 Quinn
         107 Ream
         102 Sadler
         127 Satifer               1023        110 SHM             1
         127 Satifer               1023        105 SHM             1
         127 Satifer               1023        306 SHM             1
         119 Shorter               1016        101 SHM             2
         117 Sipes                 1012
         117 Sipes                 1007
         105 Vector
         121 Wallack               1018        302 KAR             3
         106 Watson                1014
         106 Watson                1004
```

*Ergebnis 3-24*

Man kann die Bedingungen für einen Join, der aus einem Outer Join zu einer dritten Tabelle resultiert, auf zwei Arten erstellen. Obwohl die zwei untergeordneten Tabellen miteinander verknüpft sind, können Sie die dominante Ta-

belle mit jeder der untergeordneten Tabellen verknüpfen. Wenn die dominante Tabelle und die untergeordneten Tabellen über eine gemeinsame Spalte verfügen, werden die Ergebnisse dabei nicht beeinträchtigt.

## Outer Join von zwei Tabellen zu einer dritten Tabelle

Abfrage 3-25 zeigt einen Outer Join, der aus einem Outer Join einer jeden dieser zwei Tabellen zu einer dritten Tabelle resultiert. Bei diesem vierten Typ eines Outer Join sind verknüpfende Beziehungen *nur* zwischen der dominanten Tabelle und den untergeordneten Tabellen möglich:

```
SELECT c.customer_num, lname, o.order_num,
       order_date, call_dtime
    FROM customer c, OUTER orders o, OUTER cust_calls x
    WHERE c.customer_num = o.customer_num
        AND c.customer_num = x.customer_num
    ORDER BY lname
    INTO TEMP service
```

*Abfrage 3-25*

Abfrage 3-25 verknüpft die untergeordneten Tabellen **orders** und **cust_calls** einzeln mit der dominanten Tabelle **customer**; sie verknüpft nicht die untergeordneten Tabellen miteinander. Die INTO TEMP-Klausel schreibt das Ergebnis für weitere Änderungen oder Abfragen in eine temporäre Tabelle.

```
customer_num lname            order_num order_date call_dtime

         114 Albertson
         118 Baxter
         113 Beatty
         103 Currie
         115 Grant                 1010 06/17/1993
         123 Hanlon                1020 07/11/1993
         125 Henry
         104 Higgins               1003 05/22/1993
         104 Higgins               1001 05/20/1993
         104 Higgins               1013 06/22/1993
         104 Higgins               1011 06/18/1993
         110 Jaeger                1015 06/27/1993 1993-07-07 10:24
         110 Jaeger                1008 06/07/1993 1993-07-07 10:24
         120 Jewell                1017 07/09/1993
         111 Keyes                 1009 06/14/1993
         112 Lawson                1006 05/30/1993
         109 Miller
         128 Moore
         126 Neelie                1022 07/24/1993
         122 O'Brian               1019 07/11/1993
         116 Parmelee              1005 05/24/1993 1992-12-21 11:24
         116 Parmelee              1005 05/24/1993 1992-11-28 13:34
         101 Pauli                 1002 05/21/1993
         124 Putnum                1021 07/23/1993
         108 Quinn
         107 Ream
         102 Sadler
         127 Satifer               1023 07/24/1993 1993-07-31 14:30
         119 Shorter               1016 06/29/1993 1993-07-01 15:00
         117 Sipes                 1007 05/31/1993
         117 Sipes                 1012 06/18/1993
         105 Vector
         121 Wallack               1018 07/10/1993 1993-07-10 14:05
         106 Watson                1004 05/22/1993 1993-06-12 08:20
         106 Watson                1014 06/25/1993 1993-06-12 08:20
```

*Ergebnis 3-25*

Beachten Sie: Wenn in der vorherigen SELECT-Anweisung versucht worden wäre, eine Join-Bedingung zwischen den zwei untergeordneten Tabellen **o** und **x** zu erzeugen, wie es im folgenden Beispiel gezeigt wird, dann wäre dieser wechselseitige Outer Join durch eine Fehlermeldung angezeigt worden:

```
    WHERE o.customer_num = x.customer_num
```

*Abfrage 3-26*

# Unterabfragen in SELECT-Anweisungen

Eine SELECT-Anweisung, die in die WHERE-Klausel einer anderen SELECT-Anweisung *geschachtelt* ist, wird als *Unterabfrage* bezeichnet. Die SELECT-Anweisung kann auch in einer INSERT-, DELETE- oder UPDATE-Anweisung enthalten sein. Jede Unterabfrage muß eine SELECT- und eine FROM-Klausel enthalten und muß in Klammern stehen. Die Klammerung weist den Datenbankserver an, diese Operation zuerst durchzuführen.

Unterabfragen können *korreliert* oder *nicht korreliert* sein. Eine Unterabfrage (oder *innere* SELECT-Anweisung) ist korreliert, wenn der von ihr erzeugte Wert von einem Wert abhängt, den die *äußere* SELECT-Anweisung erzeugt. Die äußere SELECT-Anweisung enthält die innere SELECT-Anweisung. Jede andere Art einer Unterabfrage wird als nicht korreliert betrachtet.

Das wichtige Merkmal einer korrelierten Unterabfrage ist, daß sie wiederholt ausgeführt werden muß, da sie von einem Wert der äußeren SELECT-Anweisung abhängig ist. Die Unterabfrage wird für jeden Wert, den die äußere SELECT-Anweisung liefert, einmal ausgeführt. Eine nicht korrelierte Unterabfrage wird insgesamt nur einmal ausgeführt.

Häufig kann man zwei einzelne SELECT-Anweisungen durch eine einzige SELECT-Anweisung mit einer Unterabfrage ersetzen.

Unterabfragen in einer SELECT-Anweisung ermöglichen es Ihnen,

- einen Ausdruck mit dem Ergebnis einer anderen SELECT-Anweisung zu vergleichen.
- herauszufinden, ob ein Ausdruck in dem Ergebnis einer anderen SELECT-Anweisung enthalten ist.
- herauszufinden, ob eine andere SELECT-Anweisung Datensätze liefert.

In einer Unterabfrage wird die optionale WHERE-Klausel häufig verwendet, um die Suchbedingung einzuschränken.

Eine Unterabfrage wählt Werte aus und liefert diese an die erste oder äußere SELECT-Anweisung zurück. Eine Unterabfrage kann entweder keinen Wert, einen einzelnen Wert oder eine Anzahl von Werten zurückliefern.

- Wenn sie *keinen* Wert zurückliefert, dann liefert die Abfrage auch keine Datensätze zurück. Eine solche Unterabfrage entspricht einem NULL-Wert.
- Wenn sie *einen* Wert zurückliefert, dann liefert die Unterabfrage entweder einen Mengenausdruck zurück oder sie wählt genau einen Datensatz und eine Spalte aus. Eine solche Unterabfrage entspricht einem einzelnen Zahlen- oder CHAR-Wert.

- Wenn sie eine Liste oder *Anzahl* von Werten zurückliefert, dann liefert die Unterabfrage entweder einen Datensatz oder eine Spalte zurück.

Die folgenden Schlüsselwörter leiten eine Unterabfrage in der WHERE-Klausel einer SELECT-Anweisung ein:

- ALL
- ANY / SOME
- IN
- EXISTS

Sie können jeden der relationalen Operatoren zusammen mit ALL oder ANY verwenden, um etwas mit jedem (ALL) Wert zu vergleichen, den die Unterabfrage erzeugt. Oder um etwas mit einigen (ANY) Werten zu vergleichen, die die Unterabfrage erzeugt. Anstelle von ANY können Sie auch das Schlüsselwort SOME verwenden. Der Operator IN ist gleichbedeutend mit = ANY. Um die Umkehrung der Suchbedingung zu erzeugen, verwenden Sie das Schlüsselwort NOT oder einen anderen relationalen Operator.

Der Operator EXISTS überprüft, ob eine Unterabfrage überhaupt einen Wert ermittelt hat, d. h., ob das Ergebnis einer Unterabfrage ungleich Null ist.

Die vollständige Syntax zur Erzeugung einer Bedingung in einer Unterabfrage wird im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* erläutert. In Kapitel 13 "Datenbankserver-Abfragen optimieren" finden Sie ausführliche Informationen darüber, wie sich korrelierte und nicht korrelierte Unterabfragen auf die Performance auswirken.

## Das Schlüsselwort ALL

Um zu ermitteln, ob ein Vergleichsmuster auf jeden zurückgelieferten Wert zutrifft, stellen Sie der Unterabfrage das Schlüsselwort ALL voran. Wenn eine Unterabfrage keine Werte zurückliefert, dann ist die Suchbedingung *wahr* (true). (Wenn sie überhaupt keine Werte zurückliefert, dann ist die Bedingung für alle Null-Werte wahr.)

```
SELECT order_num, stock_num, manu_code, total_price
    FROM items
    WHERE total_price < ALL
        (SELECT total_price FROM items
            WHERE order_num = 1023)
```

*Abfrage 3-27*

Abfrage 3-27 listet die folgenden Daten für all die Aufträge auf, die einen Artikel enthalten, dessen Gesamtpreis (**total_price**) niedriger ist als der Gesamtpreis eines *jeden* Artikels, der im Auftrag 1023 enthalten ist:

```
order_num stock_num manu_code total_price

     1003         9 ANZ            $20.00
     1005         6 SMT            $36.00
     1006         6 SMT            $36.00
     1010         6 SMT            $36.00
     1013         5 ANZ            $19.80
     1013         6 SMT            $36.00
     1018       302 KAR            $15.00
```

*Ergebnis 3-27*

# Das Schlüsselwort ANY

Um zu ermitteln, ob ein Vergleichsmuster auf mindestens einen zurückgelieferten Wert zutrifft, stellen Sie der Unterabfrage das Schlüsselwort ANY (oder dessen Synonym SOME) voran. Wenn eine Unterabfrage keine Werte zurückliefert, dann ist die Suchbedingung *falsch* (false). Da keine Werte vorhanden sind, kann die Bedingung für keinen der Werte wahr sein.

```
SELECT DISTINCT order_num
    FROM items
    WHERE total_price > ANY
        (SELECT total_price
            FROM items
            WHERE order_num = 1005)
```

*Abfrage 3-28*

Abfrage 3-28 ermittelt die Auftragsnummern all der Aufträge, die einen Artikel enthalten, dessen Gesamtpreis größer als der Gesamtpreis *irgendeines* Artikels ist, der im Auftrag `1005` enthalten ist. Folgende Datensätze werden zurückgeliefert:

```
order_num

      1001
      1002
      1003
      1004
      1005
      1006
      1007
      1008
      1009
      1010
      1011
      1012
      1013
      1014
      1015
      1016
      1017
      1018
      1019
      1020
      1021
      1022
      1023
```

***Ergebnis 3-28***

# Unterabfragen, die genau einen Wert zurückliefern

Wenn Sie wissen, daß eine Unterabfrage *genau einen* Wert an die äußere Ebene der Abfrage zurückliefert,brauchen Sie die Schlüsselwörter ALL oder ANY nicht angeben. Eine Unterabfrage, die genau einen Wert zurückliefert, kann wie eine Funktion behandelt werden. Diese Art einer Unterabfrage wird häufig bei Mengenfunktionen verwendet, da diese immer einzelne Werte zurückliefern:

```
SELECT order_num FROM items
    WHERE stock_num = 9
        AND quantity =
            (SELECT MAX (quantity)
                FROM items
                WHERE stock_num = 9)
```

*Abfrage 3-29*

Abfrage 3-29 verwendet in der Unterabfrage die Mengenfunktion MAX, um die Nummer des Auftrags zu ermitteln, der die größte Anzahl Volleyball-Netze geordert hat. Sie liefert folgenden Datensatz zurück:

```
order_num

     1012
```

*Ergebnis 3-29*

Im folgenden Beispiel wird die Mengenfunktion MIN in der Unterabfrage verwendet. Damit werden diejenigen Artikel ausgewählt, bei denen der Gesamtpreis das Zehnfache des niedrigsten Preises übersteigt:

```
SELECT order_num, stock_num, manu_code, total_price
    FROM items x
    WHERE total_price >
        (SELECT 10 * MIN (total_price)
            FROM items
            WHERE order_num = x.order_num)
```

*Abfrage 3-30*

Abfrage 3-30 ermittelt folgende Datensätze :

```
order_num stock_num manu_code  total_price

     1003         8 ANZ            $840.00
     1018       307 PRC            $500.00
     1018       110 PRC            $236.00
     1018       304 HRO            $280.00
```

*Ergebnis 3-30*

# Korrelierte Unterabfragen

In Abfrage 3-31werden die zehn frühesten Verladedaten der Tabelle **orders** aufgelistet. In diesem Beispiel ist nach der Unterabfrage die ORDER BY-Klausel angegeben, um das Ergebnis zu sortieren. Dies ist erforderlich, da man in einer Unterabfrage keine ORDER BY-Klausel angeben kann.

```
SELECT po_num, ship_date FROM orders main
    WHERE 10 >
        (SELECT COUNT (DISTINCT ship_date)
            FROM orders sub
            WHERE sub.ship_date > main.ship_date)
            AND ship_date IS NOT NULL
    ORDER BY ship_date, po_num
```

*Abfrage 3-31*

Die Unterabfrage ist korreliert, da die Nummer, die von ihr ermittelt wird, von dem Wert der Spalte **main.ship_date** abhängig ist. Dieser Wert wird von der äußeren SELECT-Anweisung ermittelt. Aus diesem Grund muß die Unterabfrage für jeden Datensatz von neuem ausgeführt werden, den die äußere Abfrage ermittelt. Die Unterabfrage verwendet die Funktion COUNT, um einen Wert an die Hauptabfrage zurückzuliefern. Anschließend sortiert die ORDER BY-Klausel die Daten. Die Abfrage ermittelt diejenigen 13 Datensätze, die die zehn letzten Verladedaten aufweisen.

```
po_num      ship_date

4745        06/21/1993
278701      06/29/1993
429Q        06/29/1993
8052        07/03/1993
B77897      07/03/1993
LZ230       07/06/1993
B77930      07/10/1993
PC6782      07/12/1993
DM354331    07/13/1993
S22942      07/13/1993
MA003       07/16/1993
W2286       07/16/1993
Z55709      07/16/1993
C3288       07/25/1993
KF2961      07/30/1993
W9925       07/30/1993
```

*Ergebnis 3-31*

Wenn Sie so eine korrelierte Unterabfrage wie die eben vorgestellte auf eine sehr lange Tabelle anwenden, sollten Sie die Spalte **ship_date** indizieren. Sie verbessern dadurch die Performance. Ansonsten ist diese SELECT-Anweisung ziemlich uneffektiv, weil die Unterabfrage einmal für jeden Datensatz der Tabelle ausgeführt wird. In Kapitel 10 "Das Modell tunen" werden die Themen Indizierung und Performance erörtert.

# Das Schlüsselwort EXISTS

Das Schlüsselwort EXISTS kann man als *Kennzeichner* für *existierende* Datensätze verwenden. Die Unterabfrage ist nämlich nur dann wahr, wenn die äußere SELECT-Anweisung mindestens einen Datensatz findet.

```
SELECT UNIQUE manu_name, lead_time
    FROM manufact
    WHERE EXISTS
        (SELECT * FROM stock
            WHERE description MATCHES '*shoe*'
                AND manufact.manu_code = stock.manu_code)
```

*Abfrage 3-32a*

Häufig kann man eine Abfrage mit EXISTS so gestalten, daß sie gleichbedeutend mit einer Abfrage ist, die IN verwendet. Außerdem kann man IN durch =ANY ersetzen.

```
SELECT UNIQUE manu_name, lead_time
    FROM stock, manufact
    WHERE manufact.manu_code IN
        (SELECT manu_code FROM stock
            WHERE description MATCHES '*shoe*')
                AND stock.manu_code = manufact.manu_code
```

*Abfrage 3-32b*

Die beiden vorangegangenen Abfragen liefern Informationen über die Hersteller von Schuhen und die Lieferzeit des Produkts.

```
manu_name       lead_time

Anza               5
Hero               4
Karsten           21
Nikolus            8
ProCycle           9
Shimara           30
```

*Ergebnis 3-32*

Beachten Sie: Das Prädikat IN kann nicht verwendet werden, wenn die Unterabfrage eine TEXT- oder BYTE-Spalte enthält.

Um in einer der vorangegangenen Abfragen die Umkehrung der Suchbedingung anzugeben, fügen Sie bei IN oder EXISTS das Schlüsselwort NOT hinzu. Sie können NOT IN auch durch !=ALL ersetzen.

Es gibt hier zwei verschiedene Möglichkeiten. Bei der einen Möglichkeit wird der Datenbankserver weniger belastet ist als bei der anderen. Dies ist vom Datenbank-Design und der Größe der Tabellen abhängig. Um herauszufinden, welche Abfrage besser wäre, können Sie das Kommando SET EXPLAIN verwenden. Dieses Kommando liefert Ihnen eine Liste des Abfrageablaufs.

Die ausführliche Beschreibung des Kommandos SET EXPLAIN finden Sie in Kapitel 13 "Datenbankserver-Abfragen optimieren" sowie in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

```
SELECT customer_num, company FROM customer
    WHERE customer_num NOT IN
        (SELECT customer_num FROM orders
            WHERE customer.customer_num = orders.customer_num)

SELECT customer_num, company FROM customer
    WHERE NOT EXISTS
        (SELECT * FROM orders
            WHERE customer.customer_num = orders.customer_num)
```

*Abfrage 3-33*

Jede dieser beiden Anweisungen liefert die folgenden 11 Datensätze zurück. Die Datensätze zeigen diejenigen Kunden an, die bisher keine Aufträge erteilt haben:

```
customer_num company

         102 Sports Spot
         103 Phil's Sports
         105 Los Altos Sports
         107 Athletic Supplies
         108 Quinn's Sports
         109 Sport Stuff
         113 Sportstown
         114 Sporting Place
         118 Blue Ribbon Sports
         125 Total Fitness Sports
         128 Phoenix University
```

*Ergebnis 3-33*

Beachten Sie: Die Schlüsselwörter EXISTS und IN werden bei der Mengenoperation *Schnittmenge,* die Schlüsselwörter NOT EXISTS und NOT IN dagegen bei der Mengenoperation *Unterschiedsmenge* verwendet. Die Konzepte Schnittmenge und Unterschiedsmenge werden in diesem Kapitel weiter unten erläutert.

Abfrage 3-34 ermittelt in der Tabelle **stock** all die Artikel, die bisher nicht bestellt wurden. Um diese herauszufinden, wird eine Unterabfrage an die Tabelle **items** gestellt:

```
SELECT stock.* FROM stock
    WHERE NOT EXISTS
        (SELECT * FROM items
            WHERE stock.stock_num = items.stock_num
                AND stock.manu_code = items.manu_code)
```

*Abfrage 3-34*

Folgende Sätze werden zurückgeliefert:

```
stock_num manu_code description     unit_price unit unit_descr

      101 PRC       bicycle tires       $88.00 box  4/box
      102 SHM       bicycle brakes     $220.00 case 4 sets/case
      102 PRC       bicycle brakes     $480.00 case 4 sets/case
      105 PRC       bicycle wheels      $53.00 pair pair
      106 PRC       bicycle stem        $23.00 each each
      107 PRC       bicycle saddle      $70.00 pair pair
      108 SHM       crankset            $45.00 each each
      109 SHM       pedal binding      $200.00 case 4 pairs/case
      110 ANZ       helmet             $244.00 case 4/case
      110 HRO       helmet             $260.00 case 4/case
      112 SHM       12-spd, assmbld    $549.00 each each
      113 SHM       18-spd, assmbld    $685.90 each each
      201 KAR       golf shoes          $90.00 each each
      202 NKL       metal woods        $174.00 case 2 sets/case
      203 NKL       irons/wedge        $670.00 case 2 sets/case
      205 NKL       3 golf balls       $312.00 case 24/case
      205 HRO       3 golf balls       $312.00 case 24/case
      301 NKL       running shoes       $97.00 each each
      301 HRO       running shoes       $42.50 each each
      301 PRC       running shoes       $75.00 each each
      301 ANZ       running shoes       $95.00 each each
      302 HRO       ice pack             $4.50 each each
      303 KAR       socks               $36.00 box  24 pairs/box
      305 HRO       first-aid kit       $48.00 case 4/case
      306 PRC       tandem adapter     $160.00 each each
      308 PRC       twin jogger        $280.00 each each
      309 SHM       ear drops           $40.00 case 20/case
      310 SHM       kick board          $80.00 case 10/case
      310 ANZ       kick board          $84.00 case 12/case
      311 SHM       water gloves        $48.00 box  4 pairs/box
      312 SHM       racer goggles       $96.00 box  12/box
      312 HRO       racer goggles       $72.00 box  12/box
      313 SHM       swim cap            $72.00 box  12/box
      313 ANZ       swim cap            $60.00 box  12/box
```

*Ergebnis 3-34*

Beachten Sie: In Bezug auf die Anzahl der Unterabfragen einer SELECT-Anweisung gibt es keine logische Grenze. Es gibt jedoch eine physikalische Begrenzung, die die Länge einer jeden Anweisung betriff. Diese Grenze liegt wahrscheinlich aber außerhalb jeder praktischen Anweisung, die Sie erstellen.

Sie wollen vielleicht überprüfen, ob die Daten richtig in die Datenbank eingegeben wurden. Eine Möglichkeit, Fehler in einer Datenbank zu finden, besteht darin, eine Abfrage so zu gestalten, daß sie nur dann eine Ausgabe zurückliefert, wenn Fehler vorhanden sind. Eine Unterabfrage dieser Art dient als *Prüfabfrage.*

```
SELECT * FROM items
    WHERE total_price != quantity *
        (SELECT unit_price FROM stock
            WHERE stock.stock_num = items.stock_num
                AND stock.manu_code = items.manu_code)
```

*Abfrage 3-35*

Abfrage 3-35 liefert nur die Datensätze zurück, für die gilt: der Gesamtpreis eines Artikels von einem Auftrag stimmt nicht überein mit dem Preis pro Einheit (**unit_price**) der Tabelle **stock**, multipliziert mit der Auftragsmenge. Vorausgesetzt, daß kein Rabatt gewährt wurde, müssen solche Datensätze falsch in die Datenbank eingegeben worden sein:

```
item_num order_num stock_num manu_code quantity total_price

       1      1004         1 HRO             1     $960.00
       2      1006         5 NRG             5     $190.00
```

*Ergebnis 3-35*

Die Abfrage liefert nur dann Datensätze zurück, wenn Fehler auftreten. Werden die Informationen korrekt in die Datenbank eingefügt, so werden keine Datensätze ausgegeben.

# Mengenoperationen

Mit Hilfe der Mengenoperationen *Vereinigungsmenge (Union), Schnittmenge (Intersection)* und *Unterschiedsmenge (Difference)* können Sie die Daten der Datenbank manipulieren. Diese drei Operationen können Sie in SELECT-Anweisungen verwenden, um die Integrität Ihrer Datenbank zu überprüfen.Eine Überprüfung können Sie z.B. durchführen, nachdem Sie Daten verändert (update), eingefügt (insert) oder gelöscht (delete) haben. Diese Funktionen können z. B. nützlich sein, wenn man vor dem Löschen der Daten in der Originaltabelle die Daten zunächst in eine Hilfstabelle übertragen will, um sich dort davon zu überzeugen, daß auch die richtigen Daten gelöscht werden.

# Union

Die Operation Union (Vereinigungsmenge) verwendet das Schlüsselwort bzw. den Operator UNION, um zwei Abfragen zu einer *zusammengesetzten* Abfrage zu verbinden. Sie können zwischen zwei oder mehreren SELECT-Anweisungen das Schlüsselwort UNION verwenden, um diese Anweisungen zu vereinigen. Es wird eine temporäre Tabelle mit den Datensätzen erstellt, die in einer oder in allen Originaltabellen enthalten sind. (Beachten Sie: Der Operator UNION kann nicht innerhalb einer Unterabfrage oder in der Definition einer View verwendet werden.) Bild 3-1 zeigt die Mengenoperation UNION.

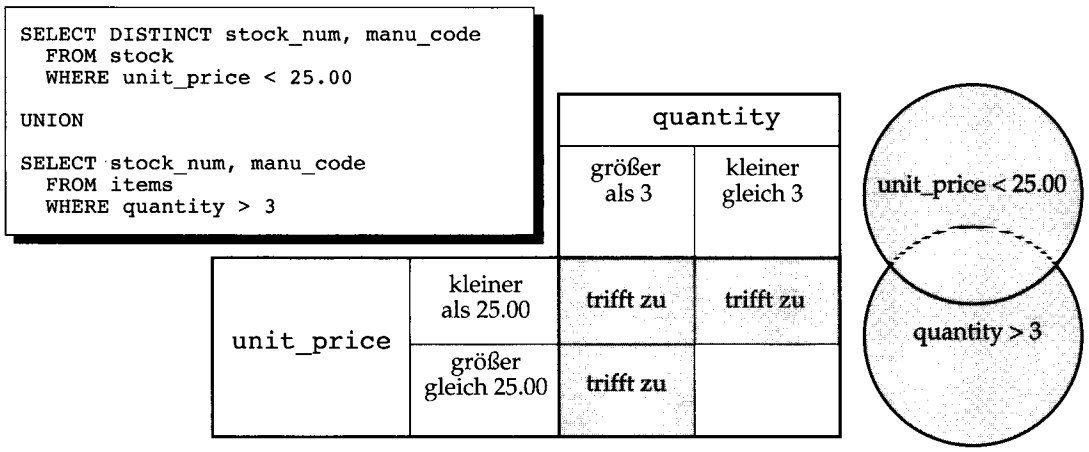



*Bild 3-1 Die Mengenoperation UNION*

Das Schlüsselwort UNION wählt aus den zwei Abfragen alle Datensätze aus, entfernt doppelt vorkommende Sätze und liefert die übriggebliebenen Sätze zurück. Da die Ergebnisse der Abfragen zu einem Ergebnis kombiniert werden, muß die Auswahlliste einer jeden Abfrage über dieselbe Spaltenanzahl verfügen. Weiterhin müssen die korrespondierenden Spalten, die aus jeder Tabelle ausgewählt werden, denselben Datentyp haben (CHARACTER-Spalten müssen gleich lang sein). Außerdem müssen diese Spalten entweder alle NULL-Werte zulassen oder alle NULL-Werte verbieten.

Abfrage 3-36 vereinigt die Spalten **stock_num** und **manu_code**, die Teil der Tabellen **stock** und **items** sind:

```
SELECT DISTINCT stock_num, manu_code
    FROM stock
    WHERE unit_price < 25.00

UNION

SELECT stock_num, manu_code
    FROM items
    WHERE quantity > 3
```

*Abfrage 3-36*

Abfrage 3-36 wählt nur diejenigen Artikel aus, deren Preis pro Einheit kleiner als $25.00 ist, oder die, von denen mehr als drei Stück bestellt wurden. Die Anweisung listet die entsprechenden Werte der Spalten **stock_num** und **manu_code** auf:

```
stock_num manu_code

        5 ANZ
        5 NRG
        5 SMT
        9 ANZ
      103 PRC
      106 PRC
      201 NKL
      301 KAR
      302 HRO
      302 KAR
```

*Ergebnis 3-36*

Wenn Sie eine ORDER BY-Klausel angeben, dann muß diese in der *letzten* SELECT-Anweisung enthalten sein. Um die Spalte anzugeben, nach der sortiert werden soll, müssen Sie eine Zahl verwenden, keinen Spaltennamen. Die Sortierung findet erst statt, nachdem die Mengenoperation durchgeführt wurde.

```
SELECT DISTINCT stock_num, manu_code
    FROM stock
    WHERE unit_price < 25.00

UNION

SELECT stock_num, manu_code
    FROM items
    WHERE quantity > 3
    ORDER BY 2
```

*Abfrage 3-37*

Abfrage 3-37 wählt dieselben Datensätze aus wie die vorherige SELECT-Anweisung, zeigt sie aber sortiert nach dem Hersteller- Code an:

```
stock_num manu_code

        5 ANZ
        9 ANZ
      302 HRO
      301 KAR
      302 KAR
      201 NKL
        5 NRG
      103 PRC
      106 PRC
        5 SMT
```

*Ergebnis 3-37*

Standardmäßig schließt das Schlüsselwort UNION doppelt vorkommende Datensätze aus. Wenn Sie die doppelt vorkommenden Werte beibehalten wollen, fügen Sie das Schlüsselwort ALL hinzu.

```
SELECT stock_num, manu_code
    FROM stock
    WHERE unit_price < 25.00

UNION ALL

SELECT stock_num, manu_code
    FROM items
    WHERE quantity > 3
    ORDER BY 2
    INTO TEMP stockitem
```

*Abfrage 3-38*

Abfrage 3-38 verwendet die Schlüsselwörter UNION ALL, um zwei SELECT-Anweisungen zu vereinigen. Die INTO TEMP-Klausel in der letzten SELECT-Anweisung bewirkt, daß das Ergebnis in eine temporäre Tabelle geschrieben wird. Es werden dieselben Datensätze wie im vorherigen Beispiel zurückgeliefert, aber es sind auch doppelte Werte enthalten.

```
stock_num manu_code

        9 ANZ
        5 ANZ
        9 ANZ
        5 ANZ
        9 ANZ
        5 ANZ
        5 ANZ
        5 ANZ
      302 HRO
      302 KAR
      301 KAR
      201 NKL
        5 NRG
        5 NRG
      103 PRC
      106 PRC
        5 SMT
        5 SMT
```

*Ergebnis 3-38*

Bei verbundenen Abfragen müssen die einander entsprechenden Spalten, die in den Auswahllisten angegeben sind, denselben Datentyp haben. Es ist aber nicht erforderlich, daß die Spalten dieselben Namen haben.

```
SELECT DISTINCT state
    FROM  customer
    WHERE customer_num BETWEEN 120 AND 125

UNION

SELECT DISTINCT code
    FROM state
    WHERE sname MATCHES '*a'
```

*Abfrage 3-39*

Abfrage 3-39 wählt die Spalte **state** aus der Tabelle **customer** und die korrespondierende Spalte **code** aus der Tabelle **state** aus. Sie liefert aus den Datensätzen die Abkürzungen derjenigen Länderkennzeichen zurück, bei denen die Kundennummer zwischen 120 und 125 liegt oder bei denen der Wert der Spalte **sname** mit `A` oder `a` endet.

```
state

AK
AL
AZ
CA
DE
FL
GA
IA
IN
LA
MA
MN
MT
NC
ND
NE
NJ
NV
OK
PA
SC
SD
VA
WV
```

*Ergebnis 3-39*

Bei zusammengesetzten Abfragen werden im Ergebnis die Spaltennamen oder Spaltenüberschriften der ersten SELECT-Anweisung angezeigt. Aus diesem Grund wird in diesem Beispiel der Spaltenname **state** von der ersten SELECT-Anweisung anstatt des Spaltennamens **code** von der zweiten verwendet.

Diese SELECT-Anweisung vereinigt drei Tabellen miteinander. Die maximale Anzahl der Unions hängt von der jeweiligen Anwendung und von Speicherbegrenzungen ab.

```
SELECT stock_num, manu_code
    FROM stock
    WHERE unit_price > 600.00

UNION ALL

SELECT stock_num, manu_code
    FROM catalog
    WHERE catalog_num = 10025

UNION ALL

SELECT stock_num, manu_code
    FROM items
    WHERE quantity = 10
    ORDER BY 2
```

*Abfrage 3-40*

Die obige zusammengesetzte Abfrage 3-40 wählt diejenigen Artikel aus, bei denen in der Tabelle **stock** der Preis pro Einheit größer als $600 ist oder bei denen der Wert in der Spalte **catalog_num** in der Tabelle **catalog** `10025` oder bei denen die Menge (**quantity**) in der Tabelle **items** gleich `10` ist. Die Datensätze werden anschließend nach den Werten der Spalte **manu_code** sortiert:

```
stock_num manu_code

        5 ANZ
        9 ANZ
        8 ANZ
        4 HSK
        1 HSK
      203 NKL
        5 NRG
      106 PRC
      113 SHM
```

*Ergebnis 3-40*

Die vollständige Syntax der SELECT-Anweisung und des Operators UNION sind im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* beschrieben. In den Kapiteln 5 und 6 dieses Handbuchs sowie in den Freigabemitteilungen und der Dokumentation zum jeweiligen Profukt finden Sie Informationen speziell für **INFORMIX-4GL**- und **INFORMIX-ESQL/C**. Außerdem finden Sie dort Informationen über spezifische Einschränkungen bei der Verwendung der INTO-Klausel und bei zusammengesetzten Abfragen.

In Abfrage 3-41 wird eine zusammengesetzte Abfrage verwendet, um ausgewählte Daten in eine temporäre Tabelle zu schreiben. Anschließend folgt eine einfache Abfrage, die die Daten sortiert . Die zusammengesetzte und die einfache Abfrage müssen mit einem Semikolon voneinander getrennt werden.

In der zusammengesetzten Abfrage wird in der Auswahlliste ein Literal verwendet, um bei der Ausgabe die jeweiligen Teile der Vereinigungsmenge zu kennzeichnen, so daß diese später unterschieden werden können. Das Kennzeichen wird als **sortkey** betitelt. Diese einfache Abfrage verwendet das Kennzeichen als Sortierschlüssel, um die abgerufenen Datensätze zu sortieren.

```
SELECT '1' sortkey, lname, fname, company,
       city, state, phone
    FROM customer x
    WHERE state = 'CA'

UNION

SELECT '2' sortkey, lname, fname, company,
       city, state, phone
    FROM customer y
    WHERE state <> 'CA'
    INTO TEMP calcust;

SELECT * FROM calcust
    ORDER BY 1
```

*Abfrage 3-41*

Dieses Abfragenpaar erstellt eine Liste, in der die Kunden aus Kalifornien zuerst erscheinen, die am meisten Aufträge erteilt haben.

```
sortkey  1
lname    Albertson
fname    Frank
company  Sporting Place
city     Redwood City
state    CA
phone    415-886-6677

sortkey  1
lname    Baxter
fname    Dick
company  Blue Ribbon Sports
city     Oakland
state    CA
phone    415-655-0011

sortkey  1
lname    Beatty
fname    Lana
company  Sportstown
city     Menlo Park
state    CA
phone    415-356-9982

sortkey  1
lname    Currie
fname    Philip
company  Phil's Sports
city     Palo Alto
state    CA
phone    415-328-4543

sortkey  1
lname    Grant
fname    Alfred
company  Gold Medal Sports
city     Menlo Park
state    CA
phone    415-356-1123
.
.
.
sortkey  2
lname    Satifer
fname    Kim
company  Big Blue Bike Shop
city     Blue Island
state    NY
phone    312-944-5691

sortkey  2
lname    Shorter
fname    Bob
company  The Triathletes Club
city     Cherry Hill
state    NJ
phone    609-663-6079
```

*Ergebnis 3-41*

# Schnittmenge

Eine *Schnittmenge* aus zwei Datensatzmengen erzeugt eine Tabelle, die nur diejenigen Datensätze enthält, die in beiden Originaltabellen vorhanden sind. Unterabfragen, die eine Schnittmenge zweier Mengen anzeigen, leiten Sie mit den Schlüsselwörtern EXIST oder IN ein. Bild 3-2 stellt die Mengenoperation Schnittmenge dar.

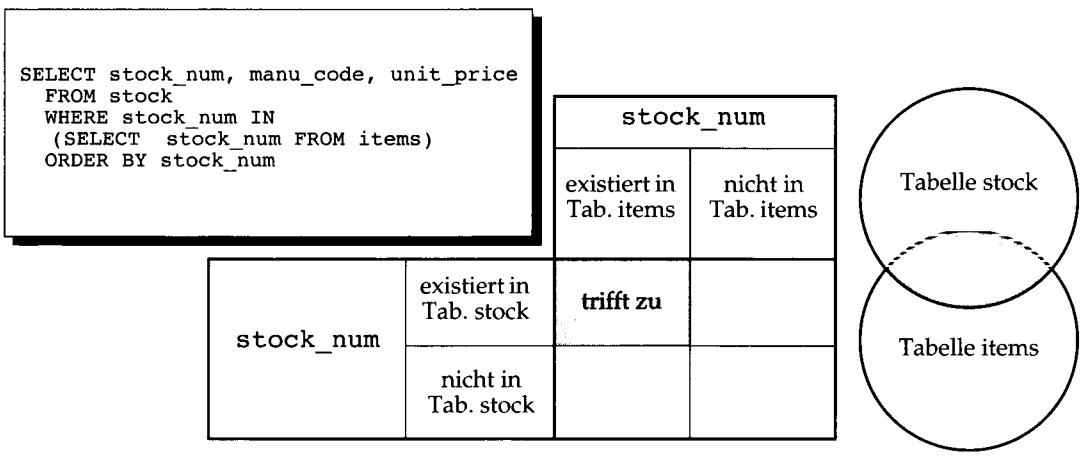



*Bild 3-2* *Die Mengenoperation Schnittmenge*

Im folgenden Beispiel einer geschachtelten SELECT-Anweisung wird die Schnittmenge aus den Tabellen **stock** und **items** gezeigt:

```
SELECT stock_num, manu_code, unit_price
    FROM stock
    WHERE stock_num IN
        (SELECT stock_num FROM items)
    ORDER BY stock_num
```

*Abfrage 3-42*

Das Ergebnis enthält alle Elemente aus beiden Mengen und liefert die folgenden 57 Datensätze zurück:

| stock_num | manu_code | unit_price |
|---|---|---|
| 1 | HRO | $250.00 |
| 1 | HSK | $800.00 |
| 1 | SMT | $450.00 |
| 2 | HRO | $126.00 |
| 3 | HSK | $240.00 |
| 3 | SHM | $280.00 |
| 4 | HRO | $480.00 |
| 4 | HSK | $960.00 |
| 5 | ANZ | $19.80 |
| 5 | NRG | $28.00 |
| 5 | SMT | $25.00 |
| 6 | ANZ | $48.00 |
| 6 | SMT | $36.00 |
| 7 | HRO | $600.00 |
| 8 | ANZ | $840.00 |
| 9 | ANZ | $20.00 |
| 101 | PRC | $88.00 |
| 101 | SHM | $68.00 |
| 103 | PRC | $20.00 |
| 104 | PRC | $58.00 |
| 105 | PRC | $53.00 |
| 105 | SHM | $80.00 |
| 109 | PRC | $30.00 |
| 109 | SHM | $200.00 |
| 110 | ANZ | $244.00 |
| 110 | HRO | $260.00 |
| 110 | HSK | $308.00 |
| 110 | PRC | $236.00 |
| 110 | SHM | $228.00 |
| 111 | SHM | $499.99 |
| 114 | PRC | $120.00 |
| 201 | ANZ | $75.00 |
| 201 | KAR | $90.00 |
| 201 | NKL | $37.50 |
| 202 | KAR | $230.00 |
| 202 | NKL | $174.00 |
| 204 | KAR | $45.00 |
| 205 | ANZ | $312.00 |
| 205 | HRO | $312.00 |
| 205 | NKL | $312.00 |
| 301 | ANZ | $95.00 |
| 301 | HRO | $42.50 |
| 301 | KAR | $87.00 |
| 301 | NKL | $97.00 |
| 301 | PRC | $75.00 |
| 301 | SHM | $102.00 |
| 302 | HRO | $4.50 |
| 302 | KAR | $5.00 |
| 303 | KAR | $36.00 |
| 303 | PRC | $48.00 |
| 304 | ANZ | $170.00 |
| 304 | HRO | $280.00 |
| 306 | PRC | $160.00 |
| 306 | SHM | $190.00 |
| 307 | PRC | $250.00 |
| 309 | HRO | $40.00 |
| 309 | SHM | $40.00 |

***Ergebnis 3-42***

# Unterschiedsmenge

Eine *Unterschiedsmenge* zwischen zwei Datensatzmengen erzeugt eine Tabelle, die diejenigen Datensätze aus der *ersten* Menge enthält, die nicht in der zweiten Menge vorhanden sind. Unterabfragen, die die Unterschiedsmenge zwischen zwei Datenmengen ergeben, leiten Sie mit den Schlüsselwörtern NOT EXISTS oder NOT IN ein. Bild 3-3 stellt die Mengenoperation Unterschiedsmenge dar.

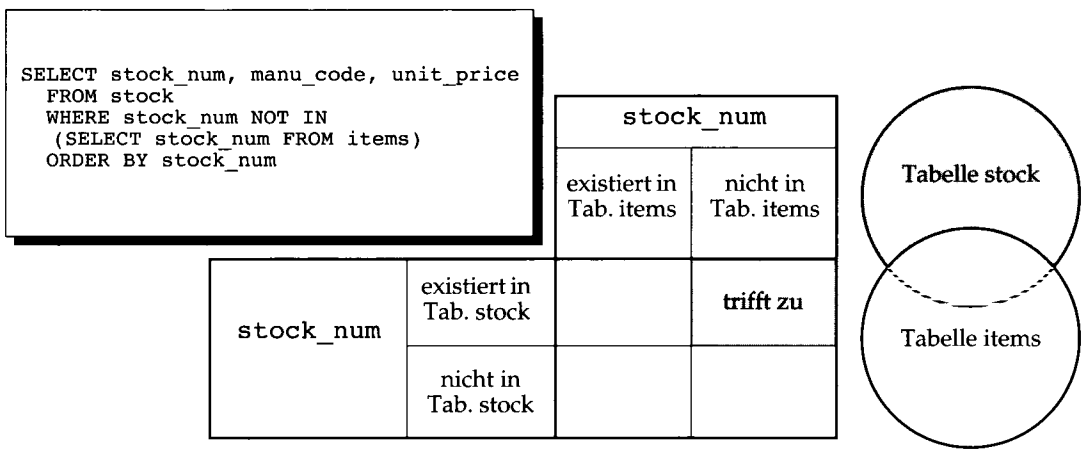



*Bild 3-3* ***Die Mengenoperation Unterschiedsmenge***

Im folgenden Beispiel einer geschachtelten SELECT-Anweisung wird die Unterschiedsmenge zwischen den Tabellen **stock** und **items** gezeigt:

```
SELECT stock_num, manu_code, unit_price
    FROM stock
    WHERE stock_num NOT IN
        (SELECT stock_num FROM items)
    ORDER BY stock_num
```

*Abfrage 3-43*

Das Ergebnis enthält all die Elemente, die nur in der ersten Menge vorhanden sind. Die folgenden 17 Datensätze werden zurückgeliefert:

```
stock_num manu_code unit_price

      102 PRC          $480.00
      102 SHM          $220.00
      106 PRC           $23.00
      107 PRC           $70.00
      108 SHM           $45.00
      112 SHM          $549.00
      113 SHM          $685.90
      203 NKL          $670.00
      305 HRO           $48.00
      308 PRC          $280.00
      310 ANZ           $84.00
      310 SHM           $80.00
      311 SHM           $48.00
      312 HRO           $72.00
      312 SHM           $96.00
      313 ANZ           $60.00
      313 SHM           $72.00
```

*Ergebnis 3-43*

# Zusammenfassung

Dem vorliegenden Kapitel lagen die Konzepte zugrunde, die in Kapitel 2 "Einfache SELECT-Anweisungen" vorgestellt wurden. Es beschrieb Syntax-Beispiele und Ergebnisse von komplexeren SELECT-Anweisungen. Diese werden verwendet, um Abfragen von einer relationalen Datenbank durchzuführen.

- Die Klauseln GROUP BY und HAVING können zusammen mit Mengenfunktionen verwendet werden, um Datensatzgruppen zurückzuliefern und um Bedingungen an diese Gruppen zu stellen.
- Mit *Rowid* können die internen Satznummern aus den Tabellen und Systemtabellen abgerufen werden. Mit der *Tabid*, der Tabellennummer, können die jeweiligen Tabellen eindeutig identifiziert werden.
- Mit Hilfe von Self-Joins kann eine Tabelle mit sich selbst verknüpft werden, um Werte in einer Spalte mit anderen Werten derselben Spalte zu vergleichen und um doppelt vorkommende Werte zu identifizieren.
- Outer Joins ermöglichen es, zwei oder mehr Tabellen unsymetrisch zu behandeln; dieses Kapitel zeigte Beispiele zu den vier Arten des Outer Join.

- Dieses Kapitel beschrieb, wie man korrelierte und nicht korrelierte Unterabfragen erstellt, wobei eine SELECT-Anweisung in die WHERE-Klausel einer anderen SELECT-Anweisung geschachtelt wird. Außerdem wurde auch die Verwendung der Mengenfunktionen in Unterabfragen vorgestellt.
- Das Kapitel zeigte die Verwendung der Schlüsselwörter ALL, ANY, EXISTS, IN und SOME beim Erstellen von Unterabfragen, und erläuterte, wie sich das Hinzufügen des Schlüsselwortes NOT oder eines relationalen Operators auswirkt.
- Die Mengenfunktionen Vereinigungsmenge, Schnittmenge und Unterschiedsmenge wurden vorgestellt.
- Mit den Schlüsselwörtern UNION und UNION ALL können Sie kombinierte Abfragen aus zwei oder mehreren SELECT-Anweisungen erstellen.

Kapitel

4

# Datenmanipulation

# Kapitelüberblick

Das Verändern von Daten unterscheidet sich grundlegend vom bloßen Abfragen der Daten. Daten abzufragen bedeutet, den Inhalt der Tabellen lediglich zu *untersuchen*. Daten zu verändern bedeutet, den Inhalt der Tabellen zu *verändern*.

Wenn die Hardware oder Betriebssystemsoftware während einer Abfrage ausfällt, kann dies ernsthafte Folgen für die Anwendung haben. Die Datenbank selbst wird dadurch aber nicht beeinträchtigt. Fällt das System jedoch während eines Änderungsvorgangs aus, dann ist der Stand der Datenbank selbst nicht vorhersagbar. Die Folgen eines Systemausfalls können also sehr weitreichend sein. Bevor Sie Datensätze in einer Datenbank löschen, einfügen oder verändern, sollten folgende Fragen geklärt werden:

- Ist der Benutzerzugriff auf die Datenbank und die Tabellen sicher, d. h. wurden die Berechtigungen auf Datenbank und Tabellen für die Datenbankbenutzer sinnvoll eingeschränkt?
- Wahren die veränderten Daten die bereits existierende Integrität der Datenbank?
- Werden Systeme verwendet, die die Datenbank relativ sicher machen gegen äußere Ereignisse, die einen System- oder Hardwareausfall bewirken könnten?

Geraten Sie nicht in Panik, wenn Sie nicht jede Frage mit ja beantworten können. Für all diese Probleme sind in den Datenbankserver Lösungen eingebaut. Diese Lösungen werden in diesem Kapitel erläutert. Die Kapitel 8 bis 11 dieses Handbuchs gehen auf diese Punkte genauer ein.

# Datenmanipulations-Anweisungen

Die folgenden drei Anweisungen verändern Daten:

- DELETE
- INSERT
- UPDATE

Diese SQL-Anweisungen sind im Vergleich zu komplexen SELECT-Anweisungen relativ einfach. Sie sollten sie aber vorsichtig verwenden, da sie den Inhalt der Datenbank verändern.

## Datensätze löschen

Die Anweisung DELETE löscht aus einer Tabelle einen beliebigen Datensatz oder eine Kombination von Datensätzen. Sobald eine Transaktion abgeschlossen ist, gibt es keine Möglichkeit mehr, den gelöschten Satz zurückzugewinnen. (Transaktionen werden im Abschnitt "Die Aktualisierung wird unterbrochen" auf Seite 4-24 erläutert. Nehmen wir vorläufig an, daß eine Transaktion und eine Anweisung identisch sind.)

Wenn Sie einen Datensatz gelöscht haben, müssen Sie auch darauf achten, daß Sie diejenigen Datensätze aus anderen Tabellen löschen, deren Werte von dem gelöschten Satz abhängen. Wenn Ihre Datenbank jedoch mit Referenz-Constraints arbeitet, können Sie mit der Option ON DELETE CASCADE in den Anweisungen CREATE TABLE- bzw. ALTER TABLE dafür sorgen, daß automatisch diejenigen Datensätze gelöscht werden, deren Werte von dem zu löschenden Datensatz abhängen. Weitere Informationen über Referenz-Contraints und über die Option ON DELETE CASCADE erhalten Sie im Abschnitt "Referentielle Integrität" auf Seite 4-21.

### Alle Datensätze einer Tabelle löschen

Die Anweisung DELETE gibt eine Tabelle an und enthält normalerweise eine WHERE-Klausel. Die WHERE-Klausel kennzeichnet den Datensatz oder die Datensätze, die aus der Tabelle gelöscht werden sollen. Wenn die WHERE-Klausel weggelassen wird, dann werden alle Datensätze gelöscht. *Führen Sie die folgende Anweisung* ***nicht*** *aus*:

```
DELETE FROM customer
```

Da diese DELETE-Anweisung keine WHERE-Klausel enthält, würden alle Datensätze aus der Tabelle **customer** gelöscht. Wenn Sie mit den Menü-Optionen von **DB-Access** oder **INFORMIX-SQL** arbeiten und versuchen, ohne eine

Bedingung zu löschen, dann werden Sie gewarnt und um Bestätigung gebeten. In einem Programm wird ein solcher Löschvorgang jedoch ohne Warnung ausgeführt.

## Eine bekannte Anzahl von Sätzen löschen

Die WHERE-Klausel hat in einer DELETE-Anweisung dieselbe Form wie in einer SELECT-Anweisung. Sie verwenden die WHERE-Klausel , um genau einen Satz oder mehrere Sätze zu bestimmen, die gelöscht werden sollen. Sie können zum Beispiel einen Kunden mit einer bestimmten Kundennummer löschen:

```
DELETE FROM customer WHERE customer_num = 175
```

In diesem Beispiel ist sichergestellt, daß höchstens ein Satz gelöscht wird, da die Spalte **customer_num** mit einem Unique-Constraint belegt ist.

### Eine unbekannte Anzahl von Sätzen löschen

Sie können Datensätze z. B. auch anhand nicht indizierter Spalten auswählen:

```
DELETE FROM customer WHERE company = 'Druid Cyclery'
```

Diese Anweisung könnte mehr als einen Datensatz löschen, da die zu überprüfende Spalte über keinen Unique-Constraint verfügt. (Druid Cyclery könnte zwei Geschäfte besitzen, die beide den gleichen Namen, aber unterschiedliche Kundennummern haben.)

Wählen Sie aus der Tabelle **customer** die Anzahl derjenigen Datensätze aus, die den Wert Druid Cyclery enthalten. Auf diese Weise können Sie herausfinden, wie viele Sätze von der DELETE-Anweisung betroffen sein könnten.

```
SELECT COUNT(*) FROM customer WHERE company = 'Druid Cyclery'
```

Sie können die Datensätze auch auswählen und anzeigen lassen, um sicher zu gehen, daß dies die Sätze sind, die Sie löschen wollen.

Wenn eine Datenbank mehreren Benutzern gleichzeitig zur Verfügung steht, dann liefert die Überprüfung mit einer SELECT-Anweisung nur einen Näherungswert. In der Zeit zwischen der Ausführung der SELECT-Anweisung

und der nachfolgenden DELETE-Anweisung könnten andere Benutzer die Tabelle und somit auch das Ergebnis verändert haben. In diesem Beispiel könnte ein anderer Benutzer

- einen neuen Datensatz für einen anderen Kunden mit dem Namen Druid Cyclery eingefügt haben
- einen oder mehrere der Sätze, die den Wert Druid Cyclery enthalten, gelöscht haben, bevor Sie dies tun
- bei einem Satz mit dem Wert Druid Cyclery den Firmennamen geändert haben; oder er könnte bei einigen anderen Kunden den Firmennamen auf Druid Cyclery geändert haben.

Es ist zwar sehr *unwahrscheinlich*, daß andere Benutzer in einer so kurzen Zeit diese Dinge ausführen , aber es ist möglich. Das gleiche Problem betrifft auch die UPDATE-Anweisung. Wege, wie man an dieses Problem herangeht, sind im Abschnitt "Parallelbearbeitung und Sperren" auf Seite 4-31 und ausführlicher in Kapitel 7 "Programmieren für Mehrneutzer-Betrieb" erläutert.

Ein anderes Problem ist ein Hardware- oder Softwareausfall vor Beendigung der Anweisung. In diesem Fall könnte die Datenbank keinen Satz, einige Sätze oder alle angegebenen Sätze gelöscht haben. Der *Status* der Datenbank ist unbekannt. Diese Situation können Sie dadurch verhindern, daß Sie mit einer Transaktionsprotokollierung arbeiten; dies wird im Abschnitt "Die Aktualisierung wird unterbrochen" auf Seite 4-24 erläutert.

## Komplizierte Lösch-Bedingungen

Die WHERE-Klausel kann in einer DELETE-Anweisung fast so kompliziert sein wie in einer SELECT-Anweisung. Die WHERE-Klausel kann mehrere Bedingungen, die durch AND und OR verknüpft sind, und Unterabfragen enthalten.

Nehmen wir an, Sie stellen fest, daß einige Sätze in der Tabelle **stock** mit einem falschen Hersteller-Code eingegeben wurden. Anstatt diese zu aktualisieren, wollen Sie sie lieber löschen, so daß sie erneut eingegeben werden können. Sie wissen, daß es zu diesen Sätzen keine übereinstimmenden Sätze in der Tabelle **manufact** gibt - im Gegensatz zu den richtigen Sätzen. Deshalb ist es möglich, die folgende DELETE-Anweisung einzugeben:

```
DELETE FROM stock
    WHERE 0 = (SELECT COUNT(*) FROM manufact
                  WHERE manufact.manu_code = stock.manu_code)
```

Die Unterabfrage zählt Datensätze, die mit der Tabelle **manufact** übereinstimmen; bei einem richtigen Datensatz aus der Tabelle **stock** ist die Anzahl 1, bei einem falschen 0. Die letzteren Datensätze werden zum Löschen ausgewählt.

Sie können eine DELETE-Anweisung mit komplizierten Bedingungen erstellen, indem Sie zuerst eine SELECT-Anweisung entwickeln, die genau die Sätze zurückliefert, die gelöscht werden sollen. Geben Sie hierbei SELECT * an; wenn die Anweisung die gewünschten Sätze zurückliefert, dann ersetzen Sie SELECT * durch DELETE und führen die Anweisung erneut aus.

In der WHERE-Klausel einer DELETE-Anweisung kann keine Unterabfrage verwendet werden, die dieselbe Tabelle überprüft. Das bedeutet, wenn Sie aus der Tabelle **stock** löschen wollen, dann können Sie in der WHERE-Klausel keine Unterabfrage verwenden, die auch aus der Tabelle **stock** auswählt.

Der Schlüssel dieser Regel liegt in der FROM-Klausel. Wenn eine Tabelle in der FROM-Klausel einer DELETE-Anweisung genannt wird, dann kann diese Tabelle nicht in der FROM-Klausel der Unterabfrage einer DELETE-Anweisung angegeben werden.

# Datensätze einfügen

Die INSERT-Anweisung fügt einen oder mehrere Datensätze in eine Tabelle ein. Diese Anweisung verfügt über zwei grundlegende Funktionen: sie kann einen neuen Datensatz erstellen, indem sie die Spaltenwerte verwendet, die Sie eingeben; oder sie kann eine Gruppe neuer Datensätze erstellen, indem sie ausgewählte Daten anderer Tabellen verwendet.

## Einzelne Datensätze

Die INSERT-Anweisung erstellt in ihrer einfachsten Form einen neuen Satz aus einer Reihe von Spaltenwerten und fügt diesen Satz in die Tabelle ein. Dieses Beispiel fügt einen Satz in die Tabelle **stock** ein:

```
INSERT INTO stock
      VALUES(115, 'PRC', 'tire pump', 108, 'box', '6/box')
```

Die Tabelle **stock** enthält folgende Spalten:

- **stock_num**, eine Nummer, die den Warentyp identifiziert
- **manu_code**, ein Fremdschlüssel zur Tabelle **manufact**
- **description**
- **unit_price**

- **unit** (Maßeinheit)
- **unit_descr** (Beschreibung der Maßeinheit)

Beachten Sie: Die Werte, die im vorangegangenen Beispiel in der VALUES-Klausel aufgelistetet sind, entsprechen genau den Spalten dieser Tabelle. Um eine VALUES-Klausel erstellen zu können, müssen Sie sowohl die Spalten der Tabelle als auch deren Reihenfolge von der ersten bis zur letzten kennen.

### Zulässige Spaltenwerte

Die VALUES-Klausel läßt nur konstante Werte zu, keine *Ausdrücke*. Sie können folgende Werte eingeben:

- Literale Zahlen
- Literale Datetime-Werte
- Literale Intervall-Werte
- Zeichenketten in Anführungszeichen
- Das Wort NULL für NULL-Werte
- Das Wort TODAY für das aktuelle Tagesdatum
- Das Wort CURRENT für das aktuelle Datum und die aktuelle Zeit
- Das Wort USER für Ihren Benutzernamen
- Das Wort DBSERVERNAME (oder SITENAME) für den Rechnernamen, auf dem der Datenbankserver läuft.

Wenn Sie versuchen, in eine Spalte einer Tabelle NULL-Werte einzufügen, für die keine NULL-Werte zugelassen sind, wird die Anweisung zurückgewiesen. Ebenso kann es sein, daß eine Spalte einer Tabelle keine doppelt vorkommenden Werte zuläßt. Wenn Sie für so eine Spalte einen Wert angeben, der bereits vorhanden ist, dann wird die Anweisung zurückgewiesen. Es kann auch sein, daß einige Spalten die möglichen Spaltenwerte *einschränken*. Diese Einschränkungen werden den Spalten durch Datenintegritäts-Constraints zugewiesen. Weitere Informationen zu Dateneinschränkungen erhalten Sie im Abschnitt "Datenbankberechtigungen" auf Seite 4-17.

In einer Tabelle kann es nur eine Spalte vom Datentyp SERIAL geben. Die Werte von SERIAL-Spalten werden vom Datenbankserver erzeugt. Um dies zu erreichen, geben Sie in Ihrer INSERT-Anweisung für die SERIAL-Spalte den Wert Null (0) an. Der Datenbankserver erzeugt dann jeweils den nächsten fortlaufenden Wert. In SERIAL-Spalten sind NULL-Werte nicht erlaubt.

Sie können in einer SERIAL-Spalte auch einen Wert ungleich Null (0) angeben (sofern dieser Wert in der Spalte nicht bereits vorhanden ist); der Datenbankserver verwendet dann diesen Wert. Dieser Wert ungleich Null kann jedoch

einen neuen Startwert für die Werte setzen, die der Datenbankserver vergibt. Der nächste Wert, den der Datenbankserver erzeugt, ist um eins größer als der größte Wert der Spalte.

Geben Sie für Spalten, die Geldbeträge enthalten, kein Währungssymbol an. Geben Sie nur den numerischen Teil des Betrags an.

Der Datenbankserver kann zwischen numerischen und CHAR-Datentypen konvertieren. Sie können einer numerischen Spalte als Wert eine numerische Zeichenkette (z. B. `'-0075.6'`) angeben. Der Datenbankserver konvertiert die numerische Zeichenkette in eine Zahl. Ein Fehler tritt nur dann auf, wenn die Zeichenkette keine Zahl darstellt.

Für eine CHAR-Spalte können Sie als Wert eine Zahl oder ein Datum angeben. Der Datenbankserver konvertiert den Wert in eine Zeichenkette. Wenn Sie z. B. als Wert für eine CHAR-Spalte TODAY angeben, dann stellt eine Zeichenkette das heutige Datum dar. (Das verwendete Format wird mit der Umgebungsvariable DBDATE festgelegt.)

### Bestimmte Spaltennamen auflisten

Sie brauchen nicht für jede Spalte einen Wert angeben. Statt dessen können Sie nach dem Tabellennamen die Spaltennamen auflisten und dann nur diesen Spalten Werte zuweisen. Im folgenden Beispiel wird ein neuer Datensatz in die Tabelle **stock** eingefügt:

```
INSERT INTO stock (stock_num, description, unit_price, manu_code)
    VALUES (115, 'tyre pump', 114, 'SHM')
```

Beachten Sie, daß in diesem Beispiel nur für die Spalten **stock_num**, **description**, **unit_price** und **manu_code** Werte angegeben werden. Den übrigen Spalten weist der Datenbankserver folgende Werte zu:

- Er erzeugt für eine nicht aufgelistete SERIAL-Spalte eine fortlaufende Nummer.
- Er versorgt eine Spalte, für die ein Standardwert definiert ist, mit diesem Standardwert.
- Er erzeugt einen NULL-Wert für jede Spalte, die NULL-Werte zuläßt und die über keinen Standardwert verfügt; für eine Spalte, für die *als Standardwert* NULL vorgesehen ist, wird hingegen kein Standardwert eingesetzt.

Das bedeutet, daß Sie all die Spalten auflisten müssen, die über keinen Standardwert verfügen oder die keine NULL-Werte zulassen. Diesen Spalten müssen Sie Werte zuweisen. Sie können diese Spalten jedoch in beliebiger

Reihenfolge angeben - sofern die Werte für diese Spalten in derselben Reihenfolge aufgelistet sind. Weitere Information über das Setzen von Standardwerten erhalten Sie in Kapitel 9 "Das Datenmodell implementieren".

Nach der Ausführung der INSERT-Anweisung befindet sich der folgende neue Datensatz in der Tabelle **stock**:

```
stock_num   manu_code  description  unit_price  unit   unit_descr
115         SHM        tyre pump    114
```

Die Felder **unit** und **unit_descr** sind beide leer; dies zeigt an, daß in beiden Spalten NULL-Werte enthalten sind. Da die Spalte **unit** NULL-Werte zuläßt, kann man nur raten, wieviele Luftpumpen für $114 gekauft wurden. Wenn für diese Spalte als Standardwert "box" definiert worden wäre, dann wäre "box" die Maßeinheit. Wenn Sie Werte in bestimmte Spalten einer Tabelle einfügen, sollten Sie auf jeden Fall darauf achten, welche Daten erforderlich sind, um einen sinnvollen Datensatz zu erhalten.

## Mehrere Datensätze und Ausdrücke

Bei der anderen Art der INSERT-Anweisung wird die VALUES-Klausel durch eine SELECT-Anweisung ersetzt. Diese Funktion ermöglicht es, folgende Daten einzufügen:

- Mehrere Datensätze mit nur einer Anweisung (für jeden Satz, den die SELECT-Anweisung zurückliefert, wird ein Datensatz eingefügt)
- Berechnete Werte (die VALUES-Klausel läßt nur Konstanten zu), da die Auswahlliste Ausdrücke enthalten kann.

Nehmen wir z. B. an, daß für jeden Auftrag, der zwar bereits bezahlt, aber noch nicht geliefert ist, ein Folgeauftrag erforderlich ist. Die folgende INSERT-Anweisung sucht diese Aufträge heraus und fügt dann für jeden Auftrag einen Datensatz in die Tabelle **cust_calls** ein:

```
INSERT INTO cust_calls (customer_num, call_descr)
    SELECT customer_num, order_num FROM orders
        WHERE paid_date IS NOT NULL
        AND ship_date IS NULL
```

Diese SELECT-Anweisung liefert zwei Spaltenwerte zurück. Die Daten dieser Spalten (von jedem ausgewählten Satz) werden in die genannten Spalten der Tabelle **cust_calls** eingefügt. Anschließend wird die Auftragsnummer (von der Spalte **order_num**, eine SERIAL-Spalte) in die Spalte **call_descr** eingefügt;

dies ist eine CHAR-Spalte. Wie Sie bereits wissen, läßt es der Datenbankserver zu, ganzzahlige Werte in eine CHAR-Spalte einzufügen. Der Datenbankserver konvertiert die fortlaufende Zahl automatisch in eine Zeichenkette, die dezimale Zeichen enthält.

### Einschränkungen bei der Auswahl zum Einfügen

In der folgenden Übersicht sind die Einschränkungen aufgelistet, die für das Einfügen von Datensätzen mit der SELECT-Anweisung gelten:

- Sie darf keine INTO-Klausel enthalten.
- Sie darf keine INTO TEMP-Klausel enthalten.
- Sie darf keine ORDER BY-Klausel enthalten.
- Sie darf sich auf keine Tabelle beziehen, in die Sie Datensätze einfügen.

Die Einschränkungen, die die Klauseln INTO, INTO TEMP und ORDER BY betreffen, sind geringfügig. Die INTO-Klausel ist in diesem Zusammenhang nicht sinnvoll (dies wird in Kapitel 5 "SQL in Programmen" erläutert). Die Einschränkung der INTO TEMP-Klausel können Sie umgehen, indem Sie zuerst die Daten auswählen, die Sie in eine temporäre Tabelle schreiben wollen und dann erst die INSERT-Anweisung angeben, um die Daten der temporären Tabelle einzufügen. Das Fehlen der ORDER BY-Klausel ist ebenso unwichtig. Wenn Sie sicherstellen müssen, daß die neuen Datensätze in der Tabelle physikalisch sortiert sind, dann können Sie diese zuerst in eine temporäre Tabelle schreiben und sortieren, und diese aus dieser temporären Tabelle einfügen. Sie können eine physikalische Sortierung der Tabelle aber auch erreichen, indem Sie nach den Einfügungen einen Cluster-Index verwenden.

Die letzte Einschränkung ist schwerwiegender, da sie Ihnen verbietet, die *gleiche* Tabelle sowohl in der INTO-Klausel der INSERT-Anweisung als auch in der FROM-Klausel der SELECT-Anweisung anzugeben. (Dies bewahrt den Datenbankserver vor einer Endlos-Schleife, in der jeder eingefügte Satz wieder ausgewählt wird und wieder eingefügt wird.) In einigen Fällen jedoch kann das Verhalten erwünscht sein. Nehmen wir z. B. an, Sie haben herausgefunden, daß die Firma Nikolus dieselben Produkte wie die Firma Anza liefert - dies aber zum halben Preis. Sie wollen der Tabelle **stock** Datensätze hinzufügen, um dies zu verdeutlichen. Am besten ist es, wenn Sie alle Datensätze der Firma Anza aus der Tabelle **stock** auswählen und diese mit dem Hersteller-Code der Firma Nikolus wieder einfügen. Sie können jedoch nicht aus der Tabelle auswählen, in die Sie auch einfügen.

Es gibt eine Möglichkeit, diese Einschränkung zu umgehen. Schreiben Sie die Daten, die Sie einfügen wollen, in eine temporäre Tabelle und wählen Sie dann in der INSERT-Anweisung aus dieser Tabelle aus. Um dies zu erreichen, sind die folgenden Anweisungen erforderlich:

```
SELECT stock_num, 'HSK' temp_manu, description, unit_price/2
        half_price, unit, unit_descr FROM stock
    WHERE manu_code = 'ANZ'
        AND stock_num < 110
    INTO TEMP anzrows

INSERT INTO stock SELECT * FROM anzrows;

DROP TABLE anzrows;
```

Diese SELECT-Anweisung nimmt vorhandene Datensätze aus der Tabelle **stock**, ersetzt den Hersteller-Code durch einen literalen Wert und berechnet den Wert des Preises pro Einheit. Die Datensätze werden dann in der temporären Tabelle **anzrows** gespeichert; die Datensätze dieser Tabelle werden anschließend in die Tabelle **stock** eingefügt.

Wenn Sie mehrere Datensätze einfügen, besteht die Gefahr, daß einer der Datensätze ungültige Daten enthält. Dies führt dazu, daß der Datenbankserver eine Fehlermeldung ausgibt und die Anweisung abgebrochen wird. Selbst wenn kein Fehler auftritt, besteht immer noch die Gefahr (wenn auch sehr gering), daß während der Ausführung der Anweisung ein Hard- oder Softwareausfall auftreten könnte (z. B. könnte die Platte voll sein).

In beiden Fällen ist es nicht einfach zu sagen, wieviele Datensätze eingefügt wurden. Wenn Sie die gesamte Anweisung wiederholen, erstellen Sie vielleicht doppelte Datensätze - vielleicht aber auch nicht. Da sich die Datenbank in einem unbekannten Zustand befindet, können Sie schlecht sagen, was zu tun ist. Die Antwort hierauf ist die Verwendung von Transaktionen; dies wird im Abschnitt "Die Aktualisierung wird unterbrochen" auf Seite 4-24 erläutert.

## Datensätze aktualisieren

Mit der UPDATE-Anweisung aktualisieren Sie den Inhalt einer oder mehrerer Spalten in einem oder mehreren Datensätzen einer Tabelle. Man unterscheidet zwei grundlegend verschiedene Formen dieser Anweisung. In der einen Form können Sie Spalten über ihren Namen bestimmte Werte zuweisen; in der anderen Form können Sie eine Reihe von Werten (die etwa von einer SELECT-Anweisung zurückgegeben wurden) einer Reihe von Spalten zuwei-

sen. Wenn Sie Datensätze aktualisieren und Datenintegritäts-Constraints für einige der Spalten definiert sind, dann müssen die von Ihnen geänderten Daten in jeder der beiden Formen die Bedingungen erfüllen, die durch die Constraints festgelegt sind. Weitere Informationen finden Sie im Abschnitt "Datenbankberechtigungen" auf Seite 4-17.

## Datensätze zum Aktualisieren auswählen

Jede der beiden Formen der UPDATE-Anweisung kann mit einer WHERE-Klausel enden, in der festgelegt ist, welche Datensätze aktualisiert werden. Wenn Sie die Klausel nicht angeben, dann werden alle Datensätze geändert. Die WHERE-Klausel kann ziemlich kompliziert werden, etwa, wenn Sie die genaue Menge der Datensätze bestimmen wollen, die geändert werden müssen. Die einzige bestehende Einschränkung ist, daß die Tabelle, die aktualisiert wird, nicht in der FROM-Klausel einer Unterabfrage angegeben werden darf.

Die erste Form einer UPDATE-Anweisung verwendet eine Folge von Zuweisungsklauseln zum Festlegen von neuen Spaltenwerten. Hier sehen Sie ein Beispiel:

```
UPDATE customer
    SET fname = 'Barnaby', lname = 'Dorfler'
    WHERE customer_num = 103
```

Die WHERE-Klausel wählt die zu ändernden Datensätze aus. Die Spalte **customer.customer_num** in der Datenbank **stores6** ist der Primärschlüssel dieser Tabelle; daher kann diese Anweisung höchstens einen Datensatz aktualisieren.

Sie können in der WHERE-Klausel auch Unterabfragen verwenden. Angenommen, die Firma Anza startet eine Rückrufaktion für ihre Tennisbälle. Aus diesem Grund müssen alle noch nicht versendeten Bestellungen mit der Bestellnummer 6 vom Hersteller ANZ zurückgenommen werden.

```
UPDATE orders
    SET backlog = y'
    WHERE ship_date IS NULL
    AND order_num IN
        (SELECT DISTINCT items.order_num FROM items
            WHERE items.stock_num = 6
            AND items.manu_code = 'ANZ')
```

Die Unterabfrage gibt eine Spalte mit Bestellnummern (keine oder mehrere) zurück. Die UPDATE-Anweisung prüft dann die Übereinstimmung jedes Datensatzes von **orders** und führt gegebenenfalls die Aktualisierung durch.

### Aktualisieren mit Einheitswerten

Jede Zuweisung, die nach dem Schlüsselwort SET steht, legt einen neuen Wert für eine Spalte fest. Dieser Wert wird einheitlich jedem Datensatz zugewiesen, der geändert wird. In den Beispielen des vorherigen Abschnitts wurden Konstanten als neue Werte verwendet; Sie können jedoch beliebige Ausdrücke zuweisen, einschließlich solcher, die auf dem alten Wert der Spalte basieren. Angenommen, der Hersteller mit der Bezeichnung HRO hat alle Preise um 5% angehoben und Sie müssen die Tabelle **stock** nun entsprechend aktualisieren.

```
UPDATE stock
    SET unit_price = unit_price * 1.05
    WHERE manu_code = 'HRO'
```

Sie können auch eine Unterabfrage als Teil des zugewiesenen Wertes verwenden. Wenn eine Unterabfrage Teil eines Ausdrucks ist, dann muß sie genau einen Wert zurückgeben (eine Spalte und einen Datensatz). Angenommen, Sie wollen für jede Bestandsnummer einen Preis festlegen, der über dem aller anderen Hersteller liegt. Sie müssen die Preise aller noch nicht versendeten Bestellungen aktualisieren.

```
UPDATE items
    SET total_price = quantity *
        (SELECT MAX (unit_price) FROM stock
            WHERE stock.stock_num = items.stock_num)
    WHERE items.order_num IN
        (SELECT order_num FROM orders
            WHERE ship_date IS NULL)
```

Die erste SELECT-Anweisung gibt einen einzelnen Wert zurück - den höchsten Preis für ein bestimmtes Produkt der Tabelle **stock**. Dies ist eine korrelierte Unterabfrage; sie muß für jeden zu aktualisierenden Datensatz ausgeführt werden, da ein Wert aus **items** in dessen WHERE-Klausel angegeben ist.

Die zweite SELECT-Anweisung liefert eine Liste mit den Nummern der noch nicht versendeten Aufträge. Es ist eine nicht korrelierte Unterabfrage; sie wird nur einmal ausgeführt.

## Aktualisierungen, die nicht durchgeführt werden können

Bei der Änderung von Daten bestehen Einschränkungen für Unterabfragen. Es ist insbesondere *nicht möglich*, eine Tabelle abzufragen, die gerade geändert wird. Sie *können* sich in einem Ausdruck auf den momentanen Wert einer Spalte beziehen, wie in dem Beispiel, in welchem die Spalte **unit_price** um 5% erhöht wurde. Sie *können* sich in der WHERE-Klausel einer Unterabfrage auf den Wert einer Spalte beziehen, wie in dem Beispiel, in welchem die Tabelle **stock** aktualisiert wurde; in diesem Beispiel wird die Tabelle **items** aktualisiert und die Spalte **items.stock_num** wird in einem Join-Ausdruck verwendet.

In einer gut entworfenen Datenbank besteht selten die Notwendigkeit, eine Tabelle zu aktualisieren und gleichzeitig abzufragen (das Datenbank-Design wird in Kapitel 8 bis Kapitel 11 dieses Handbuchs behandelt). Sie möchten diese Möglichkeit jedoch vielleicht dann verwenden, wenn Sie eine Datenbank entwickeln, bevor ein wohldurchdachter Entwurf vorhanden ist. Ein typisches Problem tritt dann auf, wenn eine Tabelle versehentlich einige Datensätze mit doppelten Werten in einer Spalte enthält, die eindeutig sein soll. Sie möchten vielleicht die doppelten Datensätze löschen oder nur die doppelten Sätze aktualisieren. In jedem Fall erfordert die Überprüfung nach doppelten Datensätzen zwangsläufig eine Unterabfrage, die in einer UPDATE- oder DELETE-Anweisung nicht erlaubt ist. Kapitel 6 "Programme zur Veränderung von Daten" erläutert, wie Sie einen Update-Cursor verwenden können, um diese Art einer Änderung durchzuführen.

## Aktualisieren mit ausgewählten Werten

Die zweite Form der UPDATE-Anweisung verwendet anstelle der Liste von Zuweisungen eine einzige kombinierte Zuweisung, in der eine Reihe von Spalten einer Reihe Liste von Werten gleichgesetzt wird. Bestehen die Werte aus einfachen Konstanten, dann unterscheidet sich diese Form von dem vorherigen Beispiel nur durch eine andere Anordnung der Werte:

```
UPDATE customer
    SET (fname, lname) = ('Barnaby', 'Dorfler')
    WHERE customer_num = 103
```

Es ist nicht vorteilhafter, die Anweisung in dieser Form schreiben. Sie ist sogar schwerer zu lesen, da es nicht offensichtlich ist, welche Werte welchen Spalten zugewiesen werden.

Diese Form ist vor allem dann sinnvoll, wenn die zuzuweisenden Werte von einer einzelnen SELECT-Anweisung geliefert werden. Angenommen, diese Adreßänderungen sollen auf mehrere Kunden angewendet werden. Statt jedesmal zu aktualisieren, wenn eine Änderung bekannt wird, werden die neuen Adressen in einer einzigen temporären Tabelle mit dem Namen **newaddr** gesammelt. Diese enthält Spalten für die Kundennummer und die Adressenfelder der Tabelle **customer**. Nun werden alle neuen Adressen in einem einzigen Schritt in die Tabelle **customer** übertragen. (Bei dieserTechnik erfaßt eine Routine Änderungen in einer gesonderten Tabelle und überträgt sie zu einer Zeit geringerer Rechnerauslastung. Es handelt sich dabei um eine der Performance-Techniken, die in Kapitel 10 "Das Modell tunen" behandelt werden.)

```
UPDATE customer
    SET (address1, address2, city, state, zipcode) =
        ((SELECT address1, address2, city, state, zipcode
        FROM newaddr
        WHERE newaddr.customer_num = customer.customer_num))
    WHERE customer_num IN
        (SELECT customer_num FROM newaddr)
```

Beachten Sie: Eine einzelne SELECT-Anweisung liefert die Werte für mehrere Spalten. Wenn Sie dieses Beispiel so ändern, daß jede zu aktualisierende Spalte eine eigene Wertzuweisung erhält, dann müssen Sie fünf SELECT-Anweisungen verwenden, eine für jede dieser Spalten. Eine solche Anweisung ist nicht nur schwerer zu schreiben, sondern wird auch langsamer ausgeführt.

***Hinweis:*** *Bei* **INFORMIX-4GL** *und Programmen mit eingebettetem SQL können Sie Record- oder Host-Variablen zum Aktualisieren von Werten verwenden. Weitere Informationen hierüber finden Sie in Kapitel 5 "SQL in Programmen".*

# Datenbankberechtigungen

Man unterscheidet zwei Ebenen von Berechtigungen in einer Datenbank: Rechte auf Datenbankebene (*Datenbankberechtigungen*) und Rechte auf Tabellenebene (*Tabellenberechtigungen*). Wenn Sie eine Datenbank erzeugen, dann können Sie solange als einzige Person darauf zugreifen, bis Sie (als Eigentümer oder DBA der Datenbank) anderen Personen Berechtigungen erteilen. Wenn Sie eine Tabelle in einer Datenbank erzeugen, die nicht ANSI-kompatibel ist, dann haben alle Benutzer solange Zugriff auf diese Tabelle, bis Sie (als Eigentümer der Tabelle) diese Tabellenberechtigungen bestimmten Benutzern entziehen.

Nachfolgend sind die drei verschiedenen Datenbank-Berechtigungen aufgeführt:

- Mit CONNECT-Berechtigung können Sie eine Datenbank öffnen, Abfragen durchführen und Indizes für temporäre Tabellen erzeugen.
- Mit RESOURCE-Berechtigung können Sie permanente Tabellen erzeugen.
- Mit DBA-Berechtigung können Sie als DBA zusätzliche Funktionen nutzen.

Man unterscheidet sieben Tabellenberechtigungen, von denen hier jedoch nur die ersten vier aufgeführt sind:

- Mit SELECT-Berechtigung können Sie Datensätze aus einer Tabelle auswählen. Diese Berechtigung wird tabellenweise vergeben und kann auf bestimmte Spalten der Tabelle beschränkt werden.
- Mit DELETE-Berechtigung können Sie Datensätze löschen.
- Mit INSERT-Berechtigung können Sie Datensätze einfügen.
- Mit UPDATE-Berechtigung können Sie Datensätze aktualisieren (d. h. ihren Inhalt ändern).

Beim Erzeugen von Datenbanken und Tabellen wird meistens CONNECT- und SELECT-Berechtigung an *public* erteilt, wodurch diese Berechtigungen allen Benutzern zuteil werden. Wenn Sie eine Tabelle abfragen können, dann besitzen Sie zumindest CONNECT-Berechtigung für diese Datenbank und SELECT-Berechtigung für die Tabelle. Weitere Informationen erhalten Sie im Abschnitt "Einzelne Benutzer und Public" in Kapitel 11 "Zugriff auf Datenbanken regeln".

Die anderen Tabellenberechtigungen sind zum Ändern von Daten erforderlich. Die Eigentümer von Tabellen verweigern häufig diese Berechtigung oder gewähren sie nur bestimmten Benutzern. Aus diesem Grund ist es unter Umständen nicht möglich, Tabellen zu ändern, die Sie ohne Einschränkungen abfragen können.

Da die Berechtigungen tabellenweise vergeben werden, kann es sein, daß Sie für eine Tabelle nur INSERT-, für eine andere hingegen nur UPDATE-Berechtigung besitzen. Die UPDATE-Berechtigung kann darüberhinaus auf einzelne Spalten in einer Tabelle beschränkt sein.

In Kapitel 11 "Zugriff auf Datenbanken regeln" werden die verschiedenen Möglichkeiten der Vergabe von Berechtigungen ausführlich beschrieben. Eine vollständige Aufstellung aller Berechtigungen und eine Zusammenfassung der Anweisungen GRANT und REVOKE finden Sie in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

## Tabellen-Berechtigungen anzeigen

Wenn Sie Eigentümer oder Eigentümerin der Tabelle sind (wenn Sie diese erzeugt haben), dann haben Sie alle Berechtigungen für diese Tabelle. Andernfalls können Sie herausfinden, welche Rechte Sie für eine bestimmte Tabelle besitzen, indem Sie das Systemverzeichnis abfragen. Das Systemverzeichnis besteht aus Systemtabellen, welche die Datenbankstruktur beschreiben. Die Berechtigungen für jede Tabelle werden in der Systemtabelle **systabauth** festgehalten. Zum Anzeigen dieser Berechtigungen benötigen Sie die eindeutige Bezeichnernummer der Tabelle. Diese Nummer finden Sie in der Systemtabelle **systables**. Sie können also die folgende SELECT-Anweisung ausführen, wenn Sie sich beispielsweise die Berechtigungen anzeigen lassen wollen , die für die Tabelle **orders** vergeben sind,:

```
SELECT * FROM systabauth
    WHERE tabid = (SELECT tabid FROM systables
                        WHERE tabname = 'orders')
```

Die Ausgabe der Abfrage gleicht der nachfolgend gezeigten:

```
grantor      grantee      tabid      tabauth

tfecit       mutator      101        su-i-x-
tfecit       procrustes   101        s--idx-
tfecit       public       101        s--i-x-
```

Der `grantor` (Erteiler) ist der Benutzer, der die Berechtigung *erteilt* hat. Es handelt sich dabei normalerweise um den Eigentümer der Tabelle; es kann aber auch jeder andere Benutzer sein, der vom Erteiler dazu ermächtigt wurde. Der `grantee` ist der Benutzer, dem die Berechtigung erteilt wurde;

*public* an dieser Stelle bedeutet "jeder Benutzer mit Connect-Berechtigung". Erscheint Ihr Benutzername nicht in der Liste, dann besitzen Sie nur die Rechte von public.

Die Spalte **tabauth** informiert über die erteilten Berechtigungen. Die Buchstaben in jeder Zeile dieser Spalte sind die Anfangsbuchstaben der Namen der Berechtigungen, mit der Ausnahme, daß `i` Insert und `x` Index bedeutet. In diesem Beispiel hat public das Auswahl-, Einfüge- und Indexberechtigung (Select, Insert und x). Nur der Benutzer `mutator` besitzt die Aktualisierungsberechtigung (Update), und nur der Benutzer `procrustes` besitzt das Löschrecht (Delete).

Bevor der Datenbankserver irgend etwas für Sie ausführt (beispielsweise eine DELETE-Anweisung), führt er eine Abfrage durch, die der vorhergehenden ähnlich ist. Falls Sie nicht der Tabelleneigentümer sind und falls der Datenbankserver weder für Ihren Benutzernamen noch für public die notwendigen Berechtigungen vorfindet, dann verweigert er die Ausführung.

# Datenintegrität

Die Anweisungen INSERT, UPDATE und DELETE ändern Daten in einer Datenbank. Immer, wenn sie vorhandene Daten ändern, kann die *Integrität* von Daten beeinträchtigt werden. So könnte z. B. die Bestellung eines nicht vorhandenen Produkts in die Tabelle **orders** eingegeben werden. Oder ein Kunde mit ausstehenden Bestellungen könnte aus der Tabelle **customer** gelöscht werden. Oder die Auftragsnummer könnte in der Tabelle **orders** und *nicht* in der Tabelle **items** geändert werden. In jedem dieser Fälle geht die Integrität der gespeicherten Daten verloren.

Datenintegrität besteht im Grunde aus drei Teilen:

- Objektintegrität (Entity-Integrität)

  Jeder Datensatz der Tabelle hat eindeutige Bezeichner.

- Semantische Integrität

  Die Daten in den Spalten geben die Informationen, welche die Spalte aufnehmen soll, richtig wieder.

- Referentielle Integrität

  Die Beziehungen zwischen Tabellen werden überwacht.

Gute Datenbank-Entwürfe beachten diese Prinzipien, so daß die Datenbank selbst alle Möglichkeiten verhindert, wodurch die Integrität der Daten verletzt werden könnte.

## Objektintegrität

Ein Objekt ist jede Person, jeder Ort oder jede Sache, die (der) in eine Datenbank aufgenommen werden soll. Jedes Objekt steht für eine Tabelle, und jeder Datensatz einer Tabelle steht für ein Beispiel dieses Objekts. Ist beispielsweise ein Auftrag ein Objekt, dann repräsentiert die Tabelle **orders** die Menge der Aufträge und *jeder* Datensatz der Tabelle repräsentiert einen bestimmten Auftrag.

Um jeden Datensatz in der Tabelle ansprechen zu können, muß ein Primärschlüssel vorhanden sein. Der Primärschlüssel ist ein eindeutiger Wert, der jeden Datensatz identifiziert. Diese Anforderung nennt man *Objektintegrität-Constraint.*

Der Primärschlüssel der Tabelle **orders** z.B ist die Spalte **order_num**. Die Spalte **order_num** enthält für jeden Datensatz in der Tabelle eine vom System erzeugte, eindeutige Auftragsnummer. Wenn Sie auf einen Datensatz der Tabelle **orders** zugreifen wollen, dann können Sie die folgende SELECT-Anweisung verwenden:

```
SELECT * FROM orders WHERE order_num = 1001
```

Indem Sie die Auftragsnummer in der WHERE-Klausel angeben, können Sie auf einfache Weise auf einen Datensatz zugreifen, da die Auftragsnummer über alle Datensätze hinweg eindeutig ist. Wenn die Tabelle doppelte Auftragsnummern zuließe, dann wäre es nahezu unmöglich, auf einen einzigen Datensatz zuzugreifen, weil alle anderen Spalten dieser Tabelle doppelte Werte zulassen.

Weitere Informationen über Primärschlüssel und Objektintegrität finden Sie in Kapitel 8 "Aufbau eines Datenmodells".

## Semantische Integrität

Semantische Integrität gewährleistet, daß der in einen Datensatz aufgenommene Wert für diesen Satz zulässig ist. Dies bedeutet, daß der Wert im *Wertebereich* liegen muß. Der Wertebereich ist die Menge der in dieser Spalte zulässigen Werte. So läßt z. B. die Spalte **quantity** der Tabelle **items** nur Zahlen zu. Wird ein Wert, der außerhalb dieses Wertebereichs liegt, in die Spalte aufgenommen, dann wird die semantische Integrität der Daten verletzt.

Semantische Integrität wird durch folgende Constraints erreicht:

- Datentyp

  Der Datentyp legt die Art der Werte fest, die in die Spalte aufgenommen werden können. Der Datentyp SMALLINT läßt z. B. in einer Spalte Werte aus dem Bereich von -32767 bis 32767 zu.

- Standardwert

  Der Standardwert wird in eine Spalte aufgenommen, wenn kein expliziter Wert angegeben ist. Beispielsweise wird der Spalte **user_id** der Tabelle **cust_calls** die Kennung des Benutzers zugewiesen, wenn kein Name angegeben ist.

- Prüf-Constraints

  Ein Prüf-Constraint legt Bedingungen für Daten fest, die in eine Spalte eingefügt werden. Jeder in die Tabelle eingefügte Datensatz muß diesen Bedingungen genügen. Beispielsweise könnte die Spalte **quantity** der Tabelle **items** prüfen, ob die eingegebenen Mengen größer oder gleich 1 sind.

  Weitere Informationen über die Verwendung von Constraints zur Wahrung der semantischen Integrität im Datenbank-Design finden Sie unter "Bestimmung der Wertebereiche" auf Seite 9-3.

# Referentielle Integrität

Referentielle Integrität betrifft die Beziehung *zwischen* Tabellen. Da jede Tabelle in einer Datenbank einen Primärschlüssel besitzen muß, kann dieser Primärschlüssel auch in anderen Tabellen erscheinen, um die Beziehungen der Daten dieser Tabellen anzuzeigen. Erscheint der Primärschlüssel einer Tabelle in einer anderen Tabelle, dann nennt man ihn einen Fremdschlüssel.

Fremdschlüssel *verknüpfen* Tabellen und stellen Abhängigkeiten zwischen ihnen her. Tabellen können eine Hierarchie von Abhängigkeiten dergestalt aufbauen, daß die Bedeutung von Datensätzen in Tabellen zerstört werden kann, sobald in einer anderen Tabelle ein Datensatz geändert oder gelöscht wird. Beispielsweise ist in der Datenbank **stores6** die Spalte **customer_num** der Tabelle **customer** der Primärschlüssel für die Tabelle **customer**. Außerdem ist die Spalte ein Fremdschlüssel in den Tabellen **orders** und **cust_calls**. Der Kunde mit der Nummer 106, George Watson, wird sowohl in der Tabelle **orders** als auch in der Tabelle **cust_calls** *angesprochen*. Wird der Kunde 106 aus der Tabelle **customer** gelöscht, dann ist die Verbindung zwischen den drei Tabellen und diesem Kunden zerstört.

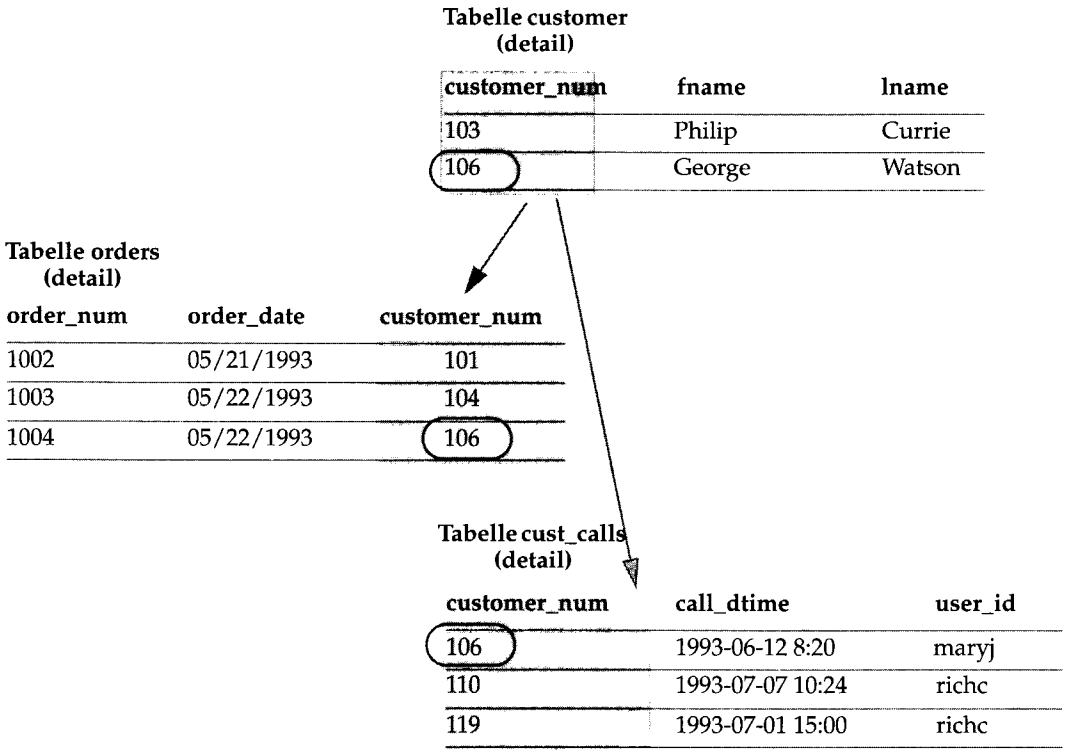



*Bild 4-1* ***Referentielle Integrität der Beispiel-Datenbank stores6***

Wenn Sie einen Datensatz löschen, der einen Primärschlüssel enthält, oder ihn mit einem anderen Primärschlüssel aktualisieren, dann zerstören Sie die Bedeutung aller Datensätze, welche diesen Wert als Fremdschlüssel enthalten. Referentielle Integrität ist die logische Abhängigkeit eines Fremdschlüssels von einem Primärschlüssel. Die *Integrität* eines Datensatzes, der einen Fremdschlüssel enthält, hängt von der Integrität des Datensatzes ab, auf den der Fremdschlüssel *verweist* (referenziert) - der Datensatz, der den entsprechenden Primärschlüssel enthält.

Normalerweise können Sie bei **INFORMIX-OnLine Dynamic Server** die referentielle Integrität nicht verletzen. Sie erhalten stets eine Fehlermeldung, wenn Sie versuchen, Sätze aus einer Elterntabelle zu löschen, bevor Sie die dazugehörigen Kindtabellen gelöscht haben. Mit der Option ON DELETE CASCADE können Sie allerdings erreichen, daß mit dem Löschen der Sätze in der Elterntabelle automatisch die entsprechenden Sätze in den Kind-Tabellen gelöscht werden. Weiter Information dazu erhalten Sie im folgenden Abschnitt "Arbeiten mit der Option ON DELETE CASCADE".

## Arbeiten mit der Option ON DELETE CASCADE

Sie können beim Löschen von Datensätzen die referentielle Integrität aufrecht erhalten, wenn Sie in den CREATE TABLE- bzw. ALTER TABLE-Anweisungen der referenzierten Tabellen die Option ON DELETE CASCADE der Klausel REFERENCING angeben. Sie sind dann in der Lage, mit einem einzigen Tastendruck Sätze aus einer Eltern-Tabelle und die dazugehörigen Sätze aus den Kind-Tabellen zu löschen.

### Sperren beim kaskadischen Löschen

Während des Löschens sind alle betroffenen Sätze der Eltern- und der Kind-Tabellen gesperrt. Wenn Sie den Lösch-Vorgang starten, werden zunächst die Sätze in den Eltern-Tabellen und erst dann die Sätze in den referenzierten Tabellen gelöscht.

### Löschen mit mehreren Kind-Tabellen

Falls Sie eine Eltern-Tabelle mit zwei Kind-Tabellen haben, von denen lediglich eine das kaskadische Löschen erlaubt, erhalten Sie eine Fehlermeldung beim Versuch, in beiden Tabellen Datensätze zu löschen. In diesem Fall wird weder in den Kind- noch in den Elterntabellen gelöscht.

### Die Protokollierung muß aktiviert sein

Um mit kaskadischem Löschen zu arbeiten, muß die Protokollierung aktiviert sein. Weitere Information über Protokollierung und kaskadisches Löschen erhalten Sie im Abschnitt "Das Transaktionsprotokoll" auf Seite 4-26.

### Beispiel:

Sie haben zwei Tabellen, zwischen denen referentielle Integrität bestehen soll. Es handelt sich um die Eltern-Tabelle **accounts** und um die Kind-Tabelle **sub_accounts**. Mit den folgenden CREATE TABLE-Anweisung wird die referentielle Integrität gewährleistet:

```
CREATE table accounts (
acc_num SERIAL primary key,
acc_type INT,
acc_descr CHAR(20));

CREATE table sub_accounts (
sub_acc INTEGER primary key,
ref_num INTEGER references accounts (acc_num) ON DELETE CASCADE,
sub_descr CHAR(20));
```

Der Primärschlüssel der Tabelle **accounts**, das Feld **acc_num**, hat den Datentyp SERIAL. In der Kind-Tabelle **sub_accounts** hat der dazugehörige Fremdschlüssel, das Feld **ref_num**, den Datentyp INTEGER. Die Kombination der Datentypen SERIAL im Primärschlüssel und INTEGER im Fremdschlüssel ist zulässig. Dies ist aber auch der einzige Fall, bei dem Sie in Primär- und Fremdschlüssel verschiedene Datentypen mischen können. Der Datentyp SERIAL ist nämlich eigentlich INTEGER, außer, daß bei SERIAL die Werte, die in das Feld eingesetzt werden, automatisch von der Datenbank erzeugt werden. Bei allen anderen Datentypen müssen Primär- und Fremdschlüssel explizit vom gleichen Datentyp sein. Beispielsweise kann ein Primärschlüssel vom Typ CHAR nur mit einem Fremdschlüssel kombiniert werden, der auch vom Typ CHAR ist.

Um einen Datensatz aus der Tabelle **accounts** zu löschen, der einen referenzierten Eintrag in der Kind-Tabelle **sub_accounts** hat, müssen Sie die Protokollierung einschalten. Anschließend können Sie den Satz mit der Account Nummer 2 mit folgender Anweisung löschen:

```
DELETE from accounts where acc_num = 2
```

### Beschränkungen beim kaskadischen Löschen

Kaskadisches Löschen funktioniert fast immer, unter anderem auch bei Abfragen, die sich auf sich selbst beziehen, sowie bei zyklischen Abfragen . Die einzige Ausnahme sind korrelierte Unterabfragen. Bei dieser Art von Abfragen hängt das Ergebnis der Unterabfrage (auch als *innerer SELECT* bezeichnet) vom Ergebnis der Hauptabfrage (*outer SELECT*) ab. Kaskadisches Löschen darf nicht in einer Kind-Tabelle aktiviert sein , die in einer korrelierten Unterabfrage verwendet wird. In diesem Fall erhalten Sie eine Fehlermeldung und das Löschen funktioniert nicht.

## Die Aktualisierung wird unterbrochen

Auch wenn die gesamte Software fehlerfrei und sämtliche Hardware vollkommen zuverlässig ist, ist es dennoch nicht ausgeschlossen, daß die Umgebung des Rechners Störungen verursachen kann. Es könnte etwa ein Blitz in das Gebäude einschlagen, die elektrische Versorgung lahmlegen und den Rechner während der Ausführung einer UPDATE-Anweisung unterbrechen. Wahrscheinlicher ist, daß der Plattenspeicher voll wird oder ein Benutzer fehlerhafte Daten eingibt, wodurch ein Einfügevorgang, der sich über mehrere Datensätze erstreckt, vorzeitig mit einer Fehlermeldung abgebrochen

wird. In jedem Fall müssen Sie damit rechnen, daß während einer Datenänderung ein unvorhersehbares Ereignis den Änderungsvorgang unterbrechen kann.

Wird eine Änderung aufgrund einer äußeren Störung unterbrochen, dann läßt sich nicht mit Sicherheit sagen, wie weit die Änderungen durchgeführt wurden. Selbst beim Einfügen eines einzigen Datensatzes können Sie nicht wissen, ob die Daten im Plattenspeicher angekommen sind und ob die Indizes richtig aktualisiert wurden.

Änderungen, die sich über mehrere Datensätze erstrecken, stellen bereits ein Problem dar. Aber Änderungen, die mehrere Anweisungen umfassen, sind noch problematischer. Dabei hilft es nicht, daß sie normalerweise in Programme eingebettet sind, wodurch Sie die einzelnen SQL-Anweisungen, die gerade ausgeführt werden, nicht sehen können. Es sind z. B. die folgenden Schritte nötig, um einen neuen Auftrag in die Datenbank **stores6** einzutragen:

- Einen Satz in die Tabelle **orders** einfügen. (Dabei wird eine Auftragsnummer erzeugt.)
- Für jeden bestellten Artikel einen Satz in die Tabelle **items** eintragen.

Es gibt zwei Möglichkeiten, eine Anwendung für die Auftragserfassung zu programmieren. Zum einen können Sie sie vollkommen interaktiv gestalten, so daß der erste Datensatz sofort eingefügt wird und jeder Artikel-Datensatz aufgenommen wird, sobald die Daten eingegeben werden. Dies ist jedoch der falsche Ansatz, da in diesem Fall zahlreiche weitere Ereignisse auftreten können, die Fehler verursachen: das Telefongespräch mit dem Kunden könnte unterbrochen werden, der Operator könnte die falsche Taste drücken, die Stromversorgung des Terminals könnte unterbrochen werden, usw.

Hier ist der richtige Weg, um eine Anwendung für die Auftragserfassung zu programmieren:

- Nehmen Sie alle Daten interaktiv auf.
- Überprüfen Sie die Daten und vervollständigen Sie sie (indem Sie z. B. die Bedeutung von Codes der Tabellen **stock** und **manufact** nachschlagen).
- Lassen Sie sich die Informationen zur Kontrolle auf dem Bildschirm anzeigen.
- Warten Sie auf die abschließende Bestätigung des Operators.
- Fügen Sie die Daten so schnell wie möglich ein.

Selbst in diesem Fall kann nach der Auftragserfassung aber noch vor der Aufnahme sämtlicher Artikel ein unvorhersehbarer Umstand das Programm unterbrechen. Wenn dies geschieht, dann befindet sich die Datenbank in einem Zustand, der nicht bekannt ist: die *Datenintegrität* ist gefährdet.

## Die Transaktion

Die Lösung all dieser Probleme ist die *Transaktion*. Eine Transaktion ist eine Folge von Änderungen, die entweder vollständig oder gar nicht durchgeführt werden. Der Datenbankserver gewährleistet, daß entweder Operationen, die im Rahmen einer Transaktion ausgeführt werden, vollständig im Speicher abgeschlossen werden oder die Datenbank in den Zustand versetzt wird, in dem sie sich zu Beginn der Transaktion befunden hat.

Eine Transaktion ist nicht nur ein Schutz gegen unvorhergesehene Ereignisse, sondern sie stellt einem Programm auch die Möglichkeit zur Verfügung, abzubrechen, sobald ein logischer Fehler entdeckt wird. (Mehr dazu finden Sie in Kapitel 6 dieses Handbuchs.)

## Das Transaktionsprotokoll

Der Datenbankserver kann jede Änderung protokollieren, die während einer Transaktion an der Datenbank vorgenommen wird. Wird durch irgendein Ereignis eine Transaktion abgebrochen, dann stellt der Datenbankserver anhand der protokollierten Daten automatisch den ursprünglichen Zustand der Datenbank wieder her.

Eine Transaktion kann aus verschiedenen Gründen scheitern. Das Anwendungsprogramm, das die SQL-Anweisung geschickt hat, kann abgestürzt oder beendet worden sein oder es ist ein Hard- oder Softwarefehler aufgetreten. Sobald der Datenbankserver bemerkt, daß eine Transaktion nicht funktioniert hat (dies kann durchaus erst nach einem Neustart des Systems sein), versetzt er die Datenbank wieder in den Zustand, in dem sie sich vor dem Beginn der Transaktion befand. Dazu verwendet er das Transaktionsprotokoll.

Den Vorgang des Aufnehmens aller Änderungen in das Protokoll nennt man *Transaktionsprotokollierung*. Das Protokoll wird üblicherweise in einer von der Datenbank gesonderten Datei auf der Platte gehalten. Bei **INFORMIX-OnLine Dynamic Server** werden die Protokolle in sogenannten logischen Protokollen verwaltet (weil die Protokolle logische Einheiten der Transkation darstellen). Bei **INFORMIX-SE** spricht man von Transaktionsprotokoll-Dateien.

Datenbanken führen ein Transaktionsprotokoll nicht automatisch. Der Datenbankadministrator muß entscheiden, ob für eine Datenbank ein Transaktionsprotokoll geführt werden soll oder nicht.

### Transaktionsprotokolle und kaskadisches Löschen

Die Transaktionsprotokollierung muß eingeschaltet sein, wenn Sie mit kaskadischem Löschen arbeiten wollen. Dies ist so, weil zuerst der Primärschlüssel der Eltern-Tabelle gelöscht wird. Falls es nun zu einem Systemabsturz käme, bevor auch die entsprechenden Sätze in den Kind-Tabellen gelöscht worden wären, wäre die referentielle Integrität verletzt. Selbst bei kurzfristiger Deaktivierung des Transaktionsprotokolles wird das kaskadische Löschen nicht funktionieren. Es ist erst dann wieder möglich, wenn die Protokollierung wieder eingeschaltet wird. Bei **INFORMIX-OnLine** Datenbankservern wird die Transaktionsprotokollierung mit CREATE DATABASE aktiviert.

## Transaktionen festlegen

Die Grenzen von Transaktionen werden mit SQL-Anweisungen festgelegt. Dies kann auf zwei Arten geschehen. Die überwiegend verwendete Art legt den Beginn einer aus mehreren Anweisungen bestehenden Transaktion mit der Anweisung BEGIN WORK fest. In Datenbanken, die mit der Option MODE ANSI erzeugt wurden, kann man darauf verzichten, den Beginn einer Transaktion zu kennzeichnen. Transaktionen werden bei diesen Datenbanken immer verwendet; Sie müssen lediglich das Ende jeder Transaktion anzeigen.

Bei beiden Arten wird das Ende einer erfolgreichen Transaktion mit der Anweisung COMMIT WORK gekennzeichnet. Diese Anweisung teilt dem Datenbankserver mit, daß das Ende einer Reihe von Anweisungen erreicht ist, die alle zusammen erfolgreich ausgeführt werden müssen. Der Datenbankserver führt alle Handlungen aus, die dazu notwendig sind, daß alle Änderungen richtig abgeschlossen und auf den Plattenspeicher übertragen werden.

Ein Programm kann eine Transaktion absichtlich abzubrechen, indem es die Anweisung ROLLBACK WORK ausführt. Diese Anweisung veranlaßt den Datenbankserver, die aktuelle Transaktion abzubrechen und alle Änderungen rückgängig zu machen.

Sie können z. B. in einer Anwendung, die neue Aufträge aufnimmt, Transaktionen in folgender Weise verwenden:

- Alle Daten interaktiv aufnehmen.
- Die Daten überprüfen und vervollständigen.

- Auf die abschließende Bestätigung des Operators warten.
- Die Anweisung BEGIN WORK ausführen.
- Datensätze in die Tabellen **orders** und **items** einfügen und dabei die vom Datenbankserver zurückgegebenen Fehlercodes prüfen.
- Die Anweisung COMMIT WORK ausführen, wenn keine Fehler aufgetreten sind; andernfalls die Anweisung ROLLBACK WORK.

Wenn die Transaktion aufgrund irgendwelcher äußeren Ereignisse nicht vollständig durchgeführt werden kann, dann wird alles, was bisher ausgeführt wurde, bei einem neuen Systemstart rückgängig gemacht. Auf alle Fälle befindet sich der Datenbankserver in einem vorhersehbaren Zustand: der neue Auftrag ist entweder vollständig oder überhaupt nicht aufgenommen worden.

# Archivieren und Protokollieren

Durch Verwendung von Transaktionen erhalten Sie die Sicherheit, daß sich die Datenbank stets in einem konsistenten Zustand befindet und Ihre Änderungen fehlerfrei in den Plattenspeicher übertragen wurden. Allerdings ist der Plattenspeicher selbst nicht vollkommen sicher. Er ist mechanisch anfällig und kann durch Überschwemmungen, Feuer oder Erdbeben zerstört werden. Die einzige Möglichkeit, sich dagegen zu sichern, ist das Erstellen mehrerer Kopien der Daten. Diese Kopien werden als *Sicherungskopien* bezeichnet.

Das Transaktionsprotokoll (auch logisches Protokoll genannt) ergänzt die Sicherungskopie einer Datenbank. Es enthält eine chronologische Aufzeichnung aller Änderungen, die seit der letzten Sicherung der Datenbank aufgetreten sind. Sollte es jemals nötig sein, die Datenbank aus einer Sicherungskopie wiederherzustellen, dann kann das Transaktionsprotokoll dazu verwendet werden, die Datenbank wieder in ihren aktuellen Zustand zu versetzen.

# Archivieren mit INFORMIX-SE

Ist eine Datenbank in Betriebssystemdateien (**INFORMIX-SE**) gespeichert, dann werden Sicherungskopien unter Verwendung der üblichen, vom Betriebssystem angebotenen Methoden zum Archivieren erstellt. Es gibt allerdings bei Datenbanken zwei Dinge zu überlegen.

Die erste ist eine praktische Überlegung: Eine Datenbank kann sehr groß werden. Es könnte sogar die größte Datei oder Anzahl von Dateien im System werden. Es kann sehr zeitaufwendig sein, eine Kopie anzufertigen. Vielleicht benötigen Sie eine besondere Prozedur zum Kopieren der Datenbank, die nichts mit der normalen Archivierungsmethode zu tun hat.

Die zweite Überlegung ist die besondere Beziehung zwischen der Datenbank und der Transaktionsprotokoll-Datei. Eine Sicherungskopie enthält den Zustand der Datenbank zu einem bestimmten Zeitpunkt. Die Protokolldatei enthält die chronologische Aufzeichnung aller Änderungen, die seit einem bestimmten Zeitpunkt an der Datenbank vorgenommen wurden. Es ist wichtig, daß diese beiden Zeitpunkte gleich sind. Anders ausgedrückt, es ist wichtig, daß eine neue Transaktionsprotokoll-Datei unmittelbar nach der Erstellung der letzten Sicherungskopie der Datenbank begonnen wird. Wenn Sie später die Datenbank aus der Sicherungskopie wiederherstellen müssen, dann erhält das Transaktionsprotokoll eine genaue chronologische Aufzeichnung der Änderungen, so daß die Datenbank wieder in den Zustand ihrer jüngsten Änderung gebracht werden kann.

Die Anweisung, die ein Protokoll auf eine wiederhergestellte Datenbank anwendet, heißt ROLLFORWARD DATABASE. Sie erstellen eine neue Transaktionsprotokoll-Datei, indem Sie mit Betriebssystemkommandos die Datei löschen und eine neue, leere Datei erzeugen oder einfach die Dateilänge auf Null setzen.

Eine Transaktionsprotokoll-Datei kann extrem groß werden. Wenn Sie einen Datensatz zehnmal aktualisieren, dann enthält die Datenbank immer noch nur diesen einen Satz - aber im Protokoll sind zehn Aktualisierungen aufgezeichnet. Wird die Größe der Protokolldatei problematisch, dann können Sie ein neues Protokoll erstellen. Wählen Sie einen Zeitpunkt, zu dem die Datenbank nicht aktualisiert wird (also keine Transaktionen durchgeführt werden), und kopieren Sie dann die bestehende Protokolldatei auf einen anderen Datenträger. Diese Kopie enthält alle Änderungen für einen bestimmten Zeitabschnitt; bewahren Sie sie sorgfältig auf. Erstellen Sie nun eine neue Protokolldatei. Sollten Sie irgendwann die Datenbank aus einer Sicherungskopie wiederherstellen müssen, dann müssen Sie alle Protokolldateien in der richtigen Reihenfolge verwenden.

# INFORMIX-OnLine Dynamic Server Datenbanken archivieren

Der Datenbankserver **INFORMIX-OnLine Dynamic Server** stellt eine große Anzahl Funktionen zur Archivierung und Protokollierung zur Verfügung. Diese Funktionen sind ausführlich im Handbuch *INFORMIX-OnLine Archive and Backup Guide* beschrieben.

Die Eigenschaften von **INFORMIX-OnLine Dynamic Server** gleichen den gerade für **INFORMIX-SE** beschriebenen, sind aber aus folgenden Gründen umfangreicher:

- **INFORMIX-OnLine Dynamic Server** stellt strenge Anforderungen an Performance und Zuverlässigkeit (z. B. können Sicherungskopien angefertigt werden, während mit der Datenbank gearbeitet wird).
- Der Plattenspeicherplatz für die logische Protokollierung wird vom Datenbankserver selbst verwaltet.
- Protokollierungen werden für alle Datenbanken gleichzeitig ausgeführt; dafür wird eine begrenzte Anzahl von Protokolldateien verwendet. Diese Dateien können auf eine anderes Speichermedium (z. B. als Sicherheitskopie) kopiert werden, während die Transaktion läuft.

Diese Optionen werden normalerweise von einer zentralen Stelle aus vom **INFORMIX-OnLine Dynamic Server**-Systemverwalter verwaltet, so daß sich die Benutzer der Datenbanken darum nicht selbst kümmern müssen.

Wenn Sie für sich selbst eine Sicherungskopie einer einzelnen Datenbank oder Tabelle anlegen wollen, dann können Sie dazu das Dienstprogramm **onunload** verwenden. Dieses Programm kopiert eine Tabelle oder eine Datenbank auf Band. Das Programm speichert binäre Abbildungen der Platten-Pages, wie sie von **INFORMIX-OnLine Dynamic Server** angelegt wurden. Aus diesem Grund kann das Kopieren sehr schnell durchgeführt werden und das entsprechende Dienstprogramm **onload** kann die Datei schnell wiederherstellen. Das Datenformat kann allerdings von keinem anderen Programm gelesen werden.

Falls Ihr Verwalter von **INFORMIX-OnLine** ON-Archive verwendet, um Archive und logische Protokolle zu erzeugen, können Sie mit ON-Archive auch Ihre eigenen Sicherheitskopien erzeugen. Weitere Informationen zu diesem Thema erhalten Sie im Handbuch *INFORMIX-OnLine Archive and Backup Guide.*

# Parallelbearbeitung und Sperren

Wenn sich Ihre Datenbank auf einem Einplatz-Rechner befindet, der über kein Netzwerk mit anderen Rechnern verbunden ist, dann betrifft Sie das Thema Parallelbearbeitung nicht. In allen anderen Fällen müssen Sie dagegen die Möglichkeit in Betracht ziehen, daß ein anderes Programm genau diejenigen Daten liest oder ändert, die Ihr Programm gerade bearbeitet. Dies ist eine *Parallelbearbeitung*: zwei oder mehrere voneinander unabhängige Zugriffe auf die gleichen Daten zum gleichen Zeitpunkt.

Ein hoher Grad an Parallelbearbeitung ist entscheidend für gute Performance in einem Datenbanksystem mit mehreren Benutzern. Parallelbearbeitung kann allerdings eine Vielzahl von negativen Auswirkungen mit sich bringen, sofern keine Mechanismen vorhanden sind, die die Verwendung der Daten einschränken. Programme könnten irrtümlicherweise Daten lesen, die nicht mehr auf dem neuesten Stand sind; Änderungen könnten verlorengehen, obwohl sie scheinbar erfolgreich durchgeführt wurden.

Der Datenbankserver verhindert solche Fehler durch die Verwendung eines *Sperrmechanismus*. Eine Sperre ist eine Art Reservierung, mit der ein Programm einen Bereich von Daten belegt. Fordert ein anderer Benutzer die Daten an, dann veranlaßt der Datenbankserver das Programm zum Warten oder liefert eine Fehlermeldung zurück.

Sie können die Auswirkungen von Sperren auf den Datenzugriff mit zwei Anweisungen regeln: SET LOCK MODE und SET ISOLATION. Die Einzelheiten dieser Anweisungen lassen sich am besten verstehen, wenn man die Beschreibung über die Verwendung von *Cursors* in Programmen gelesen hat. Sie finden sie in den Kapiteln 5 und 6 dieses Handbuchs. Zum Thema *Sperren* erhalten Sie weitere Informationen im Kapitel 7.

# Datenreplikation

Von Datenreplikation im weitesten Sinne ist die Rede, wenn Objekte einer Datenbank auf mehr als einem Rechner oder Datenbankserver (*site*) abgelegt sind. Datenreplikation liegt beispielsweise dann vor, wenn eine Datenbank auf einen Datenbankserver kopiert wird, der auf einem anderen Rechner läuft, um dort Listen erzeugen zu können, ohne gleichzeitig die Dateneingabe, die über den anderen Datenbankserer läuft, zu behindern bzw. zu verzögern.

Datenreplikation hat folgende Vorteile:

- Wenn Anwender mit gespiegelten Daten *lokal* arbeiten , werden sie einen Performance-Gewinn bemerken, da die Daten nicht über Netz bereitgestellt werden müssen.
- Überall im Netz werden die Anwender einen Performance-Gewinn feststellen , auch auf den Rechnern, auf denen keine gespiegelte Kopie der Datenbank vorliegt, da die Menge der Netzzugriffe im Ganzen reduziert wird.

Diese Vorteile sind nicht umsonst zu haben. Datenreplikation erfordert natürlich mehr Speicher und das Aktualisieren benötigt mehr Rechenzeit.

Datenreplikation kann direkt in Anwenderprogrammen realisiert werden, wobei explizit angegeben werden muß, wohin die Daten gespiegelt werden sollen. Diese Methode ist allerdings teuer, fehleranfällig und schwer zu warten. Stattdessen wird Datenreplikation häufig mit Replikationstransparenz gekoppelt. Mit dieser Funktionalität werden die Details der Datenreplikation direkt vom Datenbankserver (und nicht vom Client) übernommen.

## Datenreplikation unter INFORMIX-OnLine Dynamic Server

**INFORMIX-OnLine Dynamic Server** realisiert transparente Datenreplikation für ganze Datenbankserver. Alle Daten, die von einem **INFORMIX-OnLine Dynamic Server**-Datenbankserver verwaltet werden, werden auf einen anderen **INFORMIX-OnLine Dynamic Server**-Datenbankserver gespiegelt und dort dynamisch aktualisiert. In der Regel liegen die beiden Datenbankserver auf verschiedenen Platten bzw. auf verschiedenen Rechnern. Datenreplikation unter **INFORMIX-OnLine Dynamic Server** wird häufig auch als "Hot Site Backup" bezeichnet, da mit dieser Methode eine Kopie des gesamten Datenbankservers erzeugt wird, die im Fall eines Systemabsturzes schnell eingesetzt werden kann.

Da **INFORMIX-OnLine Dynamic Server** über Replikationstransparenz verfügt, müssen Sie sich in der Regel keine Gedanken über Datenreplikation machen. Dies übernimmt der **INFORMIX-OnLine Dynamic Server**-Datenbankverwalter für Sie. Allerdings gelten für Datenreplikation spezielle Überlegungen zum Thema "Vernetzung" bei den Client-Rechnern. Diese sind im *INFORMIX-OnLine Administrator's Guide* beschrieben.

# Zusammenfassung

Der Datenbankzugriff wird durch Berechtigungen geregelt, die Ihnen der Eigentümer der Datenbank erteilt. Die Berechtigungen zum Abfragen von Daten werden meist automatisch vergeben. Dagegen erfordert das Ändern von Daten besondere Einfüge-, Lösch- oder Aktualisierungsrechte, die tabellenweise erteilt werden.

Wenn in einer Datenbank Datenintegritäts-Constraints verwendet werden, dann werden Ihre Möglichkeiten, die Daten zu ändern, durch Constraints eingeschränkt. Die Berechtigungen auf Datenbank- und Tabellenebene, sowie alle Daten-Constraints, legen fest, wann und wie Sie Daten ändern können.

Mit der DELETE-Anweisung können Sie einen oder mehrere Sätze aus einer Tabelle löschen. Die WHERE-Klausel in dieser Anweisung wählt die zu löschenden Sätze aus. Verwenden Sie die SELECT-Anweisung mit derselben Klausel, wenn Sie die Datensätze vorher ansehen wollen, die gelöscht werden. Mit der INSERT-Anweisung werden Datensätze einer Tabelle hinzugefügt. Sie können einen einzelnen Datensatz mit den gewünschten Spaltenwerten einfügen, oder Sie können eine Gruppe von Datensätzen einfügen, die mit einer SELECT-Anweisung ausgewählt wurden.

Mit der UPDATE-Anweisung können Sie Datensätze aktualisieren. Sie können die neuen Inhalte über Ausdrücke angeben, die Unterabfragen enthalten dürfen, wodurch die neuen Daten von den Daten aus anderen Tabellen, oder sogar aus der zu aktualisierenden Tabelle, abhängen können. Die Anweisung hat zwei Formen: in der ersten Form geben Sie die neuen Werte spaltenweise an; die zweite Form verwenden Sie dann, wenn eine SELECT-Anweisung oder eine Record-Variable eine Kombination von neuen Werten liefert.

Mit der REFERENCES-Klausel können Sie in der CREATE TABLE oder in der ALTER TABLE-Anweisung Verknüpfungen zwischen Tabellen herstellen. Mit den Schlüsselwörtern ON DELETE CASCADE in der REFERENCES-Klausel können Sie mit einem Befehl Sätze aus Eltern- und den zugehörigen Kindtabellen löschen.

Mit Transaktionen können Sie verhindern, daß unvorhersehbare Unterbrechungen eines Änderungsvorgangs die Datenbank in einem inkonsistenten Zustand lassen. Wenn Änderungen im Rahmen einer Transaktion durchgeführt werden, dann werden sie nach dem Auftreten eines Fehlers rückgängig gemacht. Das Transaktionsprotokoll ergänzt darüberhinaus die regelmäßig angefertigte Sicherungskopie der Datenbank, so daß sie wieder in den jüngsten und aktuellsten Zustand gebracht werden kann, wenn sie wiederhergestellt werden muß.

Kapitel

# SQL in Programmen

5

# Kapitelüberblick

**DB-Access** und **INFORMIX-SQL** ermöglichen Ihnen einen interaktiven Zugang zu SQL. Es gibt jedoch auch andere Möglichkeiten. Anwendungsprogramme werden für jeweils bestimmte Anwendungen geschrieben. Für die Erstellung solcher Programme stehen mehrere Programmiersprachen zur Verfügung, unter anderem **INFORMIX-ESQL/C** und **INFORMIX-ESQL/COBOL**.

Fast jedes Programm kann SQL-Anweisungen enthalten, diese ausführen und Daten vom Datenbankserver abrufen. Dieses Kapitel erläutert, wie diese Aktivitäten ausgeführt werden. Das Kapitel bietet lediglich eine Einführung in die Konzepte, die für die SQL-Programmierung in einer beliebigen Sprache allgemein gültig sind. Bevor Sie ein Programm in einer bestimmten Programmiersprache schreiben können, müssen Sie die jeweilige Sprache beherrschen. Sie müssen sich außerdem mit dem Handbuch vertraut machen, das die eingebettete Sprache beschreibt.

# SQL in Programmen

Sie können Programme in einer von mehreren Programmiersprachen schreiben. Dabei können Sie SQL-Anweisungen unter die anderen Anweisungen des Programms mischen, so als ob die SQL-Anweisungen normale Anweisungen der Programmiersprache wären. Die SQL-Anweisungen werden als in das Programm *eingebettet* bezeichnet und vom Programm sagt man, daß es *eingebettetes SQL* enthält. Man kürzt dies allgemein mit **ESQL** ab.

Informix erstellt für folgende Programmiersprachen **ESQL**-Produkte:

- C
- COBOL

Alle **ESQL**-Produkte arbeiten in ähnlicher Weise wie in Bild 5-1 gezeigt. Sie schreiben ein Quellprogramm, in dem Sie SQL-Anweisungen wie ausführbaren Code behandeln. Ihr Quellprogramm wird von einem **ESQL**-*Präprozessor*

verarbeitet. Dies ist ein Programm, das die eingebetteten SQL-Anweisungen ermittelt und diese in eine Reihe von Prozeduraufrufen und spezielle Datenstrukturen konvertiert.

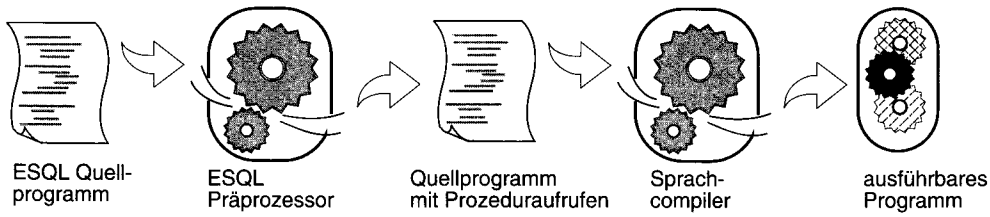



*Bild 5-1* *Überblick, wie ein Programm mit eingebetteten SQL-Anweisungen verarbeitet wird*

Anschließend wird das konvertierte Quellprogramm compiliert. Nachdem eine Bibliothek mit **ESQL**-Prozeduren hinzugebunden wurde, erzeugt der Compiler als Ausgabe ein ausführbares Programm. Die **ESQL**-Bibliotheksprozeduren werden zur Laufzeit aufgerufen; diese bauen Verbindungen zum Datenbankserver auf, um die SQL-Anweisungen auszuführen.

Während die **ESQL**-Produkte es Ihnen erlauben, SQL in die Host-Sprache einzubetten, gibt es einige Sprachen, bei denen SQL als Teil der Anweisungen existiert. Bei **INFORMIX-4GL** wird die Sprache SQL als ein fester Bestandteil der Sprache der vierten Generation unterstützt. Auch Informix Stored Procedure Language (SPL, eine Sprache für in der Datenbank gespeicherte Prozeduren) verwendet SQL als Teil der Anweisungen. Mit **INFORMIX-4GL** oder einem **ESQL**-Produkt kann man Anwendungsprogramme schreiben. Mit SPL kann man Prozeduren schreiben, die mit einer Datenbank gespeichert werden und die von einem Anwendungsprogramm aufgerufen werden.

## Statische Einbettung

Es gibt zwei Arten, SQL-Anweisungen in ein Programm einzuführen. Die einfachere und am weitesten verbreitete Art ist es, diese *statisch einzubetten*, d. h. die SQL-Anweisungen werden als Teil des Quellprogrammtextes geschrieben. Die Anweisungen werden als *statisch* bezeichnet, da sie ein fester Bestandteil des Quelltextes sind.

## Dynamische Anweisungen

Einige Anwendungen erfordern es, SQL-Anweisungen auf eine Benutzereingabe hin zu erstellen. Beispielsweise muß ein Programm in Abhängigkeit von Benutzerwünschen vielleicht jeweils andere Spalten auswählen oder unterschiedliche Kriterien auf Datensätze anwenden.

Diese Anforderung kann mit *dynamischem* SQL durchgeführt werden; hierbei erstellt das Programm im Speicher eine SQL-Anweisung als Zeichenkette und leitet diese Zeichenkette an den Datenbankserver zur Ausführung weiter. Dynamische Anweisungen sind kein Bestandteil des Quellprogrammtextes; sie werden während der Ausführung im Speicher erstellt.

## Programm- und Host-Variablen

Anwendungsprogramme können innerhalb von SQL-Anweisungen Programm-Variablen verwenden. Bei **INFORMIX-4GL** und SPL geben Sie die Programm-Variablen in die SQL-Anweisungen so ein, wie es die Syntax erlaubt. Beispielsweise kann eine DELETE-Anweisung eine Programm-Variable in der WHERE-Klausel verwenden. Bild 5-2 zeigt eine Programm-Variable bei **INFORMIX-4GL**.

```
MAIN
.
.
.
DEFINE drop_number INT
LET drop_number = 108
DELETE FROM items WHERE order_num = drop_number
.
.
.
```

*Bild 5-2* ***Verwendung einer Programm-Variablen bei INFORMIX-4GL***

Bild 5-3 zeigt eine Programm-Variable bei SPL.

```
CREATE PROCEDURE delete_item (drop_number INT)
.
.
.
DELETE FROM items WHERE order_num = drop_number
.
.
.
```

*Bild 5-3* ***Verwendung einer Programm-Variablen bei SPL***

In Anwendungen mit eingebetteten SQL-Anweisungen können die SQL-Anweisungen auf die Inhalte der Programm-Variablen verweisen. Eine in einer SQL-Anweisung genannte Programm-Variable wird als *Host-Variable* bezeichnet, da die SQL-Anweisung als "Gast" in dem Programm angesehen wird.

Die folgende DELETE-Anweisung könnte in ein COBOL-Quellprogramm eingebettet sein:

```
EXEC SQL
    DELETE FROM items
        WHERE order_num = :o-num
END-EXEC.
```

Die erste und letzte Zeile grenzen eingebettete SQL-Anweisungen von normalen COBOL-Anweisungen ab. Zwischen diesen Zeilen sehen Sie eine normale DELETE-Anweisung, so wie in Kapitel 4 beschrieben. Wenn dieser Teil des COBOL-Programms ausgeführt wird, dann wird aus der Tabelle **items** ein oder mehrere Datensätze gelöscht.

Die Anweisung enthält eine neue Funktion. Die Anweisung vergleicht die Spalte **order_num** mit einem Element, das als **:o-num** angegeben wurde. Dies ist der Name einer Host-Variablen.

Jedes **ESQL**-Produkt verfügt über spezifische Mittel, um die Namen von Host-Variablen zu kennzeichnen. Bei COBOL werden Host-Variablennamen mit einem einleitenden Doppelpunkt gekennzeichnet. Die Beispielanweisung fordert den Datenbankserver auf, die Datensätze zu löschen, bei denen die Auftragsnummer (**order_num**) gleich dem Inhalt der Host-Variablen **:o-num** ist. Dies ist vermutlich eine numerische Variable, die im Programm früher deklariert wurde und der früher ein Wert zugewiesen wurde.

Diesselbe Anweisung in ein FORTRAN-Programm eingebettet, sieht folgendermaßen aus:

```
EXEC SQL
    DELETE FROM items
        WHERE order_num = :onum
```

Dieselbe Anweisung in ein Ada-Programm eingebettet, sieht folgendermaßen aus:

```
EXEC SQL
    DELETE FROM items
        WHERE order_num = $onum;
```

Bei **INFORMIX-ESQL/Ada** wird eine Host-Variable durch ein vorangestelltes Dollarzeichen ($) gekennzeichnet und Anweisungen werden mit einem Semikolon beendet. Dasselbe gilt für **INFORMIX-ESQL/C**; hier wird die entsprechende DELETE-Anweisung wie folgt geschrieben:

```
EXEC SQL delete FROM items
    WHERE order_num = :onum;
```

Bei **INFORMIX-ESQL/C** kann eine SQL-Anweisung entweder mit einem vorangestellten Dollarzeichen ($) oder mit den Worten EXEC SQL eingeleitet werden.

Diese Syntaxunterschiede sind aber belanglos; die wesentlichen Punkte sind bei allen Sprachen (einer eingebetteten Sprache , **INFORMIX-4GL** oder SPL) die folgenden:

- Sie können SQL-Anweisungen in ein Quellprogramm einbetten, so als ob diese ausführbare Anweisungen der Host-Sprache wären.
- Sie können in SQL-Ausdrücken Programm-Variablen auf die gleiche Weise wie literale Werte verwenden.

Wenn Sie über Programmiererfahrung verfügen, dann können Sie sofort die Möglichkeiten erkennen. In diesem Beispiel wird die zu löschende Auftragsnummer der Variablen **onum** zugewiesen. Dieser Wert stammt aus einer beliebigen Quelle, die ein Programm benutzen kann: er kann aus einer Datei gelesen werden, das Programm kann den Benutzer auffordern, den Wert einzugeben oder der Wert kann aus der Datenbank gelesen werden. Die DELETE-Anweisung selbst kann ein Bestandteil einer Unterroutine sein (bei der **onum** ein Parameter der Unterroutine

); die Unterroutine kann dabei einmal oder wiederholt aufgerufen werden.

Kurz gesagt, wenn Sie SQL-Anweisungen in ein Programm einbetten, können Sie für diese Anweisung die Mächtigkeit der Host-Sprache verwenden. Sie können SQL hinter einer Vielzahl von Schnittstellen verstecken und Sie können die Funktionen von SQL auf viele Arten erweitern.

# Den Datenbankserver aufrufen

Die Ausführung einer SQL-Anweisung ist im Prinzip der Aufruf eines Datenbankservers aus einer Unterroutine heraus. Daten müssen vom Programm an den Datenbankserver und auch wieder zurück übertragen werden.

Einige dieser Übertragungen werden mittels Host-Variablen durchgeführt. Man kann sich hierbei der Host-Variablen bedienen, die in einer SQL-Anweisung als Parameter für den Prozeduraufruf des Datenbankservers angegeben wurden. Im vorangegangenen Beispiel war eine Host-Variable Bestandteil einer WHERE-Klausel. Weiterhin nehmen Host-Variablen auch Daten auf, die vom Datenbankserver zurückgeliefert werden; dies wird in einem Abschnitt weiter unten gezeigt.

## Der SQL-Übertragungsbereich (SQLCA)

Der Datenbankserver liefert immer einen Ergebnis-Code und möglicherweise auch weitere Informationen über die Auswirkungen der Operationen zurück. Die Ergebnisse werden in eine Datenstruktur übernommen, die man als Übertragungsbereich für SQL bezeichnet (SQLCA; SQL Communication Area). Wenn der Datenbankserver eine SQL-Anweisung in einer gespeicherten Prozedur (Stored Procedure) ausführt, dann enthält SQLCA der aufrufenden Anwendung diejenigen Werte, die von der SQL-Anweisung in der Prozedur ermittelt wurden.

Die Komponenten von SQLCA sind in Bild 5-4 und Bild 5-5 aufgelistet. Die Syntax, die Sie für die Beschreibung einer Datenstruktur (wie SQLCA) verwenden, hängt von der verwendeten Programmiersprache ab. Dies gilt auch für die Syntax, die Sie verwenden, um sich auf ein Element in der Datenstruktur zu beziehen. Details entnehmen Sie dem jeweiligen **ESQL** Handbuch.

Sie können außerdem die SQLSTATE-Variable aus der GET DIAGNOSTICS Anweisung verwenden, um Fehler zu entdecken und zu beheben. Sehen Sie dazu auch den Abschnitt "Der Wert SQLSTATE" auf Seite 5-13 nach.

Integer

**SQLCODE**

| | |
|---|---|
| **0** | erfolgreich. |
| **100** | Keine Daten mehr verfügbar/Daten nicht gefunden. |
| **negativ** | Fehlercode. |

Array mit 6 Integer-elementen

**SQLERRD**

| | |
|---|---|
| **erstes** | Nach einer erfolgreichen Aufbereitung einer SELECT-, UPDATE-, INSERT- oder DELETE-Anweisung oder nachdem ein Select-Cursor geöffnet wurde, enthält dieses Element die geschätzte Anzahl der verarbeiteten Sätze. Dieses Element wird bei **INFORMIX-ESQL/Ada** nicht verwendet. |
| **zweites** | Wenn SQLCODE einen Fehlercode enthält, dann enthält dieses Element entweder Null oder einen weiteren Fehlercode, genannt ISAM-Fehlercode; er erklärt die Fehlerursache.<br>Nach dem erfolgreichen Einfügen eines einzelnen Datensatzes enthält dieses Element den Wert der von SERIAL erzeugten fortlaufenden Zahl dieses Satzes. |
| **drittes** | Nach dem erfolgreichen Einfügen, Aktualisieren oder Löschen mehrerer Datensätze enthält dieses Element die Anzahl der verarbeiteten Sätze.<br>Nach dem fehlerhaften Einfügen, Aktualisieren oder Löschen mehrerer Sätze enthält dieses Element die Anzahl der Sätze, die vor Auftreten des Fehlers erfolgreich verarbeitet wurden. |
| **viertes** | Nach einer erfolgreich aufbereiteten SELECT-, UPDATE- oder DELETE-Anweisung oder nachdem der Select-Cursor geöffnet wurde, enthält dieses Element die geschätzte Anzahl der Plattenzugriffe und die aller zu verarbeiteten Sätze. Es wird bei **INFORMIX-ESQL/Ada** nicht verwendet. |
| **fünftes** | Nach einem Fehler in einer PREPARE-Anweisung enthält dieses Element den Bereich des Anweisungstextes, wo der Fehler entdeckt wurde. (Die Anweisung PREPARE wird im Abschnitt "Eine Anweisung aufbereiten" auf Seite 5-31 erläutert.) |
| **sechstes** | Nach dem erfolgreichen Holen eines ausgewählten Datensatzes oder nach einem erfolgreichen Einfügen, Aktualisieren oder Löschen enthält dieses Element die Rowid (physikalische Adresse) des letzten verarbeiteten Datensatzes. |

CHAR (8)

**SQLERRP**

Nur für interne Verwendung.

*Bild 5-4 Die Verwendung von SQLCODE und SQLERRD*

Array mit 8 CHAR-Elementen **SQLAWARN**

**Beim Öffnen einer Datenbank:**

| | |
|---|---|
| **erstes** | Wird auf *W* gesetzt, wenn irgendein anderes Element auf *W* gesetzt ist. Wenn das Element leer ist, dann müssen die anderen nicht überprüft werden. |
| **zweites** | Wird auf *W* gesetzt, wenn die aktuelle Datenbank mit Transaktionsprotokoll arbeitet. |
| **drittes** | Wird auf *W* gesetzt, wenn die aktuell geöffnete Datenbank ANSI-kompatibel ist. |
| **viertes** | Wird auf *W* gesetzt, wenn ein **INFORMIX-OnLine Dynamic Server** Datenbankserver verwendet wird. |
| **fünftes** | Wird auf *W* gesetzt, wenn der Datenbankserver den Datentyp FLOAT als DECIMAL speichert (dies wird gemacht, wenn das Host-System FLOAT-Typen nicht unterstützt). |
| **sechstes** | Wird nicht verwendet. |
| **siebtes** | Wird auf W gesetzt, wenn die Anwendung mit einem **INFORMIX-OnLine Dynamic Server** Datenbankserver als Datenreplikations-Server verwendet wird. (In diesem Fall läßt der Datenbankserver nur Lesezugriffe zu.) |
| **achtes** | Wird nicht verwendet. |

**Alle anderen Vorgänge:**

| | |
|---|---|
| **erstes** | Wird auf *W* gesetzt, wenn ein beliebiges anderes Element auf *W* gesetzt wird. |
| **zweites** | Wird auf *W* gesetzt, wenn ein Spaltenwert beim Einfügen in eine Host-Variable gekürzt wird. |
| **drittes** | Wird auf *W* gesetzt, wenn eine Mengenfunktion auf einen NULL-Wert stößt. |
| **viertes** | Wird bei einem Select-Cursor oder beim Öffnen eines Cursors auf *W* gesetzt, wenn die Anzahl der Elemente in der Auswahlliste nicht mit der Anzahl der Host-Variablen übereinstimmt, die in der INTO-Klausel angegeben sind, um die Daten aufzunehmen. |
| **fünftes** | Wird nach der Aufbereitung einer UPDATE- oder DELETE-Anweisung auf *W* gesetzt, wenn die aufbereitete Anweisung keine WHERE-Klausel enthält und sie somit die gesamte Tabelle betrifft. |
| **sechstes** | Wird nach der Ausführung einer Anweisung, die nicht der SQL-Syntax des ANSI-Standards entspricht, auf *W* gesetzt (vorausgesetzt, die Umgebungsvariable DBANSIWARN ist gesetzt). |
| **siebtes** | Wird nicht verwendet. |
| **achtes** | Wird nicht verwendet. |

CHAR (71) Element **SQLERRM**

Enthält eine Variable, wie z.B. eine Tabellennamen, der in einer Fehlermeldung erscheint. Bei manchen Netzanwendungen enthält es eine Fehlermeldung .

*Bild 5-5 Die Verwendung von SQLAWARN*

*Hinweis:* Die Feldausschnitte, über die Sie ein Element eines SQLERRD- und eines SQLAWARN-Arrays ansprechen, unterscheiden sich: die Numerierung der Array-Elemente beginnt bei **INFORMIX-ESQL/C** mit Null, bei allen anderen Sprachen beginnt die Numerierung mit 1. In dieser Erläuterung werden für die Bezeichnung der Elemente eindeutige Wörter wie z. B. *drittes* verwendet. Sie müssen diese Bezeichnungen dann auf die Syntax Ihrer Programmiersprache übertragen.

# Der SQLCODE-Array

Der SQLCODE-Array erteilt den primären Rückgabewert des Datenbankservers. Nach jeder SQL-Anweisung wird SQLCODE auf einen INTEGER-Wert gesetzt; diese Werte wurden in Bild 5-4 gezeigt. Wenn der Wert Null (0) ist, dann wurde die Anweisung fehlerfrei durchgeführt. Bei einer Anweisung, die Daten an Host-Variablen zurückgeben soll, bedeutet der Wert Null (0), daß die Daten zurückgeliefert wurden und verwendet werden können. Ein Wert ungleich Null bedeutet das Gegenteil: an die Host-Variablen wurden keine sinnvollen Daten geliefert.

Bei **INFORMIX-4GL** kann auf den SQLCODE-Array auch unter dem Namen STATUS zugegriffen werden.

## Keine Daten verfügbar

Wenn die Anweisung richtig durchgeführt wurde, aber keine Datensätze gefunden wurden, dann setzt der Datenbankserver SQLCODE auf 100. Dieser Fall kann in zwei Situationen eintreten:

Bei der Abfrage mit einem Cursor. (Diese Art von Abfragen wird im Abschnitt "Nach mehreren Sätzen suchen" auf Seite 5-20 beschrieben.) Bei diesen Abfragen holt die FETCH-Anweisung die einzelnen Werte aus der aktiven Tabelle in den Speicher. Nach dem letzten Datensatz wird die folgende FETCH-Anweisung keinen Satz mehr finden. Anschließend setzt der Datenbankserver SQLCODE auf 100, um zu zeigen, daß er am Ende angekommen ist und keine Sätze gefunden wurden.

Bei einer Abfrage ohne Cursor. Der Datenbankserver setzt SQLCODE auf 100, wenn kein Datensatz die Suchbedingung erfüllt. Bei ANSI-kompatiblen Datenbanken führen die Anweisungen SELECT, DELETE, UPDATE und INSERT alle zu dem Wert 100 in SQLCODE, wenn keine passenden Sätze gefunden wurden. Bei Datenbanken, die *nicht* ANSI-kompatibel sind, führt nur die erfolglose SELECT-Anweisung zu dem Wert 100 in SQLCODE.

### Negative Werte

Wenn während der Ausführung einer Anweisung unerwartet ein Fehler auftritt, dann liefert der Datenbankserver als SQLCODE einen negativen Wert zurück. Die Bedeutungen dieser Werte sind im Handbuch *Informix Fehlermeldungen* und in der Online-Fehlermeldungsdatei dokumentiert.

## Der Array SQLERRD

Einige Fehlerwerte, die in SQLCODE enthalten sein können, zeigen allgemeine Probleme an. Der Datenbankserver kann das zweite Element von SQLERRD mit einem ausführlichen Code belegen. Dieser Wert zeigt den ersten Fehler an, auf den die I/O-Routine des Datenbankservers oder das zugrunde liegende Betriebssystem gestoßen ist.

Die Integer-Elemente im Array SQLERRD sind nach unterschiedlichen Anweisungen auf unterschiedliche Werte gesetzt. Das erste und das vierte Element werden nur bei **INFORMIX-4GL**, **INFORMIX-ESQL/C** und **INFORMIX-ESQL/COBOL** verwendet. Die Elemente werden wie in Bild 5-4 gezeigt verwendet.

Diese zusätzlichen Angaben können sehr sinnvoll sein. Sie können z. B. den Wert des dritten Elements verwenden, um die Anzahl der gelöschten oder aktualisierten Sätze zu ermitteln. Wenn Ihr Programm eine von einem Benutzer eingegebene SQL-Anweisung aufbereitet und dabei einen Fehler entdeckt, dann ermöglicht Ihnen der Wert des fünften Elements, dem Benutzer die genaue Fehlerposition anzuzeigen. (**DB-Access** und **INFORMIX-SQL** verwenden diese Funktion für die Positionierung der Schreibmarke, wenn Sie nach einem aufgetretenen Fehler die Anweisung korrigieren wollen.)

## Der Array SQLAWARN

Die acht CHAR-Elemente des Arrays SQLAWARN sind entweder Leerzeichen oder auf `W` gesetzt; diese zeigen eine Auswahl besonderer Bedingungen an. Es werden nur die ersten sechs Elemente benutzt. Deren Bedeutung hängt von der zuletzt ausgeführten Anweisung ab.

Es gibt eine Reihe von Warnschaltern, die direkt nach dem Öffnen einer Datenbank zur Wirkung kommen, also unmittelbar nach Ausführung der Anweisungen CONNECT, DATABASE oder CREATE DATABASE. Diese Schalter zeigen Ihnen einige Eigenschaften der Datenbank als Ganzes auf.

Eine zweite Reihe von Schaltern kommt nach jeder beliebigen anderen Anweisung zur Wirkung. Diese Schalter zeigen ungewöhnliche Ereignisse während der Ausführung einer Anweisung auf; Ereignisse, die vom SQLCODE möglicherweise nicht angezeigt werden.

Die beiden Möglichkeiten für Belegungen von SQLAWARN sind in Bild 5-5 zusammengefaßt.

## Der Wert SQLSTATE

Einige INFORMIX-Produkte wie **INFORMIX-ESQL/C** oder **INFORMIX-ESQL/COBOL** unterstützen entsprechend den Standards ANSI und X/Open den Wert SQLSTATE. Die Anweisung GET DIAGNOSTICS liest den Wert aus SQLSTATE, um mögliche Fehler zu diagnostizieren, die bei einer SQL-Anweisung aufgetreten sind.

Als Ergebnis liefert der Datenbankserver eine fünfstellige Zeichenkette zurück, die in der Variable SQLSTATE abgelegt wird. Dieser Wert liefert folgende Informationen über die gerade ausgeführt SQL-Anweisung:

- Die Anweisung war erfolgreich.
- Die Anweisung war erfolgreich, es wurden aber trotzdem Warnungen ausgegeben.
- Die Anweisung war erfolgreich, hat aber keine Daten zurückgeliefert.
- Die Anweisung war nicht erfolgreich.

Weitere Informationen zu diesem Thema erhalten Sie im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*. Falls Ihre INFORMIX-Produkte GET DIAGNOSTICS und SQLSTATE unterstützen, empfehlen wir Ihnen diese als wertvolles Mittel, Fehler zu entdecken, zu diagnostizieren und zu beheben.

# Einzelne Datensätze abrufen

Um einzelne Datensätze aus der Datenbank abzurufen und um sie in Host-Variablen einzufügen, können Sie eingebettete SQL-Anweisungen verwenden. Wenn eine SELECT-Anweisung jedoch mehr als einen Datensatz zurückliefert, dann muß im Programm eine komplizierte Methode verwendet werden, um immer nur einen der Sätze zu holen. Die Auswahl mehrerer Datensätze wird in diesem Kapitel weiter unten erläutert.

Um einen einzelnen Satz abzurufen, betten Sie in Ihr Programm einfach eine SELECT-Anweisung ein. Hier ein Beispiel, wie es bei **INFORMIX-ESQL/C** geschrieben werden kann:

```
exec sql select avg (total_price)
    into :avg_price
    from items
    where order_num in
        (select order_num from orders
        where order_date < date('6/1/93'));
```

Nur durch die INTO-Klausel unterscheidet sich diese Anweisung von jedem beliebigen Beispiel aus den Kapiteln 2 oder 3. Diese Klausel gibt die Host-Variablen an, die die gelieferten Daten aufnehmen.

Wenn ein Programm eine eingebettete SELECT-Anweisung ausführt, dann führt der Datenbankserver die Abfrage durch. Die Anweisung im Beispiel wählt einen Mengenwert aus; es wird genau ein Datensatz geliefert. Dieser Satz besteht nur aus einer einzigen Spalte und der Wert dieser Spalte wird in der Host-Variablen **avg_price** abgelegt. Diese Variable kann in nachfolgenden Programmzeilen verwendet werden.

Um die Daten einzelner Sätze abzurufen und diese in Host-Variablen abzulegen, können Sie Anweisungen dieser Art verwenden. Der einzelne Datensatz kann dabei soviele Spalten enthalten, wie Sie wünschen. In diesem **INFORMIX-4GL** -Beispiel werden Host-Variablen auf zwei Arten verwendet: als Empfänger von Daten und in der WHERE-Klausel:

```
DEFINE cfname, clname, ccompany CHAR(20)
DEFINE cnumbr INTEGER
LET cnumbr = 104
SELECT fname, lname, company
    INTO cfname, clname, ccompany
    FROM customer
    WHERE customer_num = cnumbr
```

Da die Spalte **customer_num** über einen UNIQUE-Index verfügt (über einen Constraint festgelegt), liefert diese Abfrage nur einen einzigen Datensatz. Ermittelt eine Abfrage mehr als einen Datensatz, dann kann der Datenbankserver überhaupt keine Daten liefern. Anstelle von Daten liefert er eine Fehlernummer zurück.

In der INTO-Klausel sollten die Anzahl der angegebenen Host-Variablen mit der Anzahl der in der Auswahlliste enthaltenen Elemente übereinstimmen. Wenn diese Listen unterschiedlich lang sind, dann liefert der Datenbank-Server soviele Werte wie möglich zurück und setzt im vierten Element von SQLAWARN den Warnschalter.

## Die Umwandlung von Datentypen

Im folgenden Beispiel wird der Durchschnittswert einer DECIMAL-Spalte abgerufen. Der Durchschnittswert selbst ist auch eine DECIMAL-Wert. Es ist jedoch *nicht* erforderlich, daß die Host-Variable, in der der Wert abgelegt wird, vom gleichen Datentyp ist.

```
exec sql select avg (total_price) into :avg_price
    from items;
```

Im nächsten Beispiel eines **ESQL/C**-Codes wird die Deklaration der empfangenden Variable **avg_price** nicht gezeigt. Es könnte sich um eine der folgenden Definitionen handeln:

```
    int avg_price;
    double avg_price;
    char avg_price[16];
    dec_t avg_price; /* typedef of decimal number structure */
```

Der Datentyp einer jeden verwendeten Host-Variable wird vermerkt und zusammen mit der Anweisung an den Datenbankserver weitergegeben. Der Datenbankserver versucht, die Daten in das Format umzuwandeln, das von den empfangenden Variablen verwendet wird. Obwohl einige Umwandlungen einen Verlust an Genauigkeit verursachen, ist fast jede Umwandlung erlaubt. Abhängig vom jeweiligen Datentyp der empfangenden Host-Variablen unterscheiden sich die Ergebnisse des vorangegangenen Beispiels folgendermaßen:

FLOAT — Der Datenbankserver wandelt das dezimale Ergebnis in den Datentyp FLOAT um; dabei könnten eventuell einige Nachkommastellen abgeschnitten werden.

Wenn die Größe des Dezimalwertes die maximale Größe des FLOAT-Formats überschreitet, wird ein Fehler zurückgeliefert.

| | |
|---|---|
| INTEGER | Der Datenbankserver wandelt das Ergebnis in einen INTEGER-Wert um; notfalls werden dabei Nachkommastellen abgeschnitten.<br><br>Wenn der ganzzahlige Teil der umgewandelten Zahl nicht in die empfangende Variable paßt, tritt ein Fehler auf. |
| CHARACTER | Der Datenbankserver wandelt einen Dezimalwert in eine CHAR-Zeichenkette um.<br><br>Wenn die Zeichenkette für die empfangende Variable zu lang ist, wird sie abgeschnitten, das zweite Element von SQLAWARN wird auf `W` gesetzt und die Variable SQLSTATE wird auf 01000 gesetzt. |

## Die Behandlung von NULL-Werten

Was passiert, wenn ein Programm einen NULL-Wert abruft? NULL-Werte können in der Datenbank gespeichert werden, aber die Datentypen, die von den Programmiersprachen unterstützt werden, erkennen den Status NULL nicht. Um die Verarbeitung von NULL-Werten als Daten zu vermeiden, muß ein Programm über Möglichkeiten verfügen, diese Werte zu erkennen.

In Produkten mit eingebetteter Sprache übernehmen *Indikator-Variablen* diese Aufgabe. Eine Indikator-Variable ist eine zusätzliche Variable, die mit einer Host-Variablen verbunden ist; und zwar mit einer Host-Variablen, die einen NULL-Wert aufnehmen könnte. Wenn der Datenbankserver in die Haupt-Variable Daten einfügt, dann fügt er auch in die Indikator-Variable einen speziellen Wert ein; dieser zeigt an, ob ein NULL-Wert vorliegt.

```
exec sql select paid_date
         into :op_date:op_d_ind
         from orders
         where order_num = $the_order;
    if (op_d_ind < 0) /* data was null */
         rstrdate ('01/01/1900', :op_date);
```

In diesem **INFORMIX-ESQL/C**-Beispiel wird ein einzelner Datensatz ausgewählt und in die Host-Variable **op_date** wird ein einzelner Wert eingetragen. Da dieser Wert ein NULL-Wert sein könnte, wird eine Indikator-Variable namens **op_d_ind** mit der Host-Variablen verbunden. (Die Indikator-Variable muß irgendwo im Programm als "short integer" deklariert werden.)

Nach der Ausführung der SELECT-Anweisung prüft das Programm die Indikator-Variable auf einen negativen Wert hin ab. Ein negativer Wert (normalerweise -1) bedeutet, daß der Wert in der Haupt-Variablen ein NULL-Wert

ist. Wenn dies der Fall ist, dann verwendet dieses Programm eine C-Funktion, um der Host-Variablen einen Standard-Wert zuzuweisen. (Die Funktion **rstrdate** ist Bestandteil des Produktes **INFORMIX-ESQL/C**.)

Die Syntax, die Sie verwenden, um eine Indikator-Variable einzubinden, hängt von der verwendeten Sprache ab; das Prinzip ist aber bei allen gleich. Bei **INFORMIX-4GL** oder SPL werden Indikator-Variablen jedoch explizit verwendet, da diese Sprachen NULL-Werte für Variablen unterstützen. In **4GL** wird das vorangegangene Beispiel wie folgt geschrieben:

```
SELECT paid_date
    INTO op_date
    FROM orders
    WHERE order_num = the_order
IF op_date IS NULL THEN
    LET op_date = date ('01/01/1900')
END IF
```

# Fehlerbehandlung

Obwohl der Datenbankserver die Umwandlung der Datentypen automatisch durchführt, gibt es dennoch einige Dinge, die bei einer SELECT-Anweisung falsch laufen können.

## Keine Daten verfügbar

Es kommt häufig vor, daß eine Abfrage keine Sätze liefert. Dies wird nach einer SELECT-Anweisung durch den Wert 02000 in der Variablen SQLSTATE und durch den Wert 100 im SQLCODE (oder im **sqlca.sqlcode**, wie er bei **ESQL/C** auch bezeichnet wird) angezeigt. Dieser Wert zeigt je nach Anwendung einen Fehler oder aber auch ein normales Ereignis an. Wenn Sie sich sicher sind, daß es einen Satz oder mehrere Sätze geben sollte – z. B. wenn Sie einen Datensatz anhand eines Schlüsselwertes gelesen haben, den Sie gerade aus einem Satz einer anderen Tabelle gelesen haben – dann zeigt der Wert 100 an, daß in der Programmlogik ein schwerwiegender Fehler vorliegt. Andererseits kann das Fehlen von Daten ein normales Ereignis sein, wenn Sie einen Datensatz mit einem Schlüssel auszuwählen versuchen, der von einem Benutzer oder einer anderen Quelle stammt, die nicht so zuverlässig wie ein Programm ist. In diesem Fall existiert der gesuchte Datensatz tatsächlich nicht.

### Schwerwiegende Fehler

Fehler, die einen negativen Wert in der Variable SQLCODE zur Folge haben, sind in der Regel schwerwiegend. Programme, die bereits im Einsatz sind, sollten diese Fehler nicht mehr zeigen. Da man aber nicht jedes Problem vorwegnehmen kann, müssen Ihre Programme mit diesen Fehlern umgehen können.

Eine Abfrage kann beispielsweise die Fehlernummer -206 zurückliefern. Das kommt dann vor, wenn jemand nach der Programmerstellung die Tabelle gelöscht hat oder wenn das Programm aufgrund eines logischen Fehlers oder eines Eingabefehlers die falsche Datenbank geöffnet hat.

### Der Wert 100 (keine Daten verfügbar) bei Nicht-ASCII-Datenbanken

Wenn Ihre Datenbank nicht ANSI-kompatibel ist, wird der Rückgabewert 100 in SQLCODE nur nach SELECT-Anweisungen gesetzt. Außerdem wird der Wert des SQLSTATE auf 02000 gesetzt. (Andere Anweisungen wie INSERT, UPDATE und DELETE verwenden das dritte Element von SQLERRD, um anzuzeigen, wie viele Datensätze sie betreffen; dieses Thema wird im Kapitel 6 dieses Handbuchs behandelt.)

### Mengenfunktionen

Eine SELECT-Anweisung, die eine Mengenfunktion wie SUM, MIN oder AVG auswählt, ist immer erfolgreich und liefert genau einen Datensatz. Dies stimmt auch dann, wenn keine Sätze der WHERE-Klausel entsprechen; der Mengenwert, der sich aus einer leeren Anzahl von Datensätzen ergibt, ist der NULL-Wert; er ist aber auf jeden Fall vorhanden.

Ein Mengenwert ist jedoch auch dann ein NULL-Wert, wenn er sich aus Sätzen errechnet, die alle NULL-Werte enthalten. Wenn Sie den Unterschied zwischen einem Mengenwert herausfinden müssen, der auf keinen Sätzen basiert und einem , der auf Sätzen basiert, die alle NULL-Werte enthalten, dann müssen Sie in die Anweisung eine COUNT-Funktion und eine Indikator-Variable für den Mengenwert einbauen. Damit können Sie die folgenden drei Fälle herausarbeiten:

| Zählwert | Indikator | Fall |
|---|---|---|
| 0 | -1 | es wurden keine Sätze ausgewählt |
| >0 | -1 | es wurden einige Sätze ausgewählt, die alle NULL-Werte enthielten |
| >0 | 0 | es wurden einige Sätze ausgewählt, die keine NULL-Werte enthielten |

## Die Verwendung von Standardwerten

Diese unvermeidbaren Fehler können auf viele Arten behandelt werden. Einige Anwendungen enthalten mehr Programmzeilen für die Fehlerbehandlung als für normale Situationen. In den Beispielen dieses Abschnitts sollte jedoch die einfachste Methode, der Standardwert, genügen.

```
avg_price = 0; /* set default for errors */
$   SELECT AVG (total_price)
        INTO $avg_price:null_flag
        FROM items;
if (null_flag < 0) /* probably no rows */
    avg_price = 0; /* set default for 0 rows */
```

Dieses Beispiel basiert auf folgenden Überlegungen:

- Wenn die Abfrage Sätze auswählt, die keine NULL-Werte enthalten, wird der richtige Wert zurückgeliefert und verwendet. Dies ist das erwartete und auch häufigste Ergebnis.
- Wenn die Abfrage keine Sätze auswählt, wird die Indikator-Variable gesetzt und ein Standardwert wird zugewiesen. Dies geschieht auch in dem sehr unwahrscheinlichen Fall, daß nur einige Sätze ausgewählt wurden, die alle in der Spalte **total_price** (eine Spalte, die nie einen NULL-Wert enthalten sollte) NULL-Werte enthalten.
- Wenn ein anderer schwerwiegender Fehler auftritt, dann bleibt die Host-Variable unverändert; sie enthält den eingangs gesetzten Standardwert. An dieser Programmstelle sieht der Programmierer keine Notwendigkeit, solche Fehler aufzufangen und diese zu melden.

Das nächste Beispiel ist eine Erweiterung eines früheren **INFORMIX-4GL**-Beispiels; es zeigt Standardwerte an, wenn die vom Benutzer gewünschte Firma nicht gefunden wird:

```
DEFINE cfname, clname, ccompany CHAR(20)
DEFINE cnumbr INTEGER
PROMPT 'Enter the customer number: ' FOR cnumbr
LET cfname = 'unknown'
LET clname = 'person'
LET ccompany = 'noplace'
SELECT fname, lname, company
    INTO cfname, clname, ccompany
    WHERE customer_num = cnumbr
DISPLAY cfname, ' ', clname, ' at ', ccompany
```

Wenn es keinen Satz gibt, der mit der vom Benutzer eingegebenen Kundennummer übereinstimmt, wird SQLCODE auf 100 gesetzt und die Host-Variablen bleiben unverändert. Dies gilt, da diese Abfrage keine Mengenfunktionen verwendet.

# Nach mehreren Sätzen suchen

Wenn eine Abfrage mehr als einen Satz liefert, dann muß das Programm die Abfrage anders behandeln. Abfragen nach mehreren Sätzen werden in zwei Phasen ausgeführt. Zuerst startet das Programm die Abfrage; hierbei werden die Daten nicht sofort geliefert. Anschließend ruft das Programm die Datensätze einzeln ab.

Die Anweisungen werden unter Verwendung eines speziellen Datenobjekts, genannt *Cursor*, durchgeführt. Ein Cursor ist eine Datenstruktur, die den aktuellen Stand der Abfrage anzeigt. Die allgemeine Reihenfolge der Anweisungen im Programm ist wie folgt:

1. Das Programm *deklariert (declare)* den Cursor und die damit verbundene SELECT-Anweisung. Dieser Vorgang fordert nur Speicherplatz zum Halten des Cursors an.
2. Das Programm *öffnet (open)* den Cursor. Dieser Schritt beginnt die Ausführung der verbundenen SELECT-Anweisung und ermittelt jeden Fehler, der in der Anweisung enthalten ist.
3. Das Programm *holt (fetch)* einen Cursor, überträgt ihn in die Host-Variable und verarbeitet ihn.
4. Das Programm *schließt (close)* den Cursor, nachdem der letzte Satz geholt wurde.

Diese Schritte werden zusammen mit den SQL-Anweisungen DECLARE, OPEN, FETCH und CLOSE durchgeführt.

## Einen Cursor deklarieren

Ein Cursor wird mit der Anweisung DECLARE deklariert. Diese Anweisung gibt dem Cursor den Namen, bestimmt seine Verwendung und verbindet ihn mit einer Anweisung. Hier ist ein einfaches Beispiel in **INFORMIX-4GL**:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
    INTO o_num, i_num, s_num
    FROM items
```

Die Deklaration gibt dem Cursor einen Namen (in diesem Fall **the_item**) und verbindet ihn mit einer SELECT-Anweisung. (In Kapitel 6 dieses Handbuchs wird erläutert, wie ein Cursor auch mit einer INSERT-Anweisung verbunden werden kann.)

Die SELECT-Anweisung dieses Beispiels enthält eine INTO-Klausel. Dies ist eine von zwei Möglichkeiten, wie man Variablen angeben kann, die die Daten aufnehmen. Die andere Möglichkeit wird im Abschnitt "Die Position der INTO-Klausel" auf Seite 5-23 beschrieben.

Die DECLARE-Anweisung ist keine aktive Anwendung; sie richtet lediglich die Bearbeitungsmöglichkeiten für den Cursor ein und fordert für ihn Speicherplatz an. Sie können den im vorherigen Beispiel deklarierten Cursor verwenden, um die Tabelle **items** einmal zu lesen. Cursor können so deklariert sein, daß sie sowohl vorwärts als auch rückwärts lesen können (siehe "Eingabearten für den Cursor" auf Seite 5-24). Da die FOR UPDATE-Klausel fehlt, wird dieser Cursor wahrscheinlich nur zum Lesen von Daten und nicht zum Verändern der Daten verwendet. (Die Verwendung eines Cursors zum Verändern von Daten ist im Kapitel 6 dieses Handbuchs enthalten.)

## Einen Cursor öffnen

Das Programm öffnet den Cursor bei Bedarf. Die OPEN-Anweisung aktiviert den Cursor. Sie leitet die verbundene SELECT-Anweisung an den Datenbankserver weiter, der mit der Suche nach den zugehörigen Sätzen beginnt. Der Datenbankserver verarbeitet die Abfrage bis zu dem Punkt, an dem der erste Ausgabesatz gefunden wird. Der Datenbankserver liefert diesen Satz nicht wirklich, sondern übergibt an SQLCODE einen Rückgabewert. Dies ist eine OPEN-Anweisung bei **INFORMIX-4GL**:

```
OPEN the_item
```

Da der Datenbankserver zu diesem Zeitpunkt die Abfrage zum ersten Mal verarbeitet hat, ist dies auch der Zeitpunkt, an dem viele Fehler gefunden werden. Nach dem Öffnen des Cursors sollte das Programm SQLSTATE oder SQLCODE überprüfen. Wenn SQLSTATE größer als 2000 oder SQLCODE einen negativen Wert enthält, dann ist der Cursor unbrauchbar. Es kann ein Fehler in der SELECT-Anweisung vorliegen oder andere Probleme können den Datenbankserver an der Ausführung dieser Anweisung hindern.

Wenn SQLSTATE 00000 oder SQLCODE Null (0) enthält, dann ist die SELECT-Anweisung syntaktisch richtig und der Cursor ist zur Ausführung bereit. Zu diesem Zeitpunkt weiß das Programm jedoch noch nicht, ob der Cursor Sätze liefern kann.

# Datensätze holen

Das Programm verwendet die FETCH-Anweisung, um jeden Ausgabesatz abzurufen. Diese Anweisung gibt den Namen des Cursors an; sie kann auch die Host-Variablen angeben, die die Daten aufnehmen sollen. Hier ist das **INFORMIX-4GL**-Beispiel vervollständigt:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        INTO o_num, i_num, s_num
        FROM items
OPEN the_item
WHILE sqlcode = 0
    FETCH the_item
    IF sqlcode = 0 THEN
        DISPLAY o_num, i_num, s_num
    END IF
END WHILE
```

## Ermitteln, ob Daten verfügbar sind

Falls die OPEN-Anweisung einen Fehler zurückliefert, verhindert in diesem Beispiel die WHILE-Bedingung die Ausführung der Schleife. Dieselbe Bedingung beendet die Schleife, wenn SQLCODE auf 100 gesetzt wird. Dies zeigt an, daß keine Daten verfügbar sind. Jedoch wird SQLCODE auch innerhalb der Schleife überprüft. Diese Überprüfung ist notwendig, da bei einer richtigen SELECT-Anweisung, die keine passenden Sätze findet, die OPEN-Anweisung den SQLCODE Null (0) zurückliefert; das erste Holen liefert aber den Wert 100 (d. h. keine Daten verfügbar) zurück und keine Daten. Hier ist eine andere Möglichkeit, dieselbe Schleife zu schreiben:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        FROM items
OPEN the_item
IF sqlcode = 0 THEN
    FETCH the_item -- fetch first row
END IF
WHILE sqlcode = 0
    DISPLAY o_num, i_num, s_num
    FETCH the_item
END WHILE
```

In dieser Version wird der Fall, daß keine Sätze geliefert werden, früher behandelt. Aus diesem Grund gibt es keine zweite Überprüfung von **sqlcode** in der Schleife. Für diese Versionen gibt es keinen meßbaren Unterschied in der Performance, da der Zeitbedarf für die Überprüfung des **sqlcode** nur einen winzigen Bruchteil des Zeitbedarfs für das Holen ausmacht.

## Die Position der INTO-Klausel

Die INTO-Klausel gibt die Namen der Host-Variablen an, die die vom Datenbankserver gelieferten Daten aufnehmen sollen. Die Klausel muß entweder in der SELECT- oder der FETCH-Anweisung enthalten sein, jedoch nicht in beiden. Hier ist das Beispiel so umgestellt, daß die Host-Variablen in der FETCH-Anweisung angegeben werden:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        FROM items
OPEN the_item
WHILE status = 0
    FETCH the_item INTO o_num, i_num, s_num
    IF status = 0 THEN
        DISPLAY o_num, i_num, s_num
    END IF
END WHILE
```

Die zweite Form hat den Vorteil, daß unterschiedliche Datensätze in unterschiedliche Variablen geladen werden können. Diese Form können Sie z. B. verwenden, um aufeinanderfolgende Sätze in aufeinanderfolgende Elemente eines Arrays zu laden.

## Eingabearten für den Cursor

Um mit dem Cursor Eingaben durchzuführen, kann man einen Cursor auf zwei Arten verwenden: *sequentiell* oder *scrollend*. Ein sequentieller Cursor kann nur den jeweils nächsten Satz holen. Daher kann der Cursor eine Tabelle nur einmal lesen, sobald sie geöffnet ist. Ein Scroll-Cursor kann den nächsten Satz oder jeden beliebigen vorherigen Satz holen; er kann Sätze somit mehrfach lesen. Hier wird ein sequentieller Cursor in **INFORMIX-ESQL/C** deklariert:

```
EXEC SQL DECLARE pcurs CURSOR FOR
    SELECT customer_num, lname, city
        FROM customer;
```

Nachdem er geöffnet ist, kann man diesen Cursor nur für sequentielles Holen des jeweils nächsten Satzes verwenden.

```
EXEC SQL FETCH p_curs INTO :cnum, :clname, :ccity;
```

Jedes sequentielle Holen liefert einen neuen Satz zurück.

Ein Scroll-Cursor wird mit dem Schlüsselwort SCROLL deklariert; dies wird in dem **INFORMIX-ESQL/FORTRAN**-Beispiel gezeigt:

```
    EXEC SQL DECLARE s_curs SCROLL CURSOR FOR
   +    SELECT order_num, order_date FROM orders
   +        WHERE customer_num > 104
```

Ein Scroll-Cursor kann mit einer Vielzahl von Hol-Optionen verwendet werden. Die Option ABSOLUTE spezifiziert die Nummer des Satzes, der geholt werden soll.

```
    EXEC SQL FETCH ABSOLUTE numrow s_curs
   +    INTO :nordr, :nodat
```

Diese Anweisung holt den Satz, dessen Position in der Host-Variablen **numrow** angegeben ist. Es ist auch möglich, den aktuellen Satz nochmals zu holen oder den ersten Satz zu holen und dann die ganze Liste nochmal zu durchsuchen. Diese Funktionen haben jedoch Nachteile, die im nächsten Abschnitt beschrieben sind.

# Die Ergebnistabelle eines Cursors

Sobald ein Cursor geöffnet ist, steht er für die Auswahl von Datensätzen zur Verfügung. Die Menge aller Sätze, die die Abfrage liefert, wird als *Ergebnistabelle* des Cursors bezeichnet. Eine Ergebnistabelle kann man sich einfach als eine Ansammlung von Datensätzen vorstellen; und den Cursor kann man sich als einen Zeiger auf einen dieser Datensätze vorstellen. Solange keine anderen Programme dieselben Daten parallel verändern, entspricht dieses Bild der Wirklichkeit.

## Eine Ergebnistabelle erzeugen

Sobald der Cursor geöffnet ist, führt der Datenbankserver all das aus, was für das Auffinden des ersten ausgewählten Datensatzes notwendig ist. Abhängig von der Formulierung der Abfrage kann dies sehr einfach sein, es kann aber auch einen großen Arbeits- und Zeitaufwand erfordern.

```
DECLARE easy CURSOR FOR
    SELECT fname, lname FROM customer
        WHERE state = 'NJ'
```

Da dieser Cursor auf einfache Weise nur eine Tabelle abfragt, kann der Datenbankserver sehr schnell herausfinden, ob Datensätze auf die Abfrage passen und den ersten Satz ermitteln. Zu diesem Zeitpunkt ist der erste Satz der einzige, der ermittelt wurde. Die restlichen Sätze der Ergebnistabelle bleiben unbekannt.

```
DECLARE hard CURSOR FOR
    SELECT C.customer_num, O.order_num, sum (items.total_price)
        FROM customer C, orders O, items I
        WHERE C.customer_num = O.customer_num
            AND O.order_num = I.order_num
            AND O.paid_date is null
        GROUP BY C.customer_num, O.order_num
```

Die Ergebnistabelle für diesen Cursor wurde dadurch erzeugt, daß drei Tabellen miteinander verbunden und die ermittelten Sätze in Gruppen zusammengefaßt wurden. Der Optimierer könnte auch Indizes verwenden, um die Sätze in der richtigen Reihenfolge zu liefern. Im allgemeinen macht es die Verwendung der Klauseln ORDER BY oder GROUP BY aber erforderlich, daß der Datenbankserver erst alle Sätze zusammenstellt, diese in eine temporäre

Tabelle kopiert und die Tabelle sortiert. Erst dann kann er wissen, welcher Satz zuerst angezeigt werden muß. Weitere Information zu dieser Thematik erhalten Sie im Kapitel 13.

In den Fällen, in denen die Ergebnistabelle vollständig in einer temporären Tabelle eingelesen und gesichert wird, kann sich der Datenbankserver viel Zeit lassen, um den Cursor zu öffnen. Anschließend kann er dem Programm genau sagen, wieviele Sätze die Ergebnistabelle enthält. Diese Information wird jedoch nicht verfügbar gemacht. Ein Grund hierfür ist, daß Sie sich nie sicher sein können, welche Methode der Optimierer verwendet. Wenn möglich, vermeidet er das Sortieren sowie temporäre Tabellen; aber geringfügige Änderungen in der Abfrage, in der Tabellengröße oder in den verfügbaren Indizes können ihn veranlassen, die Methode zu ändern.

### Die Ergebnistabelle eines sequentiellen Cursors

Der Datenbankserver will für die Ergebnistabelle des Cursors sowenig Betriebsmittel wie möglich festlegen. Wenn möglich, hält der Datenbankserver nie mehr als den einen Satz, der als nächster geholt werden wird. Bei einem sequentiellen Cursor ist dies meistens möglich. Mit jedem Holen liefert er den Inhalt des aktuellen Satzes und ermittelt den nächsten.

### Die Ergebnistabelle eines Scroll-Cursors

In der Ergebnistabelle eines Scroll-Cursors müssen alle Sätze solange behalten werden, bis der Cursor geschlossen wird. Dies ist erforderlich, weil der Datenbankserver nicht wissen kann, welchen Satz das Programm als nächsten anfordert.

Sehr oft legt der Datenbankserver die Ergebnistabelle eines Scroll-Cursors als temporäre Tabelle ab. Der Datenbankserver wird diese Tabelle jedoch nicht sofort auffüllen (außer, er erzeugt eine temporäre Tabelle, um die Abfrage auszuführen). Er erzeugt die temporäre Tabelle normalerweise beim Öffnen des Cursors. Wenn ein Datensatz das erste Mal geholt wird, kopiert der Datenbankserver ihn in die temporäre Tabelle und liefert ihn an das Programm. Wenn ein Satz zum zweiten Mal geholt wird, dann kann er aus der temporären Tabelle entnommen werden. Sollte das Programm die Abfrage beenden, bevor alle Sätze ermittelt wurden, so benötigt dieser Ablauf die entsprechend geringsten Betriebsmittel. Sätze, die nicht geholt wurden, müssen auch nicht erzeugt bzw. gesichert werden.

## Die Ergebnistabelle und Parallelbearbeitung

Wenn nur ein Programm mit einer Datenbank arbeitet, dann können sich die Elemente der Ergebnistabelle nicht verändern. Einige Programme müssen aber für die Anwendung in einem System mit mehreren Programmen geschrieben werden; dabei können zwei, drei oder Dutzende von Programmen zur gleichen Zeit mit den gleichen Tabellen arbeiten.

Wenn andere Programme Tabellen aktualisieren können, während Ihr Cursor geöffnet ist, scheint die Idee mit der Ergebnistabelle weniger sinnvoll zu sein. Ihr Programm kann immer nur je einen Satz verarbeiten, aber in der Tabelle können alle anderen Sätze verändert werden.

Im Fall einer einfachen Abfrage kann jeder beliebige Satz geändert werden, während der Datenbankserver einen bestimmten Satz der Ergebnistabelle hält. Nachdem Ihr Programm einen Satz geholt hat, kann ein anderes Programm sofort danach denselben Satz löschen oder verändern. Dieser Satz gehört nicht mehr zur Ergebnistabelle, wenn er noch einmal überprüft wird.

Wenn die Ergebnistabelle, oder ein Teil davon, in einer temporären Tabelle gespeichert wird, entsteht ein Problem mit *veralteten Daten*. Das heißt, daß sich die Sätze der aktuellen Tabelle verändern können. Aus dieser Tabelle werden die Sätze der Ergebnistabelle abgerufen. Wenn sich Sätze ändern, dann geben einige Sätze der Ergebnistabelle nicht mehr den aktuellen Inhalt der Tabelle wieder.

Diese Vorstellung mag auf den ersten Blick beunruhigend sein, aber solange Ihr Programm die Daten nur liest, gibt es keine veralteten Daten, oder genauer gesagt, alle Daten sind gleichermaßen veraltet. Es spielt keine Rolle, wann die Daten geholt werden, da die Ergebnistabelle immer ein Abbild der Daten zu einem bestimmten Zeitpunkt ist. Ein Datensatz ist am nächsten Tag bereits anders; es macht nichts, wenn er auch in der nächsten Tausendstelsekunde anders ist. Anders gesagt, es gibt keinen wirklichen Unterschied zwischen Änderungen, die während der Laufzeit des Programms erfolgen und Änderungen, die sofort nach Beendigung des Programms gesichert und verwendet werden.

Veraltete Daten können nur dann problematisch werden, wenn das Programm die Eingabedaten dazu verwendet, um damit die Datenbank zu verändern. Dies ist z. B. bei einer Bankanwendung der Fall, wo der Kontostand gelesen, verändert und zurückgeschrieben werden muß. Im nächsten Kapitel werden Programme erläutert, die Daten verändern.

# Verwendung eines Cursors: Das Stücklisten-Problem

Durch die Verwendung eines Cursors können Sie Probleme lösen, die einfaches SQL nicht lösen kann. Eines dieser Probleme ist die Verarbeitung von Stücklisten. Den Kern dieses Problems bildet eine rekursive Beziehung zwischen Objekten, so daß ein Objekt andere Objekte (die wiederum andere haben können) enthalten kann.

Diese Problem entsteht normalerweise bei der Bestandsaufnahme einer Produktion. Eine Firma stellt eine Vielzahl an Teilen her. Einige Teile stehen für sich, andere hingegen sind aus anderen Teilen zusammengebaut.

Diese Beziehungen werden in einer einzigen Tabelle festgehalten, die **contains** heißen könnte. In der Spalte **contains.parent** sind die Teile-Nummern derjenigen Teile enthalten, die zusammengebaut sind. In der Spalte **contains.child** wird die Teile-Nummer eines Teils gespeichert, die ein Bestandteil des Eltern-Teils ist. Wenn das Teil Nr. 123400 aus neun Teilen zusammengesetzt ist, dann gibt es neun Sätze, die in der ersten Spalte den Wert 123400 und in der zweiten Spalte andere Teile-Nummern enthalten.

CONTAINS

| PARENT | CHILD |
|---|---|
| FK NN | FK NN |
| 123400<br>432100 | 432100<br>765899 |

Das Problem bei Stücklisten ist, daß bei gegebener Teile-Nummer eine Liste all der Teile erstellt wird, die Bestandteile dieses Teils sind. In Bild 5-6 wird ein Lösungsentwurf mit **INFORMIX-4GL** gezeigt.

```
DEFINE part_list ARRAY[200] OF INTEGER
FUNCTION boom (top_part)
    DEFINE this_part, child_part INTEGER
    DEFINE next_to_do, next_free SMALLINT
    DECLARE part_scan CURSOR FOR
        SELECT child INTO child_part FROM contains
            WHERE parent = this_part

    LET next_to_do = 1
    LET part_list[next_to_do] = top_part
    LET next_free = 2
    WHILE next_to_do < next_free
        this_part = part_list[next_to_do]
        FOREACH part_scan
            LET part_list[next_free] = child_part
            LET next_free = next_free + 1
        END FOREACH
        LET next_to_do = next_to_do + 1
    END WHILE
    RETURN next_free - 1
END FUNCTION
```

*Bild 5-6* *Ein allererster Ansatz zur Lösung des Problems*

Aus technischer Sicht ist jeder Satz der Tabelle **contains** ein Knotenpunkt eines gerichteten azyklischen Graphen bzw. Baums. Die Funktion in Bild 5-6 führt eine allererste Durchsuchung eines Baums durch, wobei der Parameter Teile-Nummer die Wurzel des Baums bildet. Diese Funktion verwendet einen Cursor namens **part_scan**, um all die Sätze zu ermitteln, die in der Spalte **parent** einen bestimmten Wert aufweisen. Mit der **INFORMIX-4GL**-Anweisung FOREACH ist dies sehr einfach durchzuführen. Diese Anweisung öffnet den Cursor, wiederholt die Funktion einmal für jeden Satz der Auswahlliste und schließt den Cursor.

Dies ist der Kern des Stücklisten-Problems, aber es ist auf keinen Fall eine vollständige Lösung. Das Programm in Bild 5-6 läßt es z. B. nicht zu, daß Bestandteile auf mehr als einer Ebene des Baums vorkommen. Weiterhin würde eine reale Tabelle **contains** auch über eine Spalte **count** verfügen; diese enthält die Anzahl der **Kind**-Teile, aus denen jedes **Eltern**-Teil zusammengesetzt ist. Ein Programm, das die gesamte Anzahl aller Bestandteile zurückliefert, ist etwas komplizierter.

Um an das Stücklisten-Problem heranzugehen, gibt es nicht nur den bereits beschriebenen Weg über Wiederholungen. Wenn die Anzahl der Ebenen begrenzt ist, dann können Sie das Problem auch mit einer einzigen SELECT-Anweisung lösen, indem Sie geschachtelte Outer-Self-Joins verwenden.

Wenn ein Eltern-Teil aus bis zu vier Ebenen bestehen kann, dann liefert die folgende SELECT-Anweisung all diese zurück:

```
SELECT a.parent, a.child, b.child, c.child, d.child
    FROM contains a
        OUTER (contains b,
            OUTER (contains c, outer contains d))
    WHERE a.parent = top_part_number
        AND a.child = b.parent
        AND b.child = c.parent
        AND c.child = d.parent
```

Diese SELECT-Anweisung liefert pro Ebene des absteigenden Wurzelgebildes je einen Satz; die **top_part_number** gibt hierbei das jeweilige Teil an. Bei Ebenen, die nicht vorhanden sind, werden NULL-Werte zurückgegeben. (Verwenden Sie Indikator-Variablen, um diese zu ermitteln.) Sie können diese Lösung auf mehrere Ebenen erweitern, indem Sie auf die Tabelle **contains** zusätzliche geschachtelte Outer-Joins anwenden. Sie können diese Lösung auch so überarbeiten, daß pro Ebene die jeweilige Anzahl der Teile-Nummern ermittelt wird.

# Dynamisches SQL

Obwohl statisches SQL sehr nützlich ist, erzwingt es andererseits, daß Sie bei der Programmerstellung den genauen Inhalt einer SQL-Anweisung kennen. Sie müssen beispielsweise genau angeben, welche Spalten in einer WHERE-Klausel überprüft werden sollen und welche Spalten in einer Auswahlliste angegeben werden sollen.

Wenn Sie ein Programm schreiben, das eine bestimmte, klar definierte Aufgabe durchführt, ist dies kein Problem. Aber bei einigen Programmen können die Anforderungen an die Datenbank nicht genau im voraus definiert werden. Ein Programm, das Benutzeranfragen interaktiv beantwortet, muß SQL-Anweisungen in Abhängigkeit einer Benutzereingabe erstellen können.

*Dynamisches* SQL ermöglicht einem Programm, eine SQL-Anweisung während der Ausführung zusammenzustellen, so daß die Benutzereingabe den Inhalt der Anweisung bestimmt. Dies wird in drei Stufen durchgeführt:

1. Das Programm stellt den Text einer SQL-Anweisung als eine Zeichenkette zusammen; diese wird in einer Programm-Variablen gespeichert.
2. Das Programm führt die Anweisung PREPARE aus; diese veranlaßt den Datenbankserver, den Anweisungstext zu überprüfen und die Anweisung zur Ausführung aufzubereiten.

3. Das Programm verwendet die Anweisung EXECUTE, um die aufbereitete Anweisung auszuführen.

Auf diese Weise kann ein Programm jede beliebige SQL-Anweisung erstellen und dann verwenden; hierbei kann jede beliebige Benutzereingabe verwendet werden. Beispielsweise kann ein Programm eine Datei, die SQL-Anweisungen enthält, lesen, aufbereiten und ausführen.

**DB-Access** ist ein **INFORMIX-ESQL/C**-Programm, das SQL-Anweisungen dynamisch erstellt, aufbereitet und ausführt; **DB-Access** ist das Dienstprogramm, das Sie zum interaktiven Arbeiten mit SQL verwenden können. Beispielsweise ermöglicht es **DB-Access** den Benutzern, mit Hilfe eines einfachen, interaktiven Menüs die Spalten einer Tabelle festzulegen. Wenn der Benutzer damit fertig ist, erzeugt **DB-Access** dynamisch die erforderliche CREATE TABLE- oder ALTER TABLE-Anweisung, bereitet sie auf und führt sie aus.

# Eine Anweisung aufbereiten

Von der Form her ist eine dynamische SQL-Anweisung wie jede andere SQL-Anweisung, die in ein Programm geschrieben wurde; allerdings darf sie keine Namen von Host-Variablen enthalten.

Dies führt zu zwei Einschränkungen. Erstens darf eine SELECT-Anweisung keine INTO-Klausel enthalten. Diese Klausel benennt Host-Variablen zur Aufnahme der Spaltenwerte. Host-Variablen in einer dynamischen Anweisung sind nicht erlaubt. Zweitens wird an jeder Stelle, an der in einem Ausdruck normalerweise der Name einer Host-Variablen stehen würde, ein Fragezeichen als Platzhalter geschrieben.

Eine Anweisung dieser Art können Sie mit der Anweisung PREPARE für die Ausführung aufbereiten. Hier ein Beispiel in **INFORMIX-ESQL/C**:

```
exec sql prepare query_2 from
        'select * from orders
            where customer_num = ? and
                order_date > ?';
```

In diesem Beispiel sind zwei Fragezeichen enthalten. Sie zeigen an, daß bei der Ausführung der Anweisung an diesen zwei Stellen die Werte von Host-Variablen verwendet werden.

Sie können fast jede SQL-Anweisung dynamisch aufbereiten. Die einzigen, die nicht aufbereitet werden können, sind Anweisungen für dynamisches SQL selbst, sowie Anweisungen zur Cursorverwaltung wie PREPARE und

OPEN. Nachdem Sie eine UPDATE- oder DELETE-Anweisung aufbereitet haben, sollten Sie das fünfte Element von SQLAWARN darauf hinprüfen, ob eine WHERE-Klausel verwendet wurde (siehe "Der Array SQLAWARN" auf Seite 5-12).

Das Ergebnis einer aufbereiteten Anweisung ist eine Datenstruktur, die die Anweisung repräsentiert. Diese Datenstruktur stimmt nicht mit der Zeichenkette überein, aus der die Anweisung erstellt wurde. In der PREPARE-Anweisung geben Sie den Namen der Datenstruktur an. Im obigen Beispiel lautet der Name **query_2**. Dieser Name wird für die Ausführung der aufbereiteten SQL-Anweisung verwendet.

Bei einer PREPARE-Anweisung gibt es keine Begrenzung für die Zeichenkettenlänge einer Anweisung. Eine Anweisung kann mehrere SQL-Anweisungen enthalten, die mit einem Semikolon voneinander getrennt sind. Bild 5-7 zeigt ein ziemlich komplexes Beispiel in **INFORMIX-ESQL/COBOL**.

```
    MOVE     'BEGIN WORK;
             UPDATE account
                 SET balance = balance + ?
                     WHERE acct_number = ?;
             UPDATE teller
                 SET balance = balance + ?
                     WHERE teller_number = ?;
             UPDATE branch
                 SET balance = balance + ?
                     WHERE branch_number = ?;
             INSERT INTO history VALUES(timestamp, values);

    TO BIG-QUERY.

EXEC SQL
    PREPARE BIG-Q FROM :BIG-QUERY
END-EXEC.
```

*Bild 5-7* ***Eine aufbereitete Zeichenkette , die fünf SQL-Anweisungen enthält***

Bei der Ausführung dieser Anweisungsfolge müssen Host-Variablen die Werte für sechs Platzhalter bereitstellen. Obwohl die Erstellung einer Folge mehrerer Anweisungen komplizierter ist, ist die Performance meistens besser. Dies liegt daran, daß der Dialog zwischen dem Programm und dem Datenbankserver geringer wird.

# Eine aufbereitete SQL-Anweisung ausführen

Ist eine Anweisung einmal aufbereitet, kann sie beliebig oft ausgeführt werden. Anweisungen, die keine SELECT-Anweisungen sind, und SELECT-Anweisungen, die nur einen Satz liefern, werden mit der Anweisung EXECUTE ausgeführt.

Bild 5-8 zeigt, wie **INFORMIX-ESQL/C** zur Aktualisierung eines Bankkontos eine Anweisungsfolge aufbereitet und ausführt:

```
exec sql begin declare section;
char bigquery[270] = 'begin work;';
exec sql end declare section;
stcat ('update account set balance = balance + ? where ', bigquery);
stcat ('acct_number = ?;', bigquery);
stcat ('update teller set balance = balance + ? where ', bigquery);
stcat ('teller_number = ?;', bigquery);
stcat ('update branch set balance = balance + ? where ', bigquery);
stcat ('branch_number = ?;', bigquery);
stcat ('insert into history values(timestamp, values);', bigquery);

exec sql prepare bigq from :bigquery;

exec sql execute bigq using :delta, :acct_number, :delta,
    :teller_number, :delta, :branch_number;

exec sql commit work;
```

*Bild 5-8* ***Eine aufbereitete und ausgeführte Anweisungsfolge mit* ESQL/C**

Die USING-Klausel der EXECUTE-Anweisung enthält eine Reihe von Host-Variablen; deren Werte werden in der aufbereiteten Anweisung anstelle der Fragezeichen verwendet.

## Aufbereitete SELECT-Anweisungen verwenden

Eine dynamisch aufbereitete SELECT-Anweisung kann nicht einfach ausgeführt werden; da sie mehr als einen Satz liefern könnte, und da der Datenbankserver nicht weiß, welchen von diesen er zurückgeben soll, liefert er eine Fehlermeldung. Statt dessen ist eine dynamische SELECT-Anweisung mit einem Cursor zu verbinden. Dieser Cursor wird normal geöffnet und verwendet. Ein Cursor zu einer aufbereiteten Anweisung wird mit dem Namen der Anweisung deklariert.

Ein **INFORMIX-4GL**-Beispiel zeigt dies:

```
LET select_2 = 'select order_num, order_date from orders ',
    'where customer_num = ? and order_date > ?'

PREPARE q_orders FROM select_2

DECLARE cu_orders CURSOR FOR q_orders

OPEN cu_orders USING q_c_number, q_o_date

FETCH cu_orders INTO f_o_num, f_o_date
```

Die folgende Liste zeigt die Verarbeitungsphasen dieses Beispiels:

1. Einer Programm-Variablen wird eine Zeichenkette zugewiesen, die eine SELECT-Anweisung darstellt. Diese enthält zwei Fragezeichen als Platzhalter.
2. Die PREPARE-Anweisung wandelt die Zeichenkette in eine ausführbare Zeichenkette um. Der Zeichenkette wird der Name **q_orders** zugeordnet.
3. Ein Cursor mit dem Namen **cu_orders** wird deklariert; ihm wird der Name der aufbereiteten Anweisung zugeordnet.
4. Wenn der Cursor geöffnet ist, wird die aufbereitete Anweisung ausgeführt. Die USING-Klausel der OPEN-Anweisung stellt zwei Host-Variablen zur Verfügung; deren Inhalt wird in der Originalanweisung anstelle der Fragezeichen eingesetzt.
5. Mit dem geöffneten Cursor wird der erste Datensatz geholt. Die INTO-Klausel der FETCH-Anweisung gibt die Host-Variablen an, die die geholten Werte aufnehmen sollen.

Der Cursor kann später geschlossen und nochmals geöffnet werden. Solange der Cursor geschlossen ist, kann unter dem Namen **q_orders** eine andere SELECT-Anweisung aufbereitet werden. Auf diese Weise kann ein einziger Cursor dazu verwendet werden, Daten über verschiedene SELECT-Anweisungen zu holen.

# Dynamische Host-Variablen

Bei Produkten mit eingebetteter Sprache, die eine dynamische Zuweisung von Datenobjekten unterstützen, ist es möglich, dynamische Anweisungen auf einer höheren Ebene einzusetzen. Es ist möglich, Host-Variablen, die Daten aufnehmen, dynamisch zuzuteilen. Dies erlaubt es, eine beliebige SELECT-Anweisung aus der Eingabe an das Programm zu übernehmen, die Anzahl der Werte und die Datentypen zu bestimmen und den aufnehmenden Host-Variablen die entsprechenden Datentypen zuzuweisen.

In der Anweisung DESCRIBE steckt der Schlüssel hierfür. Sie nimmt den Namen einer aufbereiteten SQL-Anweisung auf und liefert Daten über die Anweisung und deren Inhalt zurück. Sie setzt SQLCODE, um den Anweisungstyp zu bestimmen; dies geschieht über das Schlüsselwort, mit dem die Anweisung beginnt. Wenn die aufbereitete Anweisung eine SELECT- oder eine INSERT-Anweisung ist, dann liefert die DESCRIBE-Anweisung auch Informationen über die ausgewählten oder eingefügten Werte an eine Datenstruktur zurück. Die Datenstruktur ist für diesen Zweck vordefiniert und wird als Deskriptorbereich bezeichnet. Unter **INFORMIX-ESQL/C** können Sie einen Deskriptorbereich verwenden oder (alternativ) eine **sqlda** Zeigerstruktur.

Die Datenstruktur, die die DESCRIBE-Anweisung für eine SELECT-Anweisung zurückliefert oder auf die sie verweist, enthält einen Array von Strukturen. Jede Struktur beschreibt die Daten, die pro Element der Auswahlliste zurückgegeben werden. Das Programm kann den Array untersuchen und dabei herausfinden, ob ein Satz einen Dezimalwert, einen CHAR-Wert von einer bestimmten Länge oder einen ganzzahligen Wert enthält.

Unter Verwendung dieser Informationen kann das Programm Speicherplatz zuweisen, um die abgerufenen Werte zu speichern und auch die Zeiger in die Datenstruktur setzen, die der Datenbankserver benötigt.

## Speicherplatz für aufbereitete Anweisungen freigeben

Eine aufbereitete SQL-Anweisung belegt Speicherplatz. Bei einigen Datenbankservern kann sie den Speicherplatz des Datenbankservers sowie den des Programms verwenden. Der Speicherplatz des Programms wird bei Beendigung des Programms automatisch wieder freigegeben, aber Sie sollten den Speicher freigeben, sobald Sie ihn nicht mehr benötigen.

Mit der Anweisung FREE können Sie diesen Speicherplatz freigeben. Diese Anweisung benötigt entweder den Namen der Anweisung oder den Namen des Cursors, der für den Anweisungsnamen deklariert wurde. Die Anweisung gibt den Speicherplatz frei, der einer aufbereiteten Anweisung zugewiesen wurde. Wenn in einer Anweisung mehr als ein Cursor deklariert wurde, führt die Freigabe der Anweisung nicht automatisch zu einer Freigabe der Cursor.

## Schnelle Ausführung

Bei einfachen Anweisungen, die weder einen Cursor noch Host-Variablen benötigen, können Sie die Aktionen der Anweisungen PREPARE, EXECUTE und FREE zu einem Schritt zusammenfassen. Die Anweisung EXECUTE IMMEDIATE erfordert eine Zeichenkette. Die Anweisung bereitet in einem Schritt die Zeichenkette vor, führt sie aus und gibt den Speicherplatz frei.

```
EXEC SQL EXECUTE IMMEDIATE 'drop index my_temp_index';
```

Dies vereinfacht die Erstellung einfacher SQL-Anwendungen. Die Anweisung EXECUTE IMMEDIATE kann jedoch nicht bei SELECT-Anweisungen verwendet werden, da die USING-Klausel nicht erlaubt ist.

# Datendefinitionsanweisungen einbetten

Unter Datendefinitionsanweisungen sind SQL-Anweisungen, die Datenbanken erzeugen und die Definitionen der Tabellen verändern. Die Anweisungen zur Datendefinition sind normalerweise nicht im Programm enthalten. Der Grund liegt darin, daß diese selten durchgeführt werden – eine Datenbank wird genau einmal erstellt, aber sie wird oftmals abgefragt und aktualisiert.

Das Erstellen einer Datenbank und der Tabellen wird im allgemeinen interaktiv durchgeführt; hierbei wird **DB-Access** oder **INFORMIX-SQL** verwendet.

# Berechtigungen in Programmen vergeben und entziehen

Eine bestimmte Aufgabe, die mit der Datendefinition zusammenhängt, wird immer wieder ausgeführt: das Vergeben und Entziehen von Berechtigungen. In Kapitel 11 wird erläutert, warum dies wiederholt getan wird. Da Berechtigungen häufig vergeben und entzogen werden müssen und da die damit beauftragten Personen vielleicht über wenig SQL-Kenntnisse verfügen, kann es sinnvoll sein, die GRANT- und REVOKE-Anweisungen in Programme einzubinden. Auf diese Weise kann man den Benutzern eine einfachere, praktischere Schnittstelle für diese Anweisungen zur Verfügung stellen.

Die Anweisungen GRANT und REVOKE sind für dynamisches SQL besonders gut geeignet. Jede Anweisung benötigt drei Parameter:

- Eine Liste mit einer oder mehreren Berechtigung(en)
- Einen Tabellennamen
- Den Benutzernamen

Sie werden wahrscheinlich zumindest einige dieser Werte an Programme übergeben müssen (vom Benutzer, von Parametern aus der Kommandozeile oder aus einer Datei). Es kann aber keiner dieser Werte in Form einer Host-Variablen zur Verfügung gestellt werden. Die Syntax dieser Anweisungen lassen an keiner Stelle Host-Variablen zu.

Die einzige Möglichkeit wäre, die Teile der Anweisung zu einer Zeichenkette zusammenzufassen und dann die zusammengefaßte Anweisung aufzubereiten und auszuführen. Die Eingaben kann man als CHAR-Werte an das Programm zur Weiterleitung an die aufbereitete Anweisung übergeben.

Bild 5-9 zeigt eine **INFORMIX-4GL**-Funktion, die aus den Funktions-Parametern eine GRANT-Anweisung zusammensetzt und diese anschließend aufbereitet und ausführt.

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
    DEFINE  priv_to_grant char(100),
            table_name char(20),
            user_id char(20),
            grant_stmt char(200)
    LET grant_stmt =' GRANT ', priv_to_grant,
                    ' ON ', table_name,
                    ' TO ', user_id
    WHENEVER ERROR CONTINUE
    PREPARE the_grant FROM grant_stmt
    IF status = 0 THEN
        EXECUTE the_grant
    END IF
    IF status <> 0 THEN
     DISPLAY 'Sorry, got error #', status, 'attempting:'
        DISPLAY '      ',grant_stmt
    END IF
    FREE the_grant
    WHENEVER ERROR STOP
END FUNCTION
```

*Bild 5-9* ***Eine 4GL-Funktion, die eine GRANT-Anweisung erstellt, aufbereitet und ausführt***

Die erste Anweisung definiert den Namen der Funktion und die Namen der drei Parameter.

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
```

Die DEFINE-Anweisung definiert die Parameter und eine weitere lokale Variable für die Funktion. Alle vier Zeichenketten haben eine variable Länge.

```
DEFINE  priv_to_grant char(100),
        table_name char(20),
        user_id char(20),
        grant_stmt char(200)
```

Die Variable **grant_stmt** nimmt die zusammengesetzte GRANT-Anweisung auf; diese Anweisung wurde durch die Verkettung der Parameter und einiger Konstanten erstellt.

```
LET grant_stmt ='GRANT ', priv_to_grant,
                ' ON ', table_name,
                ' TO ', user_id
```

Bei **INFORMIX-4GL** wird zur Verkettung von Zeichenketten ein Komma verwendet. Die Zuweisungs-Anweisung verkettet die folgenden sechs Zeichenketten:

- 'GRANT'
- Den Parameter, der die zu vergebenden Berechtigungen angibt
- 'ON'
- Den Parameter, der den Tabellennamen angibt
- 'TO'
- Den Parameter, der den Benutzernamen angibt.

Das Ergebnis ist eine vollständige GRANT-Anweisung, die teilweise aus Eingaben an das Programm erstellt wurde. Dasselbe Ergebnis kann bei anderen Host-Sprachen unter Verwendung einer anderen Syntax erreicht werden.

```
WHENEVER ERROR CONTINUE
PREPARE the_grant FROM grant_stmt
```

Wenn der Datenbankserver an SQLCODE eine Fehlernummer liefert, dann beendet **INFORMIX-4GL** standardmäßig das Programm. Wenn Sie SQL-Anweisungen aufbereiten, die Benutzereingaben erfordern, dann werden wahrscheinlich Fehler auftreten. Um Fehler zu diagnostizieren, ist die Beendigung des Programms jedoch nicht hilfreich. In der vorherigen Anweisung verhinderte die WHENEVER-Anweisung die Beendigung. Anschließend leitet die PREPARE-Anweisung den zusammengesetzten Anweisungstext zur Analyse an den Datenbankserver weiter.

Wenn der Datenbankserver die Anweisung akzeptiert, dann liefert er den Returnwert Null. Dies heißt aber nicht, daß die Anweisung richtig ausgeführt wird; es bedeutet nur, daß die Syntax der Anweisung richtig ist. Die Anweisung kann sich auf eine nicht vorhandene Tabelle beziehen oder sie kann andere Fehlerquellen beinhalten, die erst während der Ausführung entdeckt werden können.

```
IF status = 0 THEN
    EXECUTE the_grant
END IF
```

Wenn die Aufbereitung erfolgreich war, dann wird als nächstes die aufbereitete Anweisung ausgeführt. Der restliche Teil der Funktion aus Bild 5-9 ist im Fehlerfall für die Ausgabe einer Fehlermeldung zuständig. Wie bereits gesagt, besteht zwischen einem Fehler bei einer PREPARE-Anweisung und einem Fehler bei einer EXECUTE-Anweisung kein Unterschied. Die Funktion versucht nicht, den numerischen Fehlercode zu interpretieren, sondern sie überläßt die Interpretation dem Benutzer.

# Zusammenfassung

SQL-Anweisungen können in Programm so eingefügt werden, als wären sie normale Anweisungen der Programmiersprache. In der WHERE-Klausel können Programm-Variablen verwendet werden, in die Daten der Datenbank eingelesen werden können. Der SQL-Code wird von einem Präprozessor in Prozeduraufrufe und Datenstrukturen übersetzt.

Anweisungen, die keine Daten liefern oder Abfragen, die genau einen Datensatz liefern, werden wie normale, auszuführende Anweisungen der Sprache behandelt. Abfragen, die mehr als einen Satz liefern können, werden mit einem Cursor verbunden, der auf den jeweils aktuellen Satz zeigt. Mit Hilfe eines Cursors kann das Programm bei Bedarf jeden Satz holen.

Statische SQL-Anweisungen sind im Programmtext enthalten, das Programm kann jedoch während der Laufzeit auch neue Anweisungen dynamisch zusammenstellen und diese ausführen. In den meisten Fällen kann das Programm Informationen über die Anzahl und die Typen der Spalten erhalten, die ein Abfrage geliefert hat, und kann auch dynamisch Speicherplatz zuteilen, um diese Informationen aufzunehmen.

# Programme zur Veränderung von Daten



6

# Kapitelüberblick

Das vorangegangene Kapitel stellte ein Konzept vor, wie man SQL-Anweisungen, vor allem SELECT-Anweisungen, in Programme einfügt, die in anderen Programmiersprachen geschrieben sind.

Dieses Kapitel erläutert, welche Probleme entstehen können, wenn ein Programm die Datenbank verändert, d. h. wenn es Sätze einfügtn, löscht oder aktualisiert. Ziel dieses Kapitels ist es, Ihnen die Informationen zu vermitteln, die Sie zum Lesen des Handbuchs des von Ihnen verwendeten Informix **ESQL**- oder **4GL**-Produkts benötigen.

Die allgemeine Verwendung der Anweisungen INSERT, UPDATE und DELETE ist im Kapitel 4 dieses Handbuchs beschrieben. Es ist ziemlich einfach, die Anweisungen in ein Programm einzubauen, aber es kann sehr schwierig sein, Fehler zu behandeln und sich mit parallelen Datenänderungen zu befassen, die von mehreren Programmen aus erfolgen.

# Die DELETE-Anweisung

Ein Programm löscht Sätze aus einer Tabelle, indem es eine DELETE-Anweisung ausführt. Die DELETE-Anweisung kann Datensätze auf übliche Weise mit einer WHERE-Klausel spezifizieren. Sie kann sich aber auch auf einen einzelnen Satz beziehen, der als letzter mit einem Cursor geholt wurde.

Bei jedem Löschen von Datensätzen muß man überlegen, ob in anderen Tabellen Sätze von diesen gelöschten Sätzen abhängig sind. Das Problem der abhängigen Löschungen wurde bereits in Kapitel 4 erörtert; dasselbe Problem tritt auf, wenn die Löschung von einem Programm aus erfolgt.

# Direktes Löschen

Sie können eine DELETE-Anweisung in ein Programm einbetten.Das folgende Beispiel zeigt dies unter Verwendung von **INFORMIX-ESQL/C**:

```
exec sql delete from items
    where order_num = :onum;
```

Sie können eine Anweisung derselben Form auch dynamisch aufbereiten und ausführen.

In der WHERE-Klausel dieses Beispiels wird der Wert der Host-Variablen **onum** verwendet. Nach der Operation werden die Ergebnisse in SQLSTATE und SQLCA eingetragen. Das dritte Element des Arrays SQLERRD enthält die Anzahl der gelöschten Sätze. Das gilt auch dann, wenn ein Fehler aufgetreten ist. Der Wert in SQLCODE zeigt das gesamte Ergebnis der Operation an. Wenn dieser Wert nicht negativ ist, dann sind keine Fehler aufgetreten. In diesem Fall enthält das dritte Element von SQLERRD die Anzahl aller Sätze, die der WHERE-Klausel genügten und die gelöscht wurden.

## Fehler beim direkten Löschen

Beim Auftreten eines Fehlers wird die Anweisung vorzeitig beendet. Die Werte in SQLSTATE und in SQLCODE sowie das zweite Element von SQLERRD erklären die Ursache des Fehlers. Mit der Anzahl der Sätze wird angezeigt, wieviele Sätze gelöscht wurden. In vielen Fehlerfällen ist die Anzahl Null (0), da diese Fehler den Datenbankserver daran hindern, die Ausführung überhaupt zu beginnen. Wenn z. B. eine Tabelle nicht vorhanden ist oder wenn eine Spalte, die in der WHERE-Klausel überprüft wird, umbenannt wurde, dann wird das Löschen gar nicht erst versucht.

Bestimmte Fehler können jedoch erst entdeckt werden, nachdem die Operation begonnen wurde und einige Sätze verarbeitet wurden. Der Fehler, der am häufigsten vorkommt, ist ein Konflikt beim Sperren. Bevor der Datenbankserver einen Satz löschen kann, muß er diesen exklusiv sperren. Es könnten aber auch andere Programme diesen Satz verwenden und dabei verhindern, daß der Datenbankserver diesen Satz sperrt. Da das Thema Sperrmechanismus alle Arten der Veränderung betrifft, wird es in diesem Kapitel in einem eigenen Abschnitt behandelt.

Andere, seltenere Fehler können auftreten, nachdem das Löschen begonnen wurde; beispielsweise könnte ein Hardware-Fehler während einer Aktualisierung der Datenbank auftreten.

## Die Verwendung von Transaktionsprotokollen

Die Verwendung von Transaktionsprotokollen ist die beste Art, sich auf jeden möglichen Fehler vorzubereiten, der während einer Änderung auftreten kann. Im Fehlerfall können Sie den Datenbankserver anweisen, die Datenbank auf den letzten Stand zurückzusetzen. Im folgenden Beispiel wurde das vorherige Beispiel um die Verwendung von Transaktionen erweitert:

```
exec sql begin work;                /* start the transaction*/
exec sql delete from items
      where order_num = $onum;
del_result = sqlca.sqlcode;         /* save two error */
del_isamno = sqlca.sqlerrd[1];      /* ...code numbers */
del_rowcnt = sqlca.sqlerrd[2];      /* ...and count of rows */
if (del_result < 0)                 /* some problem, */
    exec sql rollback work;         /* ...put everything back */
else                                /* everything worked OK, */
    exec sql commit work;           /* ...finish transaction */
```

Ein wichtiger Punkt dieses Beispiels ist, daß das Programm vor Beendigung der Transaktion die wichtigen Rückgabewerte des SQLCA speichert. Der Grund dafür ist, daß sowohl die ROLLBACK WORK- als auch die COMMIT WORK-Anweisung, wie alle SQL-Anweisungen, Rückgabewerte in SQLCA setzen. Wird nach einem Fehler eine ROLLBACK WORK-Anweisung ausgeführt, dann wird die Fehlernummer überschrieben; wenn diese Fehlernummer nicht gesichert wird, dann kann sie dem Benutzer auch nicht mitgeteilt werden.

Der Vorteil bei der Verwendung von Transaktionsprotokollen liegt darin, daß die Datenbank, ganz egal, was passiert, in einem bekannten, vorhersehbaren Zustand hinterlassen wird. Es taucht nie die Frage auf, wieviele der Änderungen abgeschlossen wurden. Es werden entweder alle Änderungen oder keine durchgeführt.

## Koordiniertes Löschen

Der Nutzen von Transaktionsprotokollen wird vor allem dann klar, wenn man mehr als eine Tabelle verändern muß. Betrachten wir z. B. das Problem, einen Auftrag aus der Beispiel-Datenbank zu löschen. Reduziert man dieses Problem auf das Wesentliche, dann muß man Sätze aus zwei Tabellen, **orders** und **items**, löschen; dies wird in dem **INFORMIX-4GL**-Beispiel in Bild 6-1 gezeigt.

```
WHENEVER ERROR CONTINUE{do not terminate on error}
BEGIN WORK  {start transaction}
DELETE FROM items
    WHERE order_num = o_num
IF (status >= 0) THEN{no error on first delete}
    DELETE FROM orders
        WHERE order_num = o_num
END IF
IF (status >= 0) THEN{no error on either delete}
    COMMIT WORK
ELSE        {problem on some delete}
    DISPLAY 'Error ', status, ' deleting.'
    ROLLBACK WORK
END IF
```

*Bild 6-1* ***Ein Teil einer 4GL-Anwendung; aus zwei Tabellen löschen***

Die Verwendung von Transaktionsprotokollen verändert die Programmlogik kaum. Wenn aber keine Transaktionsprotokolle verwendet werden, dann muß derjenige Benutzer, der die Fehlermeldung gesehen hat, eine Reihe sehr schwieriger Entscheidungen treffen. Abhängig davon, wann die Fehler auftraten, ergeben sich folgende Situationen:

- Es wurden keine Datensätze gelöscht, alle Datensätze mit dieser Auftragsnummer bleiben in der Datenbank.
- Einige, aber nicht alle Datensätze der Tabelle **items** wurden gelöscht; es bleibt ein Auftrags-Satz mit nur einigen Positionen (*items*) übrig.
- Es wurden alle Artikel-Sätze gelöscht, aber der Auftrags-Satz bleibt übrig.
- Es wurden alle Sätze gelöscht.

Im zweiten und dritten Fall ist die Datenbank bis zu einem gewissen Grad beschädigt. Sie enthält teilweise Daten, die bei einigen Abfragen falsche Antworten ergeben könnten. Um die Konsistenz der Datenbank wiederherzustellen, muß der Benutzer vorsichtig vorgehen. Durch die Verwendung von Transaktionsprotokollen können all diese Ungewißheiten verhindert werden.

## Mit einem Cursor löschen

Man kann in einer DELETE-Anweisung auch einen Cursor angeben, um den zuletzt geholten Satz zu löschen. Auf diese Weise kann man Löschungen programmieren, die auf Bedingungen basieren, die in einer WHERE-Klausel nicht abgeprüft werden können. Bild 6-2 zeigt hierzu ein Beispiel.

```
int delDupOrder()
{
    int ord_num;
    int dup_cnt, ret_code;
    exec sql declare scan_ord cursor for
        select order_num, order_date
            into $ord_num
            from orders for update;
    exec sql open scan_ord;
    if (sqlca.sqlcode != 0) return (sqlca.sqlcode);
    exec sql begin work;
    for(;;)
    {
        exec sql fetch next scan_ord;
        if (sqlca.sqlcode != 0) break;
        dup_cnt = 0; /* default in case of error */
        exec sql select count(*) into dup_cnt from orders
            where order_num = $ord_num;
        if (dup_cnt > 1)
        {
            exec sql delete where current of scan_ord;
            if (sqlca.sqlcode != 0)
                break;
        }
    }
    ret_code = sqlca.sqlcode;
    if (ret_code == 100)/* merely end of data */
    exec sql commit work;
    else    /* error on fetch or on delete */
    exec sql rollback work;
    return (ret_code);
}
```

*Bild 6-2* ***Eine gefährliche ESQL/C-Funktion, die mittels eines Cursors löscht (siehe Hinweis)***

***Hinweis:*** *Der Entwurf der* **ESQL/C***-Funktion ist gefährlich. Die Funktion hängt von der richtigen Ausführung auf der aktuellen "Isolationsstufe" ab (dies wird in diesem Kapitel später erläutert). Selbst wenn die Funktion korrekt arbeitet, ist die Art, wie sie arbeitet, von der physikalischen Reihenfolge der Sätze in der Tabelle abhängig; dieses Vorgehen sollte man in der Regel vermeiden.*

Der Zweck dieser Funktion ist es, diejenigen Sätze zu löschen, die doppelt vorkommende Auftragsnummern enthalten. Da in der Beispiel-Datenbank die Spalte **orders.order_num** aber über einen UNIQUE-Index verfügt, können in ihr keine doppelten Sätze enthalten sein. Für eine andere Datenbank kann man jedoch eine ähnliche Funktion schreiben.

Die Funktion deklariert den Cursor **scan_ord**, der alle Sätze der Tabelle **orders** absucht. Er ist mit der FOR UPDATE-Klausel deklariert; dies zeigt an, daß der Cursor für die Veränderung von Daten verwendet werden kann. Wenn der Cursor richtig geöffnet wird, startet die Funktion eine Transaktion und liest die Sätze einer Tabelle in einer Schleife. Die Funktion verwendet für jeden Satz eine eingebettete SELECT-Anweisung, um die Anzahl derjenigen Sätze der Tabelle zu bestimmen, deren Auftragsnummer gleich der Auftragsnummer des aktuellen Satzes ist (ohne die richtige Isolationsstufe führt dieser Schritt zu einem Fehler; dies wird in einem späteren Abschnitt beschrieben).

Bei der Beispiel-Datenbank, in der diese Tabelle über einen UNIQUE-Index verfügt, ist die an **dup_cnt** zurückgegebene Anzahl immer 1. Wenn der Wert jedoch größer ist, dann löscht die Funktion den aktuellen Satz aus der Tabelle und verringert somit die Anzahl der doppelten Sätze um 1.

Lösch-Funktionen dieser Art werden manchmal benötigt, aber der Entwurf muß im allgemeinen besser durchdacht sein. Diese Funktion löscht alle doppelt vorkommenden Sätze mit Ausnahme des letzten Satzes, der vom Datenbankserver geliefert wird. Diese Reihenfolge hat nichts mit dem Inhalt der Sätze oder deren Bedeutung zu tun. Man könnte vielleicht daran denken, die Funktion in Bild 6-2 zu verbessern, indem man bei der Deklaration des Cursors eine ORDER BY-Klausel hinzufügt. Die ORDER BY- und FOR UPDATE-Klausel kann man aber nicht gemeinsam verwenden. Weiter unten in diesem Kapitel wird ein besserer Ansatz gezeigt.

# Die INSERT-Anweisung

Man kann die INSERT-Anweisung in Programme einbetten. In Programmen wird diese Anweisung so gestaltet und verwendet, wie es in Kapitel 4 dieses Handbuchs beschrieben ist. Jedoch kann man hier bei den Klauseln VALUES und WHERE zusätzlich Host-Variablen in Ausdrücken verwenden. Darüberhinaus verfügt ein Programm über die Möglichkeit, Sätze unter Verwendung eines Cursors einzufügen.

# Die Verwendung eines Insert-Cursors

Es gibt viele Gestaltungsmöglichkeiten der DECLARE CURSOR-Anweisung. Sie wird meistens verwendet, um Cursor für unterschiedliche Arten der Datensuche zu erzeugen. Man kann aber auch einen speziellen Cursor erzeugen, einen *Insert-Cursor.* Einen Insert-Cursor verwendet man zusammen mit den Anweisungen PUT und FLUSH, um viele Datensätze effektiv in eine Tabelle einzufügen.

## Einen Insert-Cursor deklarieren

Einen Insert-Cursor erzeugt man, indem man einen Cursor für eine INSERT-Anweisung (FOR INSERT) anstatt für eine SELECT-Anweisung deklariert. Einen solchen Cursor kann man nicht zum Holen, sondern nur zum Einfügen von Sätzen verwenden. Das Beispiel in Bild 6-3 zeigt die Deklaration eines Insert-Cursors:

```
DEFINE the_company LIKE customer.company,
    the_fname LIKE customer.fname,
    the_lname LIKE customer.lname
DECLARE new_custs CURSOR FOR
    INSERT INTO customer (company, fname, lname)
        VALUES (the_company, the_fname,'the_lname)
```

*Bild 6-3* ***Ein Teil einer 4GL-Anwendung; Deklaration eines Insert-Cursors***

Beim Öffnen eines Insert-Cursors wird im Speicher ein Puffer angelegt, der einen Block von Sätzen aufnimmt. Der Puffer nimmt Sätze auf, sobald sie vom Programm zur Verfügung gestellt werden; wenn der Puffer voll ist, werden die Sätze in einem Block an den Datenbankserver geleitet. Dies verringert den Dialog zwischen einem Programm und dem Datenbankserver. Weiterhin ermöglicht dies dem Datenbankserver, die Sätze einfacher einzufügen. Daraus ergibt sich, daß das Einfügen schneller geht.

Die minimale Größe des Einfüge-Puffers wird bei jeder Implementation von eingebettetem SQL automatisch gesetzt; Sie haben darauf keinen Einfluß (der Puffer ist normalerweise ein oder zwei KByte groß). Die Größe des Puffers wird immer so groß gewählt, daß mindestens zwei einzufügende Sätze aufgenommen werden können. Wenn die Sätze kürzer als die minimale Puffergröße sind, dann kann der Puffer mehrere Sätze aufnehmen.

## Mit einem Cursor einfügen

In Bild 6-3 wird ein Insert-Cursor für die Verwendung vorbereitet. Die Fortsetzung des Codes in Bild 6-4 zeigt, wie der Cursor verwendet werden kann. Der Einfachheit halber wird in diesem Beispiel angenommen, daß die Funktion **next_cust** entweder Daten über eine neuen Kunden zurückliefert oder NULL-Werte, um damit das Eingabe-Ende anzuzeigen.

```
WHENEVER ERROR CONTINUE {do not terminate on error}
BEGIN WORK
OPEN new_custs
WHILE status = 0
    CALL next_cust() RETURNING the_company, the_fname, the_lname
    IF the_company IS NULL THEN
        EXIT WHILE
    END IF
    PUT new_custs
END WHILE
IF status = 0 THEN          {no problem in a PUT}
    FLUSH new_custs         {write any last rows}
END IF
IF status = 0 THEN          {no problem writing}
    COMMIT WORK             {..make it permanent}
ELSE
    ROLLBACK WORK           {retract any changes}
END IF
```

*Bild 6-4* ***Fortsetzung von Bild 6-3; der Programm-Code, der den Insert-Cursor verwendet***

Der Code in Bild 6-4 ruft die Funktion **next_cust** mehrfach auf. Wenn die Funktion keine NULL-Werte zurückliefert, dann übergibt die PUT-Anweisung die gelieferten Daten an den Satz-Puffer. Wenn der Puffer voll ist, werden die in ihm enthaltenen Sätze automatisch an den Datenbankserver übergeben. Wenn **next_cust** keine Daten mehr liefert, wird die Schleife auf die normale Weise beendet. Anschließend wird die FLUSH-Anweisung ausgeführt, um alle im Puffer verbliebenen Sätze zu schreiben. Nach dieser Anweisung wird die Transaktion beendet.

Betrachten Sie die INSERT-Anweisung in Bild 6-3 noch einmal genau. Sieht man von ihren Bestandteilen zur Cursor-Definition ab, fügt die Anweisung an sich einen einzelnen Satz in die Tabelle **customer** ein. Tatsächlich könnte man alle zum Insert-Cursor zugehörigen Anweisungen aus dem Beispiel-Code löschen und die INSERT-Anweisung könnte man in Bild 6-4 an die Position der PUT-Anweisung schreiben. Die Verwendung eines Insert-Cursors aber beschleunigt den Programmablauf.

## Status-Werte nach den Anweisungen PUT und FLUSH

Wenn ein Programm eine PUT-Anweisung ausführt, dann sollte es prüfen, ob der Satz in den Puffer erfolgreich eingefügt wurde. Wenn der neue Satz in den Puffer paßt, dann ist das Kopieren des Satzes in den Puffer die einzige Aktion, die die Anweisung PUT ausführt. In diesem Fall kann kein Fehler auftreten. Paßt der Satz jedoch nicht in den Puffer, so wird der gesamte Pufferinhalt an den Datenbankserver zum Einfügen weitergeleitet. In diesem Fall kann ein Fehler auftreten.

Die in SQLCA übertragenen Werte geben dem Programm die Informationen, die es für die Unterscheidung all dieser Fälle braucht. SQLCODE wird nach jeder PUT-Anweisung gesetzt – auf Null (0), wenn kein Fehler auftrat und auf einen negativen Wert, wenn ein Fehler auftrat.

Das dritte Element von SQLERRD enthält die Anzahl der Sätze, die tatsächlich in die Tabelle eingefügt wurden: Es wird Null (0) eingetragen, wenn der neue Satz nur in den Puffer geschrieben wird; wenn der Pufferinhalt ohne Fehler eingefügt wird, wird die Anzahl der Sätze eingetragen, die im Puffer waren; wenn ein Fehler auftritt, wird die Anzahl der Sätze eingetragen, die vor dem Auftreten des Fehlers eingefügt wurden.

Lesen Sie den Code in Bild 6-4 noch einmal, um zu sehen, wie SQLCODE verwendet wird.Wenn die OPEN-Anweisung einen Fehler meldet, wird die Schleife nicht ausgeführt (weil die WHILE-Bedingung scheitert). Die FLUSH-Operation wird nicht durchgeführt und die Transaktion wird zurückgenommen.

Wenn die PUT-Anweisung anschließend einen Fehler zurückliefert, dann wird die Schleife beendet (wegen der WHILE-Bedingung). Die FLUSH-Operation wird nicht durchgeführt und die Transaktion wird zurückgenommen. Dies kann nur dann auftreten, wenn die Schleife genügend Sätze erzeugt, um den Puffer mindestens einmal zu füllen; ansonsten kann die PUT-Anweisung keinen Fehler erzeugen.

Das Programm könnte die Schleife beenden, wenn im Puffer noch Sätze stehen; möglicherweise sogar, ohne Sätze eingefügt zu haben. In diesem Fall ist der SQL-Status Null (0) und die FLUSH-Operation wird durchgeführt. Wenn die FLUSH-Operation zu einem Fehler führt, wird die Transaktion zurückgenommen. Nur wenn alle Einfügungen erfolgreich durchgeführt wurden, wird eine Transaktion beendet.

## Sätze, die aus Konstanten bestehen

Der Mechanismus des Insert-Cursors unterstützt einen speziellen Fall, bei dem leicht hohe Performance erzielt werden kann. In diesem Fall sind alle Werte, die in der INSERT-Anweisung aufgelistet sind, Konstanten – keine Ausdrücke und keine Host-Variablen, sondern nur literale Zahlen und Zeichenketten. Es ist dabei unerheblich, wie oft solch eine INSERT-Operation ausgeführt wird – die von ihr erzeugten Sätze sind alle identisch. In diesem Fall ist es nicht sinnvoll, jeden identischen Satz zu kopieren, zu puffern und zu übertragen.

Statt dessen macht die PUT-Anweisung für diese Art einer INSERT-Operation nichts anderes, als einen Zähler zu erhöhen. Wenn zum Schluß die FLUSH-Operation durchgeführt wird, wird eine einzige Kopie des Satzes und die Anzahl der Einfügungen an den Datenbankserver geleitet. Der Datenbankserver führt das Erzeugen und Einfügen dieser Sätze in einem Schritt durch.

Es ist nicht üblich, eine Menge identischer Sätze einzufügen. Wenn man zum ersten Mal eine Datenbank einrichtet, könnte man dies verwenden, um eine große Tabelle mit NULL-Werten zu füllen.

## Ein Beispiel für einen Einfügungsvorgang

Im vorherigen Abschnitt über die DELETE-Anweisung ist ein Beispiel enthalten, in dem doppelte Sätze einer Tabelle gesucht und gelöscht wurden. (Siehe den Abschnitt "Mit einem Cursor löschen" auf Seite 6-7). Eine bessere Lösung dafür ist, die gewünschten Sätze auszuwählen, anstatt die nicht gewünschten zu löschen. Der **INFORMIX-4GL**-Code in Bild 6-5 zeigt eine Möglichkeit hierfür. Dieses Beispiel wurde in **INFORMIX-4GL** geschrieben, um einige Funktionen benutzen zu können, die die SQL-Programmierung einfacher machen.

```
BEGIN WORK
INSERT INTO new_orders
    SELECT * FROM ORDERS main
        WHERE 1 = (SELECT COUNT(*) FROM ORDERS minor
                    WHERE main.order_num = minor.order_num)
COMMIT WORK

DEFINE ord_row RECORD LIKE orders,
    last_ord LIKE orders.order_num
DECLARE dup_row CURSOR FOR
    SELECT * FROM ORDERS main INTO ord_row.*
        WHERE 1 < (SELECT COUNT(*) FROM ORDERS minor
                    WHERE main.order_num = minor.order_num)
        ORDER BY order_date
DECLARE ins_row CURSOR FOR
    INSERT INTO new_orders VALUES (ord_row.*)

BEGIN WORK
OPEN ins_row
LET last_ord = -1
FOREACH dup_row
    IF ord_row.order_num <> last_ord THEN
        PUT ins_row
        LET last_ord = ord_row.order_num
    END IF
END FOREACH
CLOSE ins_row
COMMIT WORK
```

*Bild 6-5* ***Ein 4GL-Programm, das eine Tabelle ohne doppelte Sätze neu erstellt***

Dieses Beispiel beginnt mit einer normalen INSERT-Anweisung, die all diejenigen Sätze der Tabelle sucht, die nur einmal vorkommen. Die Anweisung fügt diese Sätze in eine andere Tabelle ein; diese wurde vermutlich vor Beginn des Programms erstellt. Die Anweisung läßt nur die doppelt vorkommenden Sätze zurück. (Dieses Beispiel bezieht sich auf eine andere Datenbank, da die Tabelle **orders** in der Beispiel-Datenbank über einen UNIQUE-Index verfügt und keine doppelten Sätze enthalten kann.)

Bei **INFORMIX-4GL** kann man eine Datenstruktur *genauso wie* (LIKE) eine Tabelle definieren; diese Struktur erhält automatisch pro Spalte der Tabelle ein Element. Die Struktur **ord_row** ist ein Puffer, der einen Satz der Tabelle aufnehmen kann.

Der Code in Bild 6-5 deklariert zwei Cursor. Der erste, genannt **dup_row**, liefert die doppelten Sätze der Tabelle zurück. Da er nur Eingaben zur Verfügung stellt, kann er die ORDER BY-Klausel verwenden, um die doppelten Sätze in einer anderen Reihenfolge als der physikalischen zu erhalten (die

physikalische Reihenfolge wurde in Bild 6-2 auf Seite 6-7 verwendet). In diesem Beispiel werden die Sätze nach dem Datum (dem ältesten, das behalten wurde) sortiert, aber man kann auch jede beliebige andere Reihenfolge der Daten verwenden.

Der zweite Cursor ist ein Insert-Cursor. Er wurde geschrieben, um die *Stern*-Notation (*) von **INFORMIX-4GL** vorteilhaft verwenden zu können; man kann auf einfache Weise allen Spalten Werte zuweisen, indem man bei einem Record (eine Datenstruktur, die einem Datensatz entspricht) einen Stern angibt und damit auf *alle Komponenten* verweist.

Der restliche Code untersucht die von **dup_row** zurückgelieferten Sätze. Er fügt aus jeder Gruppe mit doppelten Sätzen den ersten in die neue Tabelle ein und ignoriert die restlichen.

In diesem Beispiel wird die einfachste Art der Fehlerbehandlung verwendet. Sofern nicht anders angegeben, wird ein **INFORMIX-4GL**-Programm automatisch abgebrochen, sobald in SQLCODE eine Fehlernummer eingetragen wird. In diesem Fall wird die aktuelle Transaktion zurückgenommen. Dieses Programm basiert auf diesem Verhalten; wenn das Programmende erreicht wird, nimmt es an, daß keine Fehler aufgetreten sind und daß die Transaktion beendet werden kann. Diese Art der Fehlerbehandlung kann dann durchgeführt werden, wenn Fehler unwahrscheinlich sind und wenn das Programm von Benutzern verwendet wird, die die Ursache für einen Programmabbruch nicht kennen müssen.

# Die UPDATE-Anweisung

Man kann die UPDATE-Anweisung in ein Programm einbetten. Die Anweisung kann so gestaltet werden wie in Kapitel 4 dieses Handbuchs beschrieben. Hierbei steht eine zusätzliche Funktion zur Verfügung: Man kann in den Ausdrücken der SET- und WHERE-Klausel Host-Variablen verwenden. Darüberhinaus verfügt ein Programm über die Möglichkeit, einen Satz mit einem Cursor zu aktualisieren.

Wie viele Sätze sind davon betroffen? SQLCODE und SQLERRD

Wenn in Ihrem Programm ein Cursor für die Auswahl von Sätzen verwendet wird, dann kann das Programm SQLCODE auf 100 überprüfen (den Rückgabewert, wenn keine Daten verfügbar sind). Dieser Wert wird gesetzt, um anzuzeigen, daß keine Sätze oder keine weiteren Sätze den Abfrage-Bedingungen entsprechen. Dieser Wert wird in SQLCODE nur nach SELECT-Anweisungen gesetzt, nicht aber nach DELETE-, INSERT- oder UPDATE-Anweisungen.

Eine Abfrage, die keine Daten findet, ist nicht erfolgreich. Im Gegensatz dazu wird eine UPDATE- oder DELETE-Anweisung, die keine Sätze aktualisiert bzw. löscht als erfolgreich betrachtet: die Anweisung hat genau die Anzahl Sätze aktualisiert bzw. gelöscht, die die WHERE-Klausel geliefert hat; die Anzahl war jedoch Null (0).

Auch bei der INSERT-Anweisung wird SQLCODE nicht auf 100 gesetzt; selbst dann nicht, wenn die Quelle für die bereitzustellenden Sätze eine SELECT-Anweisung ist, aber diese keine Sätze ausgewählt hat. Die INSERT-Anweisung ist erfolgreich, da sie soviele Sätze eingefügt hat, wie sie sollte (nämlich Null).

Um die Anzahl der eingefügten, aktualisierten oder gelöschten Sätze zu ermitteln, kann man in einem Programm das dritte Element von SQLERRD überprüfen. Dieses Element enthält die Anzahl der Sätze, ungeachtet des Wertes (Null (0) oder negativ) in SQLCODE.

# Die Verwendung eines Update-Cursors

Mit einem *Update-Cursor* kann man den aktuellen Satz löschen oder korrigieren; der aktuelle Satz ist der zuletzt geholte Satz. Dieses Beispiel (in **INFORMIX-ESQL/COBOL**) zeigt die Deklaration eines Update-Cursors:

```
EXEC SQL
    DECLARE names CURSOR FOR
        SELECT fname, lname, company
        FROM customer
    FOR UPDATE
END-EXEC
```

Das Programm, das diesen Cursor verwendet, kann Sätze auf die übliche Weise holen.

```
EXEC SQL
    FETCH names INTO :FNAME, :LNAME, :COMPANY
END-EXEC.
```

Wenn das Programm entscheidet, daß dieser Satz verändert werden muß, so kann es dies tun.

```
IF COMPANY IS EQUAL TO 'SONY'
    EXEC SQL
        UPDATE customer
            SET fname = 'Midori', lname = 'Tokugawa'
            WHERE CURRENT OF names
    END-EXEC.
```

Die Wörter `CURRENT OF names` nehmen in der WHERE-Klausel den Platz der üblichen Prüf-Ausdrücke ein. In anderer Hinsicht ist dies eine gewöhnliche UPDATE-Anweisung – es wird auch der Tabellenname angegeben, der zwar implizit im Cursornamen enthalten, aber trotzdem erforderlich ist.

## Der Bedeutung des Schlüsselwortes UPDATE

Die Bedeutung des Schlüsselwortes UPDATE in einem Cursor besteht darin, dem Datenbankserver mitzuteilen, daß ein Programm jeden geholten Satz aktualisieren (oder löschen) darf. Der Datenbankserver setzt bei Sätzen, die mit einem Update-Cursor geholt wurden, eine anspruchsvollere Sperre; wenn ein Satz mit einem Cursor geholt wird, der ohne dieses Schlüsselwort deklariert wurde, setzt er eine weniger anspruchsvolle Sperre. Dies führt bei gewöhnlichen Cursors zu einer besseren Performance und in einem Multiprozeß-System zu einer höheren Ebene der Parallelverarbeitung. (Sperrebenen und Ebenen der Parallelbearbeitung werden in diesem Kapitel weiter unten behandelt.)

### Bestimmte Spalten aktualisieren

Man kann einen Cursor deklarieren, um bestimmte Spalten zu aktualisieren. Diese überarbeitete Ausgabe des vorherigen Beispiels zeigt dies:

```
EXEC SQL
    DECLARE names CURSOR FOR
        SELECT fname, lname, company, phone
            INTO  :FNAME,:LNAME,:COMPANY,:PHONE FROM customer
    FOR UPDATE OF fname, lname
END-EXEC.
```

Mit diesem Cursor können nur die Spalten **fname** und **lname** aktualisiert werden. Eine Anweisung wie die folgende wird als fehlerhaft zurückgewiesen:

```
EXEC SQL
    UPDATE customer
        SET company = 'Siemens'
        WHERE CURRENT OF names
END-EXEC.
```

Wenn das Programm versucht, eine solche Aktualisierung durchzuführen, wird eine Fehlernummer zurückgeliefert und die Aktualisierung wird nicht vollzogen. Auch der Versuch, mit WHERE CURRENT OF zu löschen, wird abgelehnt, da das Löschen alle Spalten betrifft.

### Das Schlüsselwort UPDATE wird nicht immer gebraucht

Der ANSI-Standard für SQL unterstützt in der Cursor-Definition die Klausel FOR UPDATE nicht. Wenn ein Programm eine ANSI-kompatible Datenbank verwendet, kann es jeden beliebigen Cursor zum Aktualisieren und Löschen verwenden.

## Eine Tabelle „aufräumen"

Ein abschließendes, hypothetisches Beispiel, das einen Update-Cursor verwendet, zeigt ein Problem, das bei einer tatsächlichen Datenbank nie entstehen sollte. Es kann aber in der Anfangsphase einer Anwendungsentwicklung vorkommen.

Es wird eine große Tabelle namens **target** erzeugt und mit Daten gefüllt. Die Spalte **datcol** nimmt versehentlich einige NULL-Werte auf. Diese Sätze sollten gelöscht werden. Außerdem wird die neue Spalte **serials** in die Tabelle ein-

gefügt (unter Verwendung des Kommandos ALTER TABLE); diese Spalte kann eindeutige, ganzzahlige Werte aufnehmen. Bild 6-6 zeigt den INFORMIX-ESQL/C-Code, der dieses ausführt.

```
exec sql begin declare section;
char[80] dcol;
short int dcolint;
int sequence;
exec sql end declare section;

exec sql declare target_row cursor for
    select datcol
        into $dcol:dcolint
        from target
    for update of serials;
exec sql begin work;
exec sql open target_row;
if (sqlca.sqlcode == 0) $ fetch next target_row;
for(sequence = 1; sqlca.sqlcode == 0; ++sequence)
{
    if (dcolint < 0) /* null datcol */
        exec sql delete where current of target_row;
    else
        exec sql update target set serials = $sequence
            where current of target_row;
}
if (sqlca.sqlcode >= 0)
    exec sql commit work;
else
    exec sql rollback work;
```

*Bild 6-6* ***Eine selbsterstellte Tabelle mit Hilfe eines Update-Cursors aufräumen***

# Zusammenfassung

Ein Programm kann die Anweisungen INSERT, DELETE und UPDATE wie in Kapitel 4 beschrieben ausführen. Ein Programm kann eine Tabelle mit einem Cursor durchsuchen und ausgewählte Sätze aktualisieren oder löschen. Es kann einen Cursor auch zum Einfügen von Sätzen verwenden, mit dem Vorteil, daß die Sätze gepuffert werden und in Blöcken an den Datenbankserver übertragen werden.

Bei all diesen Tätigkeiten muß sich das Programm bemühen, Fehler aufzudecken und es muß die Datenbank in einen bekannten Status zurücksetzen, wenn ein Fehler auftritt. Das wichtigste Werkzeug hierfür ist die Transak-

tionsprotokollierung. Ohne eine Transaktionsportokollierung ist es für das Programm sehr schwierig, den Zustand vor dem Auftreten der Fehlern wiederherzustellen.

# Programmieren für Mehrbenutzer-Betrieb



7

# Kapitelüberblick

Wenn Ihre Datenbank auf einem Einzelplatz-Rechner läuft, das nicht über ein Netz mit anderen Rechnern verbunden ist, dann können Ihre Programme die Daten ungehindert verändern. Aber in allen anderen Fällen müssen Sie damit rechnen, daß ein anderes Programm dieselben Daten lesen oder verändern kann, die Ihr Programm gerade verändert. Dies ist *Parallelbearbeitung*: zwei oder mehrere voneinander unabhängige Programme verwenden zur selben Zeit dieselben Daten. Das vorliegende Kapitel beschäftigt sich mit Parallelbearbeitung, Sperren und Isolationsstufen.

# Parallelbearbeitung und Performance

In einem System mit mehreren Programmen ist für die Parallelbearbeitung eine gute Performance äußerst wichtig. Wenn der Zugriff auf die Daten *seriell* erfolgt, d. h. wenn immer nur ein Programm auf die Daten zugreifen kann, dann wird die Verarbeitung drastisch verlangsamt.

# Sperren und Integrität

Wenn die Verwendung der Daten nicht kontrolliert würde, könnte die Parallelbearbeitung auch eine Reihe negativer Auswirkungen mit sich bringen. Programme könnten veraltete Daten lesen oder Änderungen könnten verloren gehen.

Der Datenbankserver verhindert diese Fehler, indem er ein *Sperr*-System aufbaut. Eine Sperre ist ein Anspruch (oder eine Reservierung), den ein Programm auf einen Teil der Daten anmeldet. Der Datenbankserver garantiert, daß kein anderes Programm diese Daten ändern kann, solange sie gesperrt sind. Wenn ein anderes Programm diese Daten anfordert, veranlaßt der Datenbankserver das Programm entweder zu warten oder er beantwortet die Anforderung mit einer Fehlermeldung.

# Sperren und Performance

Da bei einer Sperre der Zugriff auf einen Teil der Daten seriell ist, verringert eine Sperre die Parallelbearbeitung. Alle anderen Programme, die auf diese Daten zugreifen wollen, müssen warten. Der Datenbankserver kann einen einzelnen Satz sperren, eine Page (die mehrere Sätze enthält), eine ganze Tabelle oder eine komplette Datenbank. Je mehr Sperren gesetzt sind und je größer die gesperrten Objekte sind, desto mehr wird die Parallelbearbeitung eingeschränkt. Je weniger Sperren gesetzt sind und je kleiner die Objekte sind, desto besser ist die Parallelbearbeitung (und die Performance).

Dieser Abschnitt erläutert, wie ein Programm folgende zwei Ziele erreichen kann:

- Alle benötigten Sperren setzen, um die Datenintegrität zu sichern.<
- 
- In Übereinstimmung mit dem vorangegangenen Ziel, so wenig Daten wie möglich sperren.

# Probleme bei der Parallelbearbeitung

Um die Gefahren bei der Parallelbearbeitung zu verstehen, muß man sich vorstellen, daß jedes Programm in seiner eigenen Geschwindigkeit läuft. Hierzu ein Beispiel. Nehmen wir an, daß Ihr Programm mit dem folgenden Cursor Sätze holt:

```
DECLARE sto_curse CURSOR FOR
    SELECT * FROM stock
        WHERE manu_code = 'ANZ'
```

Die Übertragung eines jeden Satzes vom Datenbankserver zum Programm braucht Zeit. Während und zwischen den Übertragungen können andere Programme andere Datenbank-Operationen durchführen. Nehmen wir an, daß zur gleichen Zeit, zu der Ihr Programm die von dieser Abfrage erzeugten Sätze holt, ein anderes Benutzerprogramm folgende Aktualisierung ausführt:

```
UPDATE stock
    SET unit_price = 1.15 * unit_price
    WHERE manu_code = 'ANZ'
```

Anders gesagt, beide Programme lesen dieselbe Tabelle. Dabei holt das eine bestimmte Sätze und das andere verändert dieselben Sätze. Es gibt vier Möglichkeiten, was als nächstes passieren kann:

1. Das andere Programm hat seine Aktualisierung bereits abgeschlossen, bevor Ihr Programm den ersten Satz holt.

   Ihr Programm zeigt somit nur die aktualisierten Sätze an.

2. Ihr Programm holt jeden Satz, bevor das andere Programm diesen aktualisieren kann.

   Ihr Programm zeigt nur die Originalsätze (die ursprünglichen Datensätze) an.

3. Nachdem Ihr Programm einige Originalsätze gelesen hat, holt das andere Programm auf und aktualisiert einige Sätze, die Ihr Programm noch nicht gelesen hat. Anschließend führt es COMMIT WORK aus.

   Ihr Programm könnte dann eine Mischung aus Originalsätzen und aktualisierten Sätzen zurückliefern.

4. Wie drittens, außer daß das andere Programm nach der Aktualisierung der Tabelle ROLLBACK WORK ausführt.

   Ihr Programm kann dann eine Mischung aus Originalsätzen und aktualisierten Sätzen anzeigen, die aber in der Datenbank nicht mehr vorhanden sind.

Die ersten beiden Möglichkeiten sind harmlos. Im ersten Fall ist die Aktualisierung bereits abgeschlossen, bevor Ihr Programm Sätze holt. Es besteht dabei kein Unterschied, ob es vor einer Mikrosekunde oder vor einer Woche beendet wurde.

Im zweiten Fall ist Ihre Abfrage bereits abgeschlossen, bevor die Aktualisierung beginnt.

Die anderen beiden Möglichkeiten können jedoch für einige Anwendungen sehr wichtig sein. Im dritten Fall liefert die Abfrage eine Mischung aus aktualisierten Sätzen und Originalsätzen zurück. Bei einigen Anwendungen darf dies auf keinen Fall passieren, bei anderen kann es eventuell gar nichts ausmachen.

Im vierten Fall kann es sich katastrophal auswirken, wenn ein Programm aufgrund einer zurückgesetzten Transaktion einige Datensätze liefert, die gar nicht mehr in der Tabelle vorhanden sind.

Ein weiteres Problem entsteht, wenn Ihr Programm einen Cursor verwendet, um den zuletzt geholten Satz zu aktualisieren oder zu löschen. Die folgende Handlungssequenz führt zu Fehlern:

- Ihr Programm holt den Satz.
- Ein anderes Programm aktualisiert oder löscht den Satz.
- Ihr Programm aktualisiert oder löscht mit der Klausel WHERE CURRENT OF *names*.

Parallelbearbeitungen wie diese kann man kontrollieren, indem man die Funktionen Sperren und *Isolationsstufe* des Datenbankservers verwendet.

# Die Arbeitsweise von Sperren

Der Datenbankserver **INFORMIX-OnLine Dynamic Server** unterstützt eine Reihe komplexer, flexibler Sperrfunktionen, die in diesem Abschnitt beschrieben sind. Einige dieser Funktionen arbeiten anders auf **INFORMIX-SE**-Datenbankservern.(Weitere Informationen entnehmen Sie dem Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen*.)

## Arten von Sperren

**INFORMIX-OnLine Dynamic Server** unterstützt drei Arten von Sperren, die es in unterschiedlichen Situationen verwendet:

*shared* — Eine Share-Sperre sieht vor, daß man Objekte nur lesen kann. Sie verhindert, daß die Objekte bei bestehender Sperre verändert werden. Eine Share-Sperre kann von mehr als einem Programm auf dasselbe Objekt gesetzt werden.

*exclusive* — Eine Exklusiv-Sperre reserviert das Objekt für ein einzelnes Programm. Sie wird verwendet, wenn ein Programm das Objekt verändert.

Eine Exklusiv-Sperre kann nur dort gesetzt werden, wo noch keine Sperre vorhanden ist. Sobald eine Sperre bereits gesetzt wurde, kann keine weitere auf dasselbe Objekt gesetzt werden.

*promotable* — Eine Promotable-Sperre legt die Aktualisierungs-Absicht fest. Diese Sperre kann nur dort gesetzt werden, wo noch keine andere Sperre vorhanden ist; sobald sie jedoch gesetzt ist, können Share-Sperren hinzu kommen. Diese Sperre kann später in eine Exklusiv-Sperre umgewandelt werden (*to promote = aufwerten, befördern*).

# Sperrbereich

Sie können Sperren auf komplette Datenbanken, ganze Tabellen, Pages, einzelne Sätze und Index-Schlüsselwerte setzen. Die Größe des gesperrten Objekts wird als Sperr*bereich* übergeben (manchmal auch *Sperreinheit* genannt). Allgemein gilt, je größer der Sperrbereich ist, desto mehr wird die Parallelbearbeitung eingeschränkt. Aber desto einfacher wird auch die Programmierung.

## Sperren auf Datenbank-Ebene

Sie können eine komplette Datenbank sperren. Beim Öffnen einer Datenbank wird auf die angegebene Datenbank eine Sperre gesetzt. Eine Datenbank wird mit den Anweisungen CONNECT, DATABASE oder CREATE DATABASE geöffnet. Solange ein Programm eine Datenbank geöffnet hält, verhindert die Share-Sperre, daß andere Programme die Datenbank löschen oder eine Exklusiv-Sperre auf sie setzen können.

Mit der folgenden Anweisung kann man eine Datenbank exklusiv sperren:

```
DATABASE datenbankname EXCLUSIVE
```

Diese Anweisung wird erfolgreich ausgeführt, wenn kein anderes Programm diese Datenbank bereits geöffnet hat. Sobald die Sperre gesetzt ist, kann kein anderes Programm die Datenbank öffnen, nicht einmal zum Lesen (weil der Versuch fehlschlägt, eine Share-Sperre auf die angegebene Datenbank zu setzen).

Eine Datenbank-Sperre wird nur durch das Schließen der Datenbank aufgehoben. Eine Datenbank kann man mit der Anweisung DISCONNECT oder CLOSE DATABASE explizit schließen oder implizit, indem man eine weitere DATABASE-Anweisung ausführt.

Da das Sperren einer Datenbank die Parallelbearbeitung in dieser Datenbank unmöglich macht, vereinfacht dies die Programmierung – negative Auswirkungen, die auf gleichzeitigem Zugriff beruhen, können nicht vorkommen. Man sollte jedoch nur dann eine Datenbank sperren, wenn keine anderen Programme auf sie zugreifen dürfen. Häufig sperrt man eine Datenbank, um umfassende Datenänderungen vorzunehmen; in der Regel geschieht dies außerhalb der Hauptarbeitszeit.

## Sperren auf Tabellen-Ebene

Man kann ganze Tabellen sperren. In einigen Fällen geschieht dies automatisch. Bei der Ausführung einer der folgenden Anweisungen sperrt **INFORMIX-OnLine Dynamic Server** immer eine ganze Tabelle:

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- RENAME COLUMN
- RENAME TABLE

Die Beendigung der Anweisung (oder das Ende der Transaktion) hebt die Sperre auf. Eine ganze Tabelle kann auch während bestimmter Abfragen automatisch gesperrt werden.

Mit der Anweisung LOCK TABLE kann man eine ganze Tabelle explizit sperren. Die Anweisung ermöglicht es, eine Share-Sperre oder eine Exklusiv-Sperre auf eine ganze Tabelle zu setzen.

Eine Share-Sperre auf eine Tabelle verhindert, daß die Tabelle aktualisiert wird, während Ihr eigenes Programm die Tabelle liest. **INFORMIX-OnLine Dynamic Server** erreicht durch das Setzen einer Isolationsstufe denselben Schutz (dies wird im nächsten Abschnitt beschrieben); dies ermöglicht eine bessere Parallelbearbeitung. Die Anweisung LOCK TABLE wird jedoch von allen Informix Datenbankservern unterstützt.

Eine Exklusiv-Sperre auf eine Tabelle verhindert jeden parallelen Zugriff auf die Tabelle. Wenn viele andere Programme um die Benutzung der Tabelle konkurrieren, wirkt sich dies nachteilig auf die Performance aus. Eine Exklusiv-Sperre auf eine Tabelle wird – wie eine Exklusiv-Sperre auf eine Datenbank – häufig dann verwendet, wenn umfangreiche Aktualisierungen außerhalb der Hauptarbeitszeit durchgeführt werden. Um nicht während der Hauptarbeitszeit Aktualisierungen direkt vorzunehmen, können Sie z. B. die Aktualisierungen in ein *Aktualisierungsjournal* schreiben. Außerhalb der Hauptarbeitszeit wird das Journal gelesen und alle Aktualisierungen werden in einem Block verarbeitet.

## Sperren einer Page, eines Satzes und eines Schlüssels

Ein Satz einer Tabelle ist das kleinste Objekt, das gesperrt werden kann. Ein Programm kann einen einzigen Satz oder eine Reihe von Sätzen sperren, während gleichzeitig andere Programme andere Sätze derselben Tabelle bearbeiten.

**INFORMIX-OnLine Dynamic Server** speichert Daten in Einheiten, die *Pages* genannt werden (die Methoden zur Plattenspeicherung werden im Kapitel 10 "INFORMIX-OnLine Dynamic Server Plattenspeicher" auf Seite 10-4 dieses Handbuchs beschrieben). Eine Page enthält einen oder mehrere Sätze. In einigen Fällen ist es besser, eine Page zu sperren als die einzelnen Sätze der Page.

Beim Erstellen einer Tabelle wird festgelegt, ob Sätze oder Pages gesperrt werden können. **INFORMIX-OnLine Dynamic Server** unterstützt die Klausel LOCK MODE; mit ihr kann man festlegen, ob eine Page oder eine Satz gesperrt werden soll. Den Sperrmodus kann man mit der CREATE TABLE-Anweisung festlegen und später mit der Anweisung ALTER TABLE verändern. (Andere Informix- Datenbankserver bieten diese Wahlmöglichkeit nicht an; sie sperren eine Page oder einen Satz – je nachdem, was besser durchzuführen ist.)

Die Verwendung von Page- und Satzsperren ist identisch. Jedesmal, wenn **INFORMIX-OnLine Dynamic Server** einen Satz sperren muß, wird entweder der Satz oder die Page mit dem Satz gesperrt. Dies ist von dem für diese Tabelle eingerichteten Sperrmodus abhängig.

In bestimmten Fällen muß der Datenbankserver einen Satz sperren, der nicht vorhanden ist – in Wirklichkeit sperrt er den Platz in der Tabelle, an der der Satz stehen würde, wenn er vorhanden wäre. Hierfür setzt der Datenbankserver eine Sperre auf einen Index-Schlüsselwert. Schlüsselsperren werden wie Satzsperren verwendet. Wenn eine Tabelle Satzsperren verwendet, dann werden Schlüsselsperren zum Sperren eines imaginären Satzes gesetzt. Wenn die Tabelle Page-Sperren verwendet, wird die Schlüsselsperre auf die Index-Page gesetzt, die den Schlüssel enthält oder enthalten würde, wenn der Schlüssel vorhanden wäre.

## Die Sperrdauer

Das Programm kontrolliert die Dauer einer Datenbank-Sperre. Eine Datenbanksperre wird erst beim Schließen der Datenbank aufgehoben.

Die Dauer einer Tabellen-Sperre hängt davon ab, ob die Datenbank Transaktionen verwendet. Wenn sie dies nicht tut (d. h. wenn es kein Transaktionsprotokoll gibt und wenn die Anweisung COMMIT WORK nicht verwendet wird), dann wird die Tabellen-Sperre erst durch Ausführung der Anweisung UNLOCK TABLE aufgehoben.

Die Dauer von Tabellen-, Satz- und Indexsperren hängt davon ab, welche SQL-Anweisungen und ob Transaktionen verwendet werden.

Bei der Verwendung von Transaktionen hebt das Ende der Transaktion alle Tabellen-, Satz-, Page- und Indexsperren auf.Es werden also mit Transaktionsende *alle Sperren aufgehoben.*

## Sperren bei Änderungen

Wenn der Datenbankserver einen Satz mit einem Update-Cursor holt, setzt er auf diesen Satz eine Promotable-Sperre. Wenn dies gelingt, weiß das Programm, daß kein anderes Programm diesen Satz ändern darf. Da eine Promotable-Sperre nicht exklusiv ist, können andere Programme weiterhin diesen Satz lesen. Dies ist der Performance dienlich, da das Programm, das diesen Satz geholt hat, etwas Zeit brauchen kann, bevor es eine UPDATE- oder DELETE-Anweisung ausführt.

Zu dem Zeitpunkt, zu dem ein Satz verändert wird, setzt der Datenbankserver eine Exklusiv-Sperre auf den Satz. Wenn bereits eine Promotable-Sperre besteht, wird diese in eine Exklusiv-Sperre umgewandelt.

Auch die Dauer einer Exklusiv-Sperre hängt davon ab, ob Transaktionen verwendet werden. Werden keine verwendet, wird die Sperre aufgehoben, sobald der geänderte Satz auf die Platte geschrieben wurde. Wenn Transaktionen verwendet werden, bleiben alle Exklusiv-Sperren bis zum Ende der Transaktion bestehen.

Wenn Transaktionen verwendet werden, dann wird bei jedem Löschen eines Satzes eine Schlüsselsperre gesetzt. Dies verhindert das Auftreten des folgenden Fehlers:

- Programm *A* löscht einen Satz.
- Programm *B* fügt einen Satz mit gleichem Schlüssel ein.
- Programm *A* nimmt die Transaktion zurück und zwingt den Datenbankserver, den gelöschten Satz wiederherzustellen. Was soll mit dem von *B* eingefügten Satz geschehen?

Durch das Sperren des Index kann der Datenbankserver ein zweites Programm davon abhalten, einen Satz einzufügen, bis das erste Programm die Transaktion abschließt.

Die beim Lesen gesetzten Sperren werden von der aktuellen Isolationsstufe kontrolliert.

# Die Isolationsstufe setzen

Die *Isolationsstufe* gibt den Grad an, mit dem Ihr Programm von den Aktionen anderer Programme isoliert ist. **INFORMIX-OnLine** stellt vier Isolationsstufen zur Verfügung. Der Datenbankserver führt diese aus, indem er unterschiedliche Regeln dafür aufstellt, wie ein Programm Sperren während des Lesen verwendet. (Dies betrifft nicht die Lesezugriffe bei UPATE-Cursor.)

Die Isolationsstufe setzt man mit dem Kommando SET ISOLATION LEVEL. Dieses Kommando wird von allen **INFORMIX-OnLine**-Datenbankservern unterstützt.

## Isolationsstufe DIRTY READ

In der einfachsten Isolationsstufe, DIRTY READ, wird fast gar keine Isolation durchgeführt. Wenn ein Programm einen Satz holt, wird keine Sperre gesetzt und es wird auch keine Sperre beachtet. Es werden Sätze einfach aus der Datenbank kopiert, ohne Rücksicht darauf, was die anderen Programme tun.

Ein Programm empfängt immer vollständige Datensätze. Selbst unter der Isolationsstufe DIRTY READ wird an ein Programm niemals ein Satz geliefert, in dem nur einige, aber nicht alle Spalten aktualisiert sind. Ein Programm, das die Isolationsstufe DIRTY READ verwendet, liest jedoch manchmal aktualisierte Sätze, bevor das aktualisierende Programm seine Transaktion beendet hat. Wenn das aktualisierende Programm später die Transaktion

zurücksetzt, dann verarbeitet das lesende Programm Daten, die nie wirklich vorhanden waren (siehe Fall 4 in der Übersicht der Probleme bei der Parallelbearbeitung auf Seite 7-5).

DIRTY READ ist die effizienteste Isolationsstufe. Das lesende Programm wartet nie und es veranlaßt auch nie ein anderes Programm zu warten. In folgenden Fällen ist dies die bevorzugte Isolationsstufe:

- Alle Tabellen sind statisch; d. h. parallel arbeitende Programme lesen nur die Daten, verändern diese aber nicht.
- Die Datenbank bleibt exklusiv gesperrt.
- Es ist sichergestellt, daß nur ein Programm die Datenbank verwendet.

## Isolationsstufe COMMITTED READ

Wenn ein Programm die Isolationsstufe COMMITTED READ anfordert, garantiert **INFORMIX-OnLine Dynamic Server**, daß nur Sätze zurückgeliefert werden, die bereits in die Datenbank geschrieben wurden. Dies verhindert die Situation, die in der Übersicht der Probleme bei der Parallelbearbeitung in Fall 4 auf Seite 7-5 beschrieben ist (das Lesen von Daten, die noch nicht geschrieben wurden und die anschließend in den alten Stand zurückgesetzt werden).

Die Durchführung der Isolationsstufe COMMITTED READ ist sehr einfach. Bevor ein Satz geholt wird, überprüft der Datenbankserver, ob ein Aktualisierungsvorgang den Satz gesperrt hat. Da die Sätze gesperrt sind, die bereits aktualisiert, aber nicht geschrieben wurden, stellt diese Überprüfung sicher, daß das Programm keine Daten liest, die noch nicht geschrieben wurden.

Die Isolationsstufe COMMITTED READ sperrt den geholten Satz nicht wirklich; sie ist fast so effizient wie die Isolationsstufe DIRTY READ. Sie wird angewendet, wenn jeder Datensatz als unabhängige Einheit verarbeitet wird, ohne Beziehung zu anderen Sätzen in derselben oder in anderen Tabellen.

## Isolationsstufe CURSOR STABILITY

Die nächste Isolationsstufe ist CURSOR STABILITY. Wenn sie verwendet wird, sperrt der Datenbankserver den zuletzt geholten Satz. Für einen gewöhnlichen Cursor setzt er eine Share-Sperre, für einen Update-Cursor eine Promotable-Sperre. Es wird immer nur ein Satz gesperrt, d. h. immer wenn ein Satz geholt wird, wird die Sperre des vorherigen Satzes aufgehoben (sofern der Satz nicht aktualisiert wurde; in diesem Fall bleibt die Sperre bis zum Ende der Transaktion bestehen).

Die Isolationsstufe CURSOR STABILITY stellt sicher, daß ein Satz nicht verändert wird, während er vom Programm überprüft wird. Dies ist wichtig, wenn das Programm einige andere Tabellen aktualisiert, wobei die aus diesem Satz gelesenen Daten zugrunde liegen. Mit der Isolationsstufe CURSOR STABILITY ist im Programm sichergestellt, daß der Aktualisierung aktuelle Daten zugrunde liegen.

Das folgende Beispiel, das sich auf die Beispiel-Datenbank bezieht, illustriert diesen Punkt. Programm *A* möchte einen neuen Artikel des Herstellers Hero einfügen. Parallel hierzu will das Programm *B* den Hersteller Hero und alle mit diesem verbundenen Artikelsätze löschen. Folgende Ereignisse können auftreten:

1. Programm *A*, das unter der Isolationsstufe CURSOR STABILITY arbeitet, holt aus der Tabelle **manufact** den Satz Hero, um den Herstellercode zu lesen. Auf denSatz wird eine Share-Sperre gesetzt.
2. Programm *B* führt für denselben Satz eine DELETE-Anweisung durch. Wegen der Sperre veranlaßt der Datenbankserver das Programm zu warten.
3. Programm *A* fügt in die Tabelle **stock** einen neuen Satz ein. Es verwendet hierbei den Herstellercode, den es aus der Tabelle **manufact** erhalten hat.
4. Programm *A* schließt den Cursor auf die Tabelle **manufact** (oder liest einen anderen Satz daraus), und hebt die Sperre auf.
5. Programm *B*, das nun nicht mehr warten muß, führt das Löschen des Satzes durch und fährt fort, die Sätze in der Tabelle **stock** zu löschen, die den Herstellercode HRO verwenden. Hierzu gehört auch der eben erst von Programm *A* eingefügte Satz.

Wenn Programm *A* eine niedrigere Isolationsstufe verwendet hätte, könnte statt dessen folgendes passieren:

1. Programm *A* liest aus der Tabelle **manufact** den Satz Hero, um den Herstellercode zu lesen. Es wird keine Sperre gesetzt.
2. Programm *B* setzt für denselben Satz eine DELETE-Anweisung ab. Diese wird erfolgreich durchgeführt.
3. Programm *B* löscht in der Tabelle **stock** alle Sätze, die den Herstellercode HRO verwenden.
4. Programm *B* beendet sich.
5. Programm *A*, das nicht weiß, daß seine Kopie des Satzes Hero jetzt ungültig ist, fügt unter Verwendung des Herstellercodes HRO einen neuen Satz in die Tabelle **stock** ein.
6. Programm *A* beendet sich.

Am Ende gibt es in der Tabelle **stock** einen Satz, zu dem es in der Tabelle **manufact** keinen übereinstimmenden Herstellercode gibt. Außerdem liegt in Programm *B* offensichtlich ein Fehler vor; es hat nicht die Sätze gelöscht, die es löschen sollte. Die Verwendung der Isolationsstufe CURSOR STABILITY verhindert derartig negative Auswirkungen.

(Es ist möglich, die vorangegangene Situation so umzugestalten, daß sogar dann ein Fehler auftritt, wenn die Isolationsstufe CURSOR STABILITY verwendet wird. Hierbei muß Programm *B* die Tabellen nur in umgekehrter Reihenfolge bearbeiten als Programm *A*. Wenn Programm *B* erst die Sätze aus der Tabelle **stock** löscht, bevor es den Satz aus der Tabelle **manufact** löscht, dann kann keine Isolationsstufe diesen Fehler verhindern. Wenn diese Fehlerquelle möglich ist, ist es sehr wichtig, daß alle Programme in derselben Reihenfolge zugreifen.)

Da die Isolationsstufe CURSOR STABILITY immer nur einen Satz sperrt, wird die Parallelbearbeitung weniger eingeschränkt als dies beim Sperren von Tabellen oder Datenbanken der Fall ist.

## Isolationsstufe REPEATABLE READ

Die Isolationsstufe REPEATABLE-READ bewirkt, daß der Datenbankserver jeden Satz sperrt, den das Programm holt. Für einen normalen Cursor wird eine Share-Sperre und für einen Update-Cursor eine Promotable-Sperre gesetzt. Die Sperren werden beim Holen des Satzes gesetzt. Sie werden erst aufgehoben, wenn der Cursor geschlossen wird, oder wenn die Transaktion beendet wird.

Die Isolationsstufe REPEATABLE READ ermöglicht es, daß ein Programm, das einen Scroll-Cursor verwendet, die ausgewählten Sätze mehr als einmal liest. Sie gewährleistet auch, daß diese Sätze während der Bearbeitung nicht aktualisiert werden (Scroll-Cursor sind im Kapitel 5 dieses Handbuchs beschrieben). Keine andere Isolationsstufe kann garantieren, daß die Sätze beim zweiten Lesen immer noch vorhanden und unverändert sind.

Die Isolationsstufe REPEATABLE READ setzt die meisten Sperren und läßt sie am längsten bestehen. Deshalb schränkt diese Stufe die Parallelbearbeitung am meisten ein. Wenn ein Programm diese Isolationsstufe verwendet, sollte man sorgfältig darüber nachdenken, wieviele Sperren gesetzt werden, wie lang sie bestehen bleiben und wie sich dies auf andere Programme auswirken könnte.

Zusätzlich zu der Auswirkung auf die Parallelbearbeitung kann die große Anzahl an Sperren zu einem Problem werden. Der Datenbankserver zeichnet für alle Programme, die er verwaltet, nur eine bestimmte Anzahl Sperren auf (diese Anzahl kann man einstellen). Wenn die Sperr-Tabelle voll ist, kann der

Datenbankserver keine weitere Sperre setzen und liefert eine Fehlernummer zurück. Der Administrator eines **INFORMIX-OnLine Dynamic Server**-Systems kann die Sperr-Tabelle überwachen und Ihnen sagen, wann die Tabelle ausgelastet ist.

In ANSI-kompatiblen Datenbanken ist die Isolationsstufe automatisch REPEATABLE READ. Diese Isolationsstufe ist erforderlich, um sicherzustellen, daß die Operationen entsprechend dem ANSI-Standard für SQL durchgeführt werden.

# Den Sperrmodus setzen

Der Sperrmodus legt fest, was passiert, wenn ein Programm auf gesperrte Daten stößt. Wenn ein Programm versucht, gesperrte Daten zu holen oder zu verändern, tritt eine der drei folgenden Möglichkeiten ein:

- Das Programm erhält vom Datenbankserver eine sofortige Rückmeldung mit einer Fehlernummer für SQLCODE.
- Das Programm setzt solange aus, bis die Sperre von dem Programm aufgehoben wird, das sie gesetzt hat.
- Das Programm setzt eine bestimmte Zeit aus. Wenn die Sperre nicht aufgehoben wurde, erhält das Programm vom Datenbankserver eine Fehlermeldung.

Mit dem Kommando SET LOCK MODE kann man eine dieser Möglichkeiten auswählen.

## Auf das Aufheben von Sperren warten

Wenn man warten will (und dies ist bei vielen Anwendungen das Beste), führt man folgendes Kommando aus:

```
SET LOCK MODE TO WAIT
```

Wenn jetzt das Programm auf einen Satz zugreifen muß, der von einem anderen Programm gesperrt wurde, wartet das Programm, bis die Sperre aufgehoben wird. Die Verzögerungen sind normalerweise nicht wahrnehmbar.

## Auf das Aufheben der Sperren nicht warten

Der Nachteil beim Warten auf das Aufheben von Sperren ist, daß das Warten unter Umständen sehr lang dauern kann (obwohl richtig entworfene Anwendungen die Dauer der Sperren sehr kurz halten sollten). Wenn eine mögliche, lange Verzögerung nicht akzeptabel ist, kann ein Programm das folgende Kommando ausführen:

```
SET LOCK MODE TO NOT WAIT
```

Wenn ein Programm einen gesperrten Satz anfordert, erhält es sofort eine Fehlernummer (z. B. Fehler -107, `Satz gesperrt.`). Die aktuelle SQL-Anweisung wird abgebrochen.

Beim Aufruf eines Programms ist 'nicht warten' die Anfangseinstellung. Wenn man SQL interaktiv verwendet und einen Fehler erhält, der auf Sperren verweist, dann sollte man den Sperrmodus auf Warten setzen. Wenn Sie ein Programm schreiben, dann denken Sie daran, diese Anweisung als eine der ersten vom Programm auszuführenden eingebetteten SQL-Kommandos anzugeben.

## Eine bestimmte Zeit auf das Aufheben der Sperre warten

Wenn man **INFORMIX-OnLine Dynamic Server** verwendet, hat man eine weitere Auswahlmöglichkeit: Man kann den Datenbankserver anweisen, eine obere Grenze für das Warten zu setzen. Man könnte z. B. folgendes Kommando absetzen:

```
SET LOCK MODE TO WAIT 17
```

Dies legt für jedes Warten eine obere Grenze von 17 Sekunden fest. Wenn eine Sperre innerhalb dieser Zeit nicht aufgehoben wurde, wird eine Fehlernummer zurückgeliefert.

## Der Umgang mit Deadlocks

Ein *Deadlock* ist eine Situation, in der zwei Programme sich gegenseitig am Ablaufen hindern. Jedes Programm hat ein Objekt gesperrt, auf das das andere Objekt zugreifen möchte. Ein Deadlock entsteht nur dann, wenn alle beteiligten Programme ihren Sperrmodus auf Warten gesetzt haben.

Wenn nur Daten eines einzelnen Netzwerk-Servers beteiligt sind, entdeckt **INFORMIX-OnLine Dynamic Server** Deadlocks sofort. **INFORMIX-OnLine** verhindert das Auftreten eines Deadlocks, indem an das zweite Programm, das eine Sperre setzen will, eine Fehlernummer (Fehler -143 `ISAM error: Ein DEADLOCK wurde erkannt.`) zurückgegeben wird. Diese Fehlernummer empfängt ein Programm nur dann, wenn der Sperrmodus auf "nicht warten" gesetzt ist. Wenn das Programm eine Fehlernummer erhält, die sich auf Sperren bezieht, obwohl der Sperrmodus auf Warten gesetzt ist, wissen Sie, daß die Ursache ein bevorstehender Deadlock ist.

## Der Umgang mit externen Deadlocks

Ein Deadlock kann auch zwischen Programmen auftreten, die auf unterschiedlichen Datenbankservern laufen. In diesem Fall kann **INFORMIX-OnLine** den Deadlock nicht sofort entdecken. (Eine vollkommene Deadlock-Aufdeckung erfordert übermäßige Kommunikation zwischen allen Datenbankservern in einem Netzwerk.) Statt dessen setzt jeder Datenbankserver eine Begrenzung der Zeit, die ein Programm warten kann, um in einem anderen Datenbankserver eine Sperre auf Daten zu setzen. Wenn die Zeit verstrichen ist, nimmt der Datenbankserver an, daß der Grund hierfür ein Deadlock war und er liefert eine Fehlernummer zurück, die sich auf Sperren bezieht.

Anders gesagt, wenn externe Datenbanken verwendet werden, läuft jedes Programm mit einer maximalen Wartezeit. Das Maximum wird für jeden einzelnen Datenbankserver gesetzt und kann vom Datenbankverwalter geändert werden.

# Einfache Parallelbearbeitung

Wenn man unsicher ist, welche Wahl man bezüglich der Sperren und der Parallelbearbeitung treffen soll und wenn die Anwendung einfach ist, dann sollte man am Anfang des Programms (sofort nach der ersten DATABASE-Anweisung) die folgenden Kommandos ausführen:

```
SET LOCK MODE TO WAIT
SET ISOLATION TO REPEATABLE READ
```

Ignorieren Sie die Rückgabewerte beider Anweisungen. Arbeiten Sie so weiter, als ob keine anderen Programme existieren würden. Wenn keine Performance-Probleme entstehen, brauchen Sie kein differenzierteres Verfahren einschlagen.

# Sperren bei anderen Datenbankservern

**INFORMIX-OnLine Dynamic Server** führt eine eigene Verwaltung von Sperren durch, so daß unterschiedliche Arten von Sperren und Isolationsstufen zur Verfügung gestellt werden können. Diese wurden in den vorangegangenen Abschnitten beschrieben. Andere Informix Datenbankserver führen Sperren mit den Mitteln des Betriebssystems durch; sie können nicht dieselben Funktionen zur Verfügung stellen.

Einige Betriebssyteme stellen Sperrfunktionen als Dienstprogramme des Betriebssystems zur Verfügung. In diesen Systemen unterstützen die Datenbankserver die Anweisung SET LOCK MODE.

Einige Betriebssysteme stellen keine *Kernel-Locking*-Funktionen zur Verfügung. In diesen Systemen führt der Datenbankserver eigene Sperren durch, die auf kleinen Dateien beruhen; diese werden im Dateiverzeichnis der Datenbank angelegt. (Die Dateien haben das Suffix **.lok**.)

Führen Sie die Anweisung SET LOCK MODE aus, um den Sperrmodus zu definieren, unter dem Ihr Datenbankserver läuft. Überprüfen Sie anschließend die Fehlernummer, so wie es im folgenden Ausschnitt eines **INFORMIX-ESQL/C**-Codes gezeigt wird:

```
#define LOCK_ONLINE 1
#define LOCK_KERNEL 2
#define LOCK_FILES 3
int which_locks()
{
    int locktype;

    locktype = LOCK_FILES;
    exec sql set lock mode to wait 30;
    if (sqlca.sqlcode == 0)
        locktype = LOCK_ONLINE;
    else
    {
        exec sql set lock mode to wait;
        if (sqlca.sqlcode == 0)
            locktype = LOCK_KERNEL;
    }
    /* restore default condition */
    exec sql set lock mode to not wait;
    return(locktype);
}
```

Wenn der Datenbankserver die Anweisung SET LOCK MODE nicht unterstützt, arbeitet Ihr Programm immer im Modus NOT WAIT. D. h. jedesmal, wenn Ihr Programm versucht, einen Satz zu sperren, der bereits von einem anderen Programm gesperrt ist, wird sofort eine Fehlernummer zurückgeliefert.

## Isolation beim Lesen

Mit Ausnahme von **INFORMIX-OnLine Dynamic Server** setzen Informix-Datenbankserver beim Holen eines Satzes keine Sperren. Es gibt nichts, was mit den von **INFORMIX-OnLine Dynamic Server** verwendeten Share-Sperren der Isolationsstufe CURSOR STABILITY verglichen werden kann.

Wenn ein Programm einen Satz mit einer einzelnen SELECT-Anweisung oder mit einem Cursor holt, der nicht für die Aktualisierung (FOR UPDATE) deklariert wurde, dann wird der Satz sofort geholt; unabhängig davon, ob er gesperrt ist oder von einer nicht beendeten Transaktion verändert wurde.

Dieses Verhalten führt zur besten Performance. Aber man muß sich bewußt sein, daß das Programm auch Sätze lesen kann, die von nicht beendeten Transaktionen geändert wurden.

Die Wirkung der Isolationsstufe CURSOR STABILITY kann man erzielen, indem man einen Cursor FOR UPDATE deklariert und diesen dann für Eingaben verwendet. Der Datenbankserver setzt jedesmal, wenn er einen Satz mit einem Update-Cursor holt, eine Sperre auf den geholten Satz (Wenn der Satz bereits gesperrt ist, wartet das Programm oder es erhält eine Fehlermeldung; dies ist vom Sperrmodus abhängig.). Wenn das Programm einen anderen Satz holt, ohne den aktuellen verändert zu haben, wird die Sperre des aktuellen Satzes aufgehoben und der neue Satz wird gesperrt.

Wenn man Sätze mit einem Update-Cursor holt, kann man daher sicher sein, daß der geholte Satz während der Bearbeitung gesperrt ist (Der Satz kann nicht *veralten.*). Man kann auch sicher sein, nur Daten zu erhalten, die auf die Platte geschrieben sind. Der Grund dafür ist, daß die Sperren auf aktualisierte Sätze bis zum Ende der Transaktion bestehen bleiben. Abhängig vom Betriebssystem und dem Datenbankserver stellt man vielleicht einen Performance-Nachteil fest, wenn man einen Update-Cursor auf diese Weise verwendet.

## Aktualisierte Sätze sperren

Wenn ein Cursor für das Aktualisieren (FOR UPDATE) deklariert wurde, werden Sperren wie folgt behandelt: Bevor ein Satz geholt wird, wird er gesperrt. Wenn er nicht gesperrt werden kann, wird das Programm zum Warten veranlaßt oder es wird eine Fehlermeldung zurückgeliefert.

Bei der nächsten Anweisung zum Holen eines Datensatzes registriert der Datenbankserver, ob der aktuelle Satz geändert wurde (wobei entweder die UPDATE- oder die DELETE-Anweisung mit der Klausel WHERE CURRENT OF verwendet wurde) und ob eine Transaktion noch nicht beendet ist. Wenn beides der Fall ist, bleibt die Sperre dieses Satzes bestehen. Ansonsten wird die Sperre aufgehoben.

Wenn man Aktualisierungen mit einer Transaktion durchführt, bleiben alle aktualisierten Sätze bis zum Ende der Transaktion gesperrt. Sätze, die nicht geändert werden, sind nur dann gesperrt, solange sie der aktuelle Satz sind. Bei Sätzen, die außerhalb einer Transaktion oder in einer Datenbank, die kein Transaktionsprotokoll verwendet, aktualisiert werden, wird ebenfalls die Sperre aufgehoben, sobald der nächste Satz geholt wird.

# Permanenter Cursor: „Hold Cursor"

Bei der Verwendung von Transaktionsprotokollen gewährleistet der Datenbankserver, daß jede innerhalb einer Transaktion durchgeführte Aktion am Ende einer Transaktion rückgängig gemacht werden kann. Um dies zuverlässig auszuführen, wendet der Datenbankserver normalerweise folgende Regeln an:

- Alle Cursor werden beim Beenden einer Transaktion geschlossen.
- Alle Sperren werden beim Beenden einer Transaktion aufgehoben.

Diese Regeln gelten für alle Datenbanken, die Transaktionen unterstützen; sie verursachen bei den meisten Anwendungen keine Probleme. Der folgende Programmentwurf ist verwendbar, wenn es keine Transaktionsverarbeitung gibt. Bei Transaktionsverarbeitung wird der Entwurf jedoch unpraktikabel. Der Entwurf wird in Bild 7-1 in Pseudocode dargestellt.

```
DECLARE master CURSOR FOR ...
DECLARE detail CURSOR FOR ... FOR UPDATE
OPEN master
LOOP:
    FETCH master INTO ...
    IF (the fetched data is appropriate) THEN
        BEGIN WORK
        OPEN detail USING data read from master
        FETCH detail ...
        UPDATE ... WHERE CURRENT OF detail
        COMMIT WORK
    END IF
END LOOP
CLOSE MASTER
```

*Bild 7-1* ***Eine Skizze einer allgemeinen Form einer Datenbank-Anwendung in Pseudocode***

Bei diesem Entwurf wird ein Cursor für das Durchsuchen einer Tabelle verwendet. Die ausgewählten Sätze werden als Basis für die Aktualisierung einer anderen Tabelle benutzt. Das Problem besteht darin, daß die COMMIT WORK-Anweisung nach der Aktualisierung alle Cursor schließt, wenn jede Aktualisierung als eine separate Transaktion behandelt wird (wie in Bild 7-1 dargestellt), auch den Master-Cursor.

Die einfachste Möglichkeit ist, die Anweisungen COMMIT WORK und BEGIN WORK als erste bzw. letzte Anweisung zu schreiben, so daß das ganze Durchsuchen der Mastertabelle eine einzige große Transaktion ist. Dies ist in man-

chen Fällen möglich. Aber es kann unpraktisch werden, wenn viele Sätze aktualisiert werden müssen. Die Anzahl der Sperren kann zu groß werden und für die gesamte Programmdauer bestehen bleiben.

Sie könnnen jedoch bei der Deklaration des Master-Cursors die Schlüsselworte WITH HOLD hinzuzufügen. Ein solcher Cursor gilt als ein *permanenter Cursor* (Hold Cursor), der am Ende einer Transaktion nicht geschlossen wird. Der Datenbankserver schließt weiterhin alle anderen Cursor und hebt weiterhin alle Sperren auf. Der permanente Cursor bleibt solange geöffnet, bis er explizit geschlossen wird.

Bevor man einen permanenten Cursor verwendet, sollte man die Wirkungsweise der Sperrmechanismen verstanden haben und man sollte auch die parallel arbeitenden Programme verstehen. Der Grund dafür ist, daß jedesmal, wenn COMMIT WORK ausgeführt wird, alle Sperren aufgehoben werden, inklusive der Sperren auf Sätze, die mit einem permanenten Cursor bearbeitet wurden.

Wenn der Cursor wie beabsichtigt für eine einzige Durchsuchung der Tabelle verwendet wird, ist dies nicht sehr wichtig. Man kann WITH HOLD jedoch für jeden Cursor angeben, inklusive Update- und Scroll-Cursor. Bevor man dies jedoch tut, sollte man sich der Tatsache bewußt sein, daß alle Sperren (inklusive Sperren auf eine ganze Tabelle) am Ende einer Transaktion aufgehoben werden.

# Zusammenfassung

Wenn mehrere Programme auf eine Datenbank parallel zugreifen (und wenn zumindest eines davon Daten verändern kann), müssen alle Programme damit rechnen, daß ein anderes Programm die Daten verändern kann, während sie diese lesen. Der Datenbankserver stellt Sperrmechanismen und Isolationsstufen zur Verfügung. Dadurch können Programme so ablaufen, als würden nur sie auf die Daten zugreifen.

# Aufbau eines Datenmodells



8

# Kapitelüberblick

Der erste Schritt zum Aufbau einer Datenbank ist das Erstellen eines Datenmodells – einer genauen, vollständigen Beschreibung der zu speichernden Daten. Das vorliegende Kapitel beschreibt eine Methode der Datenmodellierung. Die folgenden Kapitel beschreiben, wie man ein entworfenes Datenmodell implementiert.

Um dieses Kapitel verstehen zu können, benötigen Sie zumindest grundlegende SQL-Kenntnisse. Außerdem sollten Sie das Konzept relationaler Datenbanken verstanden haben.

# Warum erstellt man ein Datenmodell

Wahrscheinlich haben Sie schon eine ungefähre Vorstellung Ihrer Datenbank im Kopf. Wenn Sie beim Skizzieren Ihres Datenmodells eine formalisierte Schreibweise verwenden, wird Ihnen diese auf zwei Arten bei der Entwicklung helfen:

- Sie durchdenken das Datenmodell.

  Ein gedankliches Modell enthält oft nicht überprüfte Annahmen. Die Formalisierung des Entwurfs bringt diese Punkte zum Vorschein.

- Es ist leichter, den Entwurf anderen Personen mitzuteilen.

  Eine formale Anweisung macht das Modell klarer, so daß andere Personen Anmerkungen und Vorschläge in der gleichen Form anbringen können.

# Entity-Relationship Datenmodell - ein Überblick

In vielen Büchern werden für die Modellierung der Daten verschiedene formale Methoden vorgestellt. Die meisten Methoden erfordern eine gründliche und genaue Arbeitsweise. Wenn Sie bereits eine Methode kennen, sollten Sie diese auf jeden Fall benutzen.Dieses Kapitel zeigt eine Zusammenfassung des Entity-Relationship Datenmodells. Es handelt sich dabei um eine Modellier-Methode, die auf dem ursprünglichen Datenmodell von E. F. Codd basiert. Diese Modellier-Methode besteht aus mehreren Schritten:

1. Bestimmung der Objekte (Entities (elementare Objekte), Relationen (Beziehungen) und Attribute (Spalten ), die die Datenbank beschreiben,
2. Aufzeichnen der Objekte mit Hilfe des Entity-Relationship-Ansatzes,
3. Überführen der Objekte des Entity-Relationship Modells in eine Tabellenstruktur,
4. Reorganisation des logischen Datenmodells,
5. Normalisierung des logischen Datenmodells.

Die Schritte 1 bis 5 werden in diesem Kapitel beschrieben. In Kapitel 9 wird der letzte Schritt hinzugefügt: Das Umwandeln des logischen Datenmodells in eine physikalische Struktur.

Das Endprodukt des Modellierens ist ein voll definiertes Datenbankdesign in Diagrammdarstellung, wie in Bild 8-17 gezeigt. Das Diagramm zeigt die endgültige Form von Tabellen für ein persönliches Adreßbuch. Das persönliche Adreßbuch ist ein Beispiel, das in diesem Kapitel entwickelt wird. Dieses wird hier anstelle der Beispieldatenbank stores6 benutzt, die im restlichen Handbuch verwendet wird. Das persönliche Adreßbuch ist klein genug, um in einem Kapitel vollständig entwickelt zu werden, aber auch groß genug, um die komplette Methode zu erläutern.

# Bestimmen der grundlegenden Datenobjekte

Der erste Schritt beim Aufbau eines Entity-Relationship Datenmodells besteht darin, die grundlegenden Datenobjekte zu identifizieren und zu definieren. Bei diesen Objekten handelt es sich um Entities, Relationen und Attribute.

## Entities festlegen

Eine Entity ist ein grundlegendes Datenobjekt, das für den Benutzer von besonderem Interesse ist. Es handelt sich dabei z.B. um eine Person, einen Ort, einen Gegenstand oder ein Ereignis, das in der Datenbank gespeichert wird. Wenn das Datenmodell eine Sprache beschreiben würde, wären die Substantive die Entities. Die Beispieldatenbank stores6 enthält folgende Entities: Kunden, Bestellungen, Artikel, Lager, Katalog, Anrufe von Kunden, Arten von Anrufen, Hersteller und Staat.

In einem ersten Schritt müssen die Entities gefunden werden, die in der Datenbank gespeichert werden sollen. Die meisten Ihrer Entities werden in Ihrem Modell zu Tabellen.

### Auswahl möglicher Entities

Falls Sie schon eine Vorstellung von Ihrer Datenbank haben, können Sie sicher sofort einige Entities aufzählen. Falls Ihre Datenbank aber auch von anderen Benutzern verwendet werden soll, sollten Sie die Punkte aufarbeiten, die Ihre Datenbank im wesentlichen beinhalten soll. Machen Sie also eine vorläufige Liste aller Entities. Befragen Sie zukünftige Benutzer der Datenbank danach, was ihrer Meinung nach in die Datenbank aufgenommen werden sollte. Legen Sie für jede Entity grundlegende Eigenschaften fest, wie beispielsweise „zu jeder Adresse gehört mindestens ein Name". All diese Entscheidungen, die Sie beim Festlegen Ihrer Entities treffen, kommen in die Sammlung Ihrer „Grundannahmen" . Der Abschnitt "Das Adressenverzeichnis" auf Seite 8-7 zeigt einige Grundannahmen für das Beispiel, das wir in diesem Kapitel verwenden.

Wenn Sie Ihr Modell später in die Normalform überführen, werden einige Entities möglicherweise noch erweitert oder verwandeln sich in andere Datenobjekte. Weitere Informationen dazu erhalten Sie im Abschnitt "Das Datenmodell normalisieren" auf Seite 8-32.

### Liste der Entities überarbeiten

Sobald die Liste Ihrer Entities komplett ist, sollten Sie sie nach folgenden Gesichtspunkten überarbeiten:

- Die Entity ist wichtig.

  Listen Sie nur Entities auf, die für die Benutzer der Datenbank wichtig sind und den Aufwand ihrer Programmierung wert sind.

- Sie bezeichnet eine Gattungsart.

  Listen Sie nur Gattungsarten von Dingen auf, keine einzelnen Ausprägungen. Beispielsweise könnte *Symphonie* ein Objekt sein, aber *Beethoven's Fünfte* wäre die Ausprägung oder Instanz einer Entitiy.

- Sie ist elementar.

  Listen Sie nur Entities auf, die eigenständig vorhanden sind und nichts anderes zu ihrer Erklärung benötigen. Alles, was man als Eigenschaft, Merkmal oder Beschreibung bezeichnen könnte, ist nur ein Attribut, keine Entität. Beispielsweise kann eine *Teile-Nummer* nicht ohne das elementarere Objekt *Teile* existieren. Listen Sie weiterhin keine Dinge auf, die man aus anderen Objekten ableiten kann; vermeiden Sie z. B. jede Summe, jeden Durchschnittswert oder andere Mengen, die man in einem SELECT- Ausdruck berechnen kann.

- Sie ist einheitlich.

  Stellen Sie sicher, daß jede genannte Entitiät eine einzelne Klasse repräsentiert – daß sie nicht in Unterkategorien mit jeweils eigenen Merkmalen aufgeteilt werden kann. Bei der Planung des Telefonverzeichnis-Modells (siehe "Das Adressenverzeichnis" auf Seite 8-7) stellt sich heraus, daß eine anscheinend einfache Entität, die Telefonnummer, aus drei Kategorien mit jeweils unterschiedlichen Merkmalen besteht (Telefon, Fax, Modem).

Diese Auswahl ist weder einfach, noch geschieht sie automatisch. Um die beste Auswahl an Entities zu treffen, muß man sehr genau über die Beschaffenheit der zu speichernden Daten nachdenken. Dies ist natürlich genau der Punkt, warum man ein formales Datenmodell erstellt.

## Das Adressenverzeichnis

Nehmen wir an, Sie möchten eine Datenbank erstellen, die ein Adressenverzeichnis beinhaltet, das Sie bisher in schriftlicher Form mit sich herumgetragen haben. Das Datenmodell besteht aus Namen, Adressen und Telefonnummern, die die Benutzer geschäftlich und privat nutzen.

Zunächst müssen die Entities festgelegt werden. Sehen wir uns dazu eine Seite aus einem Adressbuch an, um festzustellen, welche Entities dort vorhanden sind.

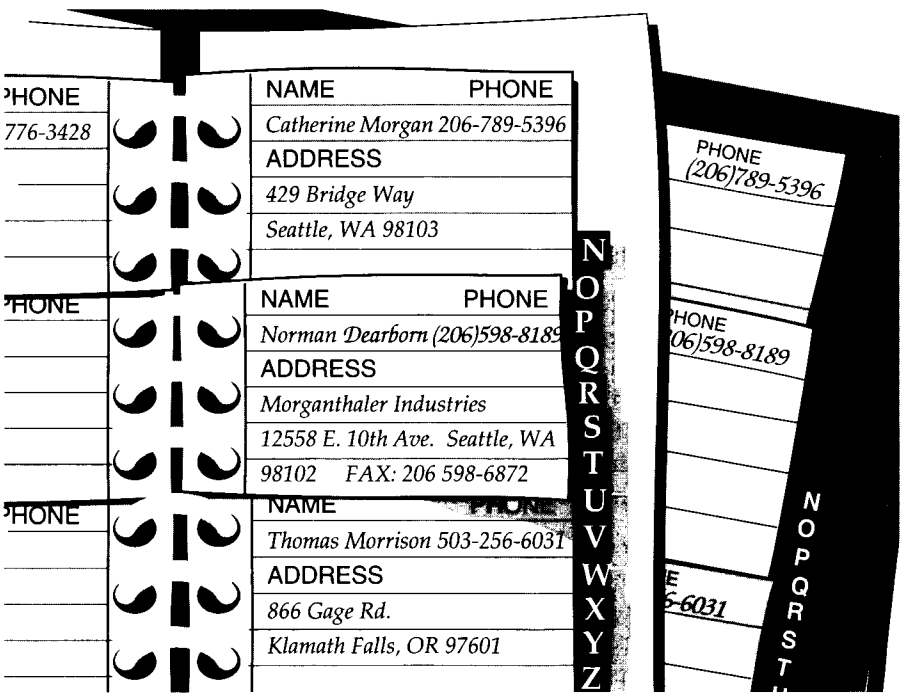



*Bild 8-1* ***Seite eines Adressenverzeichnisses***

Die hier vorhandene physikalische Form der Daten kann irreführend sein. Lassen Sie sich nicht durch den Aufbau der Seiten und der Einträge des Adreßbuchs dazu verleiten, eine Entity zu bestimmen, die einen Eintrag des Buchs repräsentiert – eine Art alphabetischen Eintrag mit Feldern für den Namen, eine Telefonnummer und eine Adresse. Denken Sie daran, daß Sie nicht den Datenträger modellieren wollen, sondern die Daten.

Auf den ersten Blick enthalten die elementaren Entities, die in einem Adreßbuch aufgezeichnet sind:

- Namen (von Personen und Firmen)
- Adressen
- Telefonnummern

Stimmen diese Entities mit den vorhin angegebenen Kriterien überein? Sie sind für das Modell sehr wichtig.

Sind sie elementar? Ein guter Test hierfür besteht in der Untersuchung, ob eine Entity sich in der Anzahl unabhängig von einer anderen Entity verändern kann. Sie können unschwer erkennen, daß ein Adreßbuch manchmal Personen enthält, die keine Telefonnummer oder aktuelle Adresse haben (Personen, die gerade umziehen oder die Stelle wechseln). Weiterhin kann ein Adreßbuch sowohl Adressen als auch Telefonnummern enthalten, die für mehr als eine Person gelten. Jede dieser drei Entities kann sich in der Anzahl selbständig verändern; dies deutet darauf hin, daß sie elementar und nicht abhängig sind.

Sind sie einheitlich? Namen können in Personennamen und Firmennamen eingeteilt werden. Nachdem Sie darüber nachgedacht haben, entscheiden Sie sich dafür, daß in dem Modell alle Namen dieselben Merkmale haben sollten; d. h. Sie wollen über eine Firma keine anderen Daten als über eine Person festhalten. Ebenso verhält es sich, wenn Sie entscheiden, daß es nur eine Art Adressen gibt; Privat- und Geschäftsadressen müssen nicht unterschiedlich behandelt werden.

Sie erkennen jedoch, daß es nicht nur eine Art von Telefonnummern gibt, sondern drei. Es gibt *Ruf*nummern (telefon_nr), die einen mit Personen verbinden, *Fax*nummern (fax_nr), die einen mit einem Faxgerät verbinden und *Modem*nummern (modem_nr), die einen mit einem Computer verbinden. Sie entscheiden sich, unterschiedliche Informationen für jede Art der Nummern festzuhalten, so daß dies drei unterschiedliche Entities sind.

Ihr Telefonbuch soll also aus fünf Entities bestehen:

- Name
- Adresse (Postadresse)
- Rufnummer
- Faxnummer
- Modemnummer

### Entities skizzieren

Später werden Sie in diesem Kapitel erfahren, wie Sie mit dem Entity-Relationship Diagramm arbeiten können. Für den Moment genügt es aber, wenn Sie für jede Entity ein Rechteck zeichnen. Im Abschnitt "Datenobjekte skizzieren" auf Seite 8-19 werden Sie sehen, wie Sie diese Entities verbinden.


*Bild 8-2* ***Entities skizzieren.***

## Beziehungen festlegen

Nachdem Sie die Entities bestimmt haben, müssen Sie nun deren Beziehungen festlegen. Die vorhandenen Beziehungen sind nicht immer offensichtlich. Es ist aber außerordentlich wichtig, daß Sie alle wichtigen Beziehungen zwischen den Entities herausfinden. Dies ist nur möglich, indem sie **alle** möglichen Beziehungen auflisten. Sie müssen also alle Kombinationen aus zwei Entities A und B zusammenfassen und sich fragen, welche Beziehung zwischen A und B besteht.

Eine Beziehung besteht zwischen zwei Entities. In der Regel wird die Art dieser Beziehung durch ein Verb oder durch eine Präposition ausgedrückt. Eine Beziehung kann mit den Termini Connectivity, Existenzabhängigkeit oder als Kardinalität bezeichnet werden.

## Connectivity

Der Terminus *Connectivity* bezieht sich auf die zulässige Anzahl der Ausprägungen der Entities. Eine Ausprägung oder Instanz ist ein ganz bestimmtes Exemplar der Entity. Die drei Arten der Connectivity sind *Eins-zu-Eins* (1:1), *Eins-zu-Viele* (1:n) und *Viele-zu-Viele* (m:n).

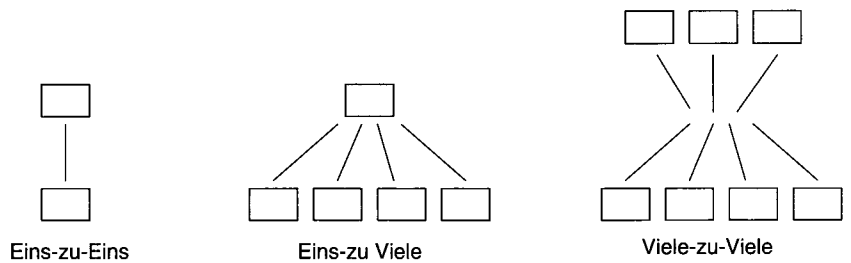



*Bild 8-3* ***Connectivity in Relationships***

Beispielsweise kann eine Adresse mit mehr als einem Namen verbunden sein. Die Entities Name und Adresse haben also eine Eins-zu-Viele Beziehung.

## Existenzabhängigkeit

Dieser Terminus beschreibt, ob eine Entity in einer Beziehung optional ist oder immer verhanden sein muß. Sehen Sie in Ihren Grundannahmen nach, ob eine bestimmte Entity in einer Beziehung vorhanden sein muß. Beispielsweise könnten Sie beschlossen haben, daß eine Adresse immer mit einem Namen verbunden sein muß. In diesem Fall besteht also eine verbindliche Existenzabhängigkeit zwischen den beiden Entities Name und Adresse. Eine optionale Existenzabhängigkeit besteht beispielsweise zwischen dem Namen einer Person und der Frage, ob sie Kinder hat.

## Kardinalität

Kardinalität beschränkt die Anzahl der Häufigkeit, in der eine Entity in einer Beziehung vorkommen darf. Die Kardinalität in einer 1:1 Beziehung ist immer eins. Die Kardinalität einer 1:n Beziehung ist immer offen, da für n jede beliebige Zahl stehen kann. Falls Sie eine Obergrenze für n setzen wollen, tun Sie dies, indem Sie die Kardinalität der Beziehung festlegen. Beispielsweise könnten Sie die Anzahl der Gegenstände beschränken, die ein Kunde zu ei-

nem bestimmten Zeitpunkt bestellen kann. Die Kardinalität legen Sie in der Regel in Ihrem Anwendungsprogramm oder in gespeicherten Prozeduren fest.

Weitere Informationen über Cardinality entnehmen Sie anderem Material über das Entity-Relationship Modell. In der Zusammenfassung dieses Kapitels wird außerdem auf zwei Bücher verwiesen, die sich mit Datenmodellierung beschäftigen.

# Die Beziehungen aufdecken

Der kompakteste Weg für die Aufdeckung der Beziehungen, ist die Vorbereitung einer Matrix; diese gibt sowohl in den Zeilen als auch in den Spalten alle Namen der Entities an. Bild 8-4 zeigt eine Matrix, die die Entities des persönlichen Adreßbuchs enthält:

| | name | adresse | telefon_nr | fax_nr | modem_nr |
|---|---|---|---|---|---|
| name | | | | | |
| adresse | | | | | |
| telefon_nr | | | | | |
| fax_nr | | | | | |
| modem_nr | | | | | |

*Bild 8-4* ***Eine Matrix, die die Objekte eines persönlichen Adreßbuchs enthält***

Sie können den schraffierten Bereich der Matrix ignorieren. Betrachten Sie die diagonalen Zellen; d. h. fragen Sie sich "Welche Beziehung ist zwischen einem *A* und einem anderen *A?*". In diesem Modell lautet die Antwort immer "keine". Es gibt keine Beziehung zwischen einem Namen und einem anderen Namen oder zwischen einer Adresse und einer anderen Adresse, zumindest keine, die in diesem Modell aufzeichnungswert wäre.

Tragen Sie in alle Zellen, für die die Antwort "keine" lautet, `keine` in die Matrix ein. Die Matrix sieht nun wie Bild 8-5 aus.

| | name | adresse | telefon_nr | fax_nr | modem_nr |
|---|---|---|---|---|---|
| name | keine | | | | |
| adresse | | keine | | | |
| telefon_nr | | | keine | | |
| fax_nr | | | | keine | |
| modem_nr | | | | | keine |

*Bild 8-5* ***Eine Matrix, in der sich keine Objekte auf sich selbst beziehen***

Obwohl sich in diesem Modell keine Entities auf sich selbst beziehen, trifft dies bei anderen Modellen nicht immer zu. Ein typisches Beispiel hierzu ist ein Angestellter, der der Vorgesetzte eines anderen Angestellten ist. Ein weiteres Beispiel gibt es bei der industrieller Produktion, wenn eine Entity *Teil* Bestandteil eines anderen Teils ist.

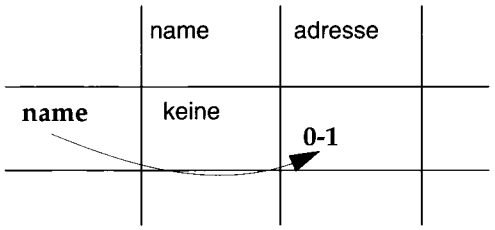
In der Matrix spiegelt sich die Erkenntnis wieder, daß es keine Beziehung zwischen einer Faxnummer und einer Modemnummer gibt.

In die verbleibenden Zellen tragen Sie die Art der Beziehung ein, die zwischen der Entity der Zeile und der Entity der Spalte besteht. Drei Beziehungsarten sind möglich:

- *Eins-zu-Eins* (geschrieben 1:1), wobei es nie mehr als eine einzige Entity *A* für eine einzige Entity *B* gibt, und nie mehr als ein einziges *B* für ein einziges *A*.
- *Eins-zu-Viele* (geschrieben 1:n), wobei es nie mehr als eine einzige Entity *A* gibt, aber es kann mehrere *Entities B* geben, die sich darauf beziehen (oder umgekehrt).
- *Viele-zu-Viele* (geschrieben m:n), wobei sich mehrere *Entities A* auf ein einziges *B* beziehen können und mehrere Entities *B* können sich auf ein einziges *A* beziehen.

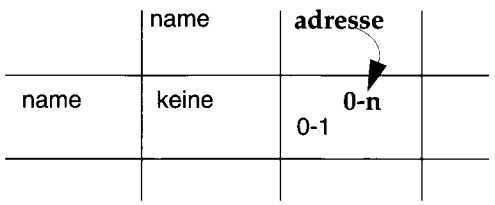
Eins-zu-Viele Beziehungen kommen am häufigsten vor; im Adreßbuch-Modell sind aber Beispiele zu allen drei Beziehungen enthalten.

Wie Sie in Bild 8-5 sehen können, zeigt die erste unausgefüllte Zelle die Beziehung zwischen Namen (name) und Adressen (adresse). Welcher Art ist diese Beziehung? Sie **entscheiden**, daß ein Name keine oder eine einzige Adresse haben kann, aber nicht mehr als eine. Sie schreiben `0-1` neben **name** und unterhalb von **adresse**, so wie im folgenden Diagramm gezeigt:




Eine Adresse kann jedoch mit mehr als einem Namen verbunden sein. Beispielsweise kann man mehrere Personen in einer Firma kennen, oder zwei oder mehr Personen, die an derselben Adresse wohnen.

Kann eine Adresse mit *keinem* Namen verbunden sein? D. h., sollte es möglich sein, daß eine Adresse vorhanden ist, die zu keinem Namen gehört? Sie entscheiden, daß dies möglich sein kann. Unterhalb von **adresse** und neben **name** tragen Sie `0-n` ein, so wie im folgenden Diagramm gezeigt:




Falls Sie entschieden haben, daß eine Adresse nicht vorhanden sein kann, die nicht mit mindestens einem Namen verbunden ist, dann tragen Sie `1-n` anstatt `0-n` ein.

Wenn die Beziehung auf einer der beiden Seiten auf 1 begrenzt ist, dann ist es eine 1:n-Beziehung. Die Beziehung zwischen Namen und Adressen ist eine 1:n-Beziehung.

Betrachten Sie nun die nächste Zelle, die Beziehung zwischen einem Namen und einer Rufnummer (telefon_nr). Mit wievielen Rufnummern kann ein Name verbunden sein, mit einer oder mit mehr als einer? Werfen Sie einen Blick in Ihr Adreßbuch; Sie sehen, daß Sie für eine Person häufig mehr als

eine Telefonnummer notiert haben – bei einem beschäftigten Vertriebsmitarbeiter kennen Sie eine Privatnummer, eine Geschäftsnummer, eine Funktelefonnummer und eine Autotelefonnummer. Es kann aber auch Namen geben, die mit keiner Nummer verbunden sind. Tragen Sie `0-n` neben **name** und unterhalb von **telefon_nr** ein, so wie im folgenden Diagramm gezeigt:

| | name | adresse | telefon_nr |
|---|---|---|---|
| **name** | keine | 0-n<br>0-1 | **0-n** |
| | | | |

Wie ist die andere Seite dieser Beziehung? Wieviele Namen können mit einer Rufnummer verbunden sein? Sie legen fest, daß jeweils nur ein Name mit einer Rufnummer verknüpft werden kann. Kann eine Rufnummer ohne Name gespeichert werden? Sie entscheiden sich für Nein; es gibt keinen Grund, eine Telefonnummer festzuhalten, die von niemandem verwendet wird, den Sie kennen. Tragen Sie `1` unterhalb von **telefon_nr** und neben **name** ein.

| | name | adresse | **telefon_ nr** |
|---|---|---|---|
| name | keine | 0-n<br>0-1 | **1**<br>0-n |
| | | | |

Füllen Sie die restliche Matrix nach demselben Muster aus, wie in Bild 8-6 gezeigt.

| | name | adresse | telefon_nr | fax_nr | modem_nr |
|---|---|---|---|---|---|
| name | keine | 0-n<br>0-1 | 1<br>0-n | 1-n<br>0-n | 1<br>0-n |
| adresse | | keine | keine | keine | keine |
| telefon_nr | | | keine | keine | keine |
| fax_nr | | | | keine | keine |
| modem_nr | | | | | keine |

*Bild 8-6* ***Eine vollständige Matrix für ein Adreßbuch***

- Ein Name kann mit mehr als einer Faxnummer verbunden sein; eine Firma kann beispielsweise mehrere Faxgeräte haben. Auf der anderen Seite kann eine Faxnummer mit mehr als einem Namen verbunden sein; beispielsweise können mehrere Personen dieselbe Faxnummer verwenden.
- Eine Modemnummer muß mit genau einem Namen verbunden sein. (Dies ist eine willkürliche Entscheidung, um das Beispiel komplizierter zu machen; stellen Sie sich vor, daß dies eine Anforderung des Entwurfs ist.) Ein Name kann jedoch mit mehr als einer Modemnummer verbunden sein; beispielsweise kann ein Firmencomputer mehrere Amtsleitungen haben.
- Obwohl es in Wirklichkeit eine Beziehung zwischen einer Rufnummer und einer Adresse, sowie einer Modemnummer und einer Adresse gibt, muß in diesem Modell keine aufgezeichnet werden. Es gibt bereits eine Beziehung zwischen *name* und Rufnummer.

Bei einigen dieser Entscheidungen sind Sie vielleicht anderer Meinung (z. B. darüber, daß eine Beziehung zwischen Ruf- und Modemnummern nicht unterstützt wird). Um unseres Beispiels willen sollen unsere Grundannahmen so aussehen.

### Beziehungen aufzeichnen

Im Abschnitt "Datenobjekte skizzieren" auf Seite 8-19 werden Sie lernen, wie Sie ein Entity-Relationship-Diagramm anfertigen.

## Attribute bestimmen

Entities beinhalten Attribute, also Kennzeichen, Eigenschaften, Bestimmer, Beträge und anderes mehr. Ein Attribut ist ein Fakt oder Stück Information über die Entity, das nicht mehr weiter aufgeschlüsselt werden kann. Wenn wir Entities später als *Tabellen* darstellen, werden Ihre Attribute als *Spalten* verwendet werden.

### Wie gewinnt man Attribute

Bevor Sie Attribute herausfinden können, müssen Sie natürlich die Entities bestimmen. Anschließend sollten Sie sich folgende Frage stellen: „Welche Eigenschaften der jeweiligen Entity muß ich kennen?" In der Entity *Adresse* werden Sie beispielsweise Informationen über die Straße, den Ort und die Postleitzahl benötigen. Alle diese Merkmale der Entity *Adresse* werden zu Attributen.

### Auswahl der Attribute

Bei der Attributauswahl sollten solche ausgesucht werden, die über folgende Eigenschaften verfügen:

- Sie sind wichtig.

  Nehmen Sie nur Attribute auf, die für die Datenbank-Benutzer wichtig sind.

- Sie sind direkt und nicht abgeleitet.

  Ein Attribut, das von bestehenden Attributen abgeleitet werden kann – z. B. anhand eines Ausdrucks in einer SELECT-Anweisung – sollte kein Bestandteil des Modells werden. Das Vorhandensein abgeleiteter Daten macht die Pflege der Datenbank außerordentlich kompliziert.

  In einem späteren Entwicklungsstadium (dies wird im Kapitel 10 erläutert), können Sie das Hinzufügen abgeleiteter Attribute in Betracht zie-

hen, um die Performance zu verbessern; im momentanen Stadium sollten Sie dies ausschließen.

- Sie können nicht mehr weiter zerlegt werden.

  Attribute enthalten nur einzelne Werte, niemals Listen oder sich wiederholende Gruppen. Zusammengesetzte Werte müssen in einzelne Attribute aufgeteilt werden.

- Sie enthalten Daten vom jeweils selben Datentyp.

  In ein Feld mit dem Attribut *geb* dürfen beispielsweise nur Datumswerte eingegeben werden, aber keine Namen oder Telefonnummern.

Die Regeln für die Definition von Attributen sind identisch mit denen für die Definition von Spalten. Weitere Information zum Anlegen von Spalten erhalten Sie im Abschnitt "Spalten mit Constraints belegen" auf Seite 8-24. Wenn wir unserem Beispiel die folgenden Attribute hinzufügen, erhalten wir eine Struktur wie auf Bild 8-11 auf Seite 8-21 gezeigt:

- Straße, Ort, Staat, und ZIP-Code sind in der Entity adresse enthalten.
- Die Entity name erhält die Attribute Geburtstag, E-Mail Adresse, Jubiläum und die Vornamen der Kinder.
- Die Entitiy telefon erhält das Attribut Typ, um zwischen Autotelefon, privatem und geschäftlichem Telefon zu unterscheiden. Eine Telefonnummer kann nur jeweils von einem Typ sein.
- Die Entity fax erhält ein Attribut mit der Information, zu welchen Zeiten das Faxgerät verfügbar ist.
- Die Entity modem enthält ein Attribut mit der Information, ob es 300, 1200 oder 2400 Baud unterstützt.

## Attribute auflisten

Vorläufig genügt es, wenn Sie einfach die Attribute für jede einzelne Entity auflisten. Ihre Liste sollte ungefähr folgendermaßen aussehen:

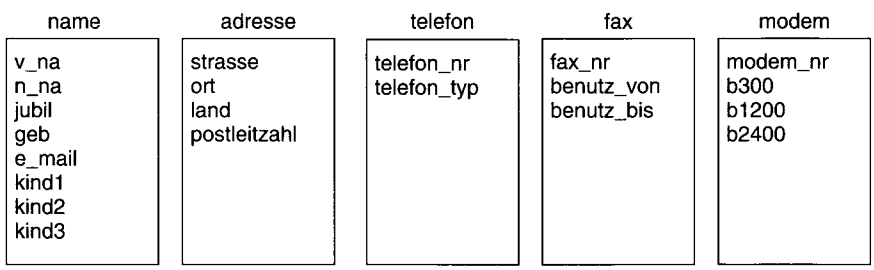



*Bild 8-7 Attribute unseres Adressbuchbeispiels*

## Die einzelnen Realisierungen der Entity

Die Realisierungen der Entity stellen ein weiteres Datenobjekt dar, das Sie kennen sollten. Jeder Datensatz einer Tabelle stellt eine Realisierung der Entity dar. Wenn die Entity customer in einer Tabelle customer dargestellt wird, enthält jeder Datensatz einen spezifischen Kunden, beispielsweise Sue Smith. Aus Entities werden also Tabellen und aus Attributen werden Spalten. Die einzelnen Realisierungen der Entity entsprechen den einzelnen Datensätzen.

# Datenobjekte skizzieren

Jetzt haben Sie die Konzepte *Entity* und *Relationship* kennengelernt. Dies sind die entscheidenden Punkte beim Entwerfen einer Datenbank. Sobald Sie Entities und Relationships festgelegt haben, wird es Ihnen helfen, mit einer entsprechenden Technik Ihre Gedanken zu skizzieren.

Praktisch alle Arten der Datenmodellierung bedienen sich in irgendeiner Form der grafischen Darstellung von Entities und Beziehungen. Das hier vorgestellte Entity-Relationship Diagramm dient dabei den folgenden Zwekken:

- Es bildet den Informationsbedarf einer Organisation ab.
- Es stellt Entities und ihre Beziehungen dar.
- Es liefert einen Ausgangspunkt für die Datendefinition (Datenfluß-Diagramm).
- Für Anwendungsentwickler, Datenbank- und Systemverwalter ist es eine ideale Informationsquelle.
- Es schafft den logischen Aufbau der Datenbank, der dann in eine physikalische Form gebracht wird.

Es gibt verschiedene Arten des Entity-Relationship Diagramms. Falls Sie bereits eine seiner Varianten verwenden, sollten Sie dies auch weiterhin tun. Die folgenden Symbole stellen ein einfaches Entity-Relationship Diagramm dar:

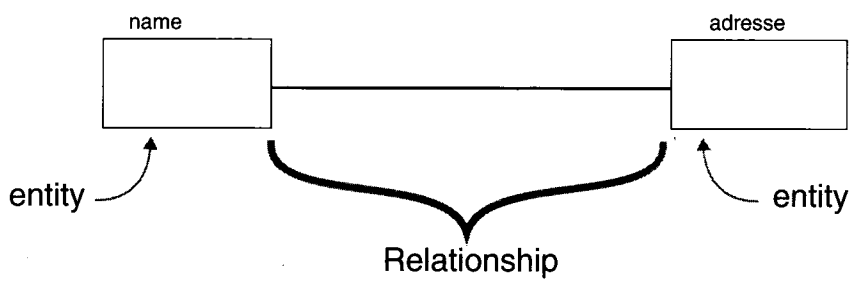



*Bild 8-8* ***Symbole in einem Entity-Relationship-Diagramm***

Entities werden als Rechtecke dargestellt, Beziehungen als Linien zwischen ihnen. Um die einzelnen Merkmale einer Beziehung darzustellen, verwenden Sie die folgenden Symbole:

- Ein Kreis in einer Beziehungslinie zeigt an, daß diese *optional* ist.
- Ein Querstrich in einer Beziehungslinie zeigt an, daß genau ein Exemplar der Entity in Beziehung zu der anderen Entity steht (Sie können sich den Querstrich als 1 vorstellen.).
- Der Krähenfuß zeigt an, mehrere Exemplare der Entity in Beziehung zu der anderen Entity stehen können.

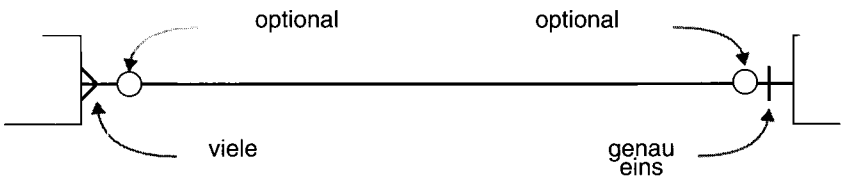



*Bild 8-9* ***Teile einer Relationship in einem Entity-Relationship-Diagramm***

## Entity-Relationship Diagramme lesen

Sie lesen die Diagramme zunächst von links nach rechts, und dann von rechts nach links. Die in Bild 8-10 dargestellte name-adresse Beziehung wird folgendermaßen gelesen: Namen können mit keiner oder genau einer Adresse verbunden sein. Adressen können mit keiner, einer oder mit beliebig vielen Namen verbunden sein.

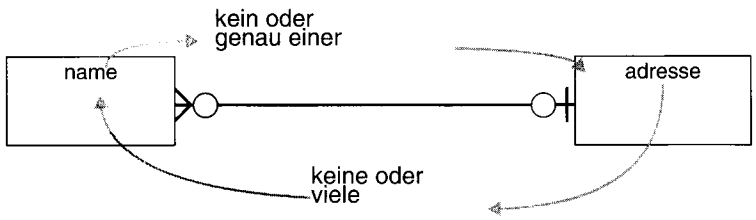



*Bild 8-10* ***Ein Entity-Relationship-Diagramm lesen***

## Ein Beispiel

Das folgende Entity-Relationship Diagramm zeigt das Adreßbuch-Beispiel mit Entities, Relationships und Attributen. Es zeigt die Beziehungen, die wir vorher in einer Matrix dargestellt haben. Vergleichen Sie die Beziehungssymbole in Bild 8-11 mit der Matrix in Bild 8-6 auf Seite 8-15. Vergewissern Sie sich, daß in beiden Abbildungen die gleichen Beziehungen dargestellt sind.

Eine Matrix wie in Bild 8-6 macht Sinn, wenn Sie den ersten Entwurf Ihres Modells anfertigen und dabei Entities, Beziehungen und Attribute festlegen. Sie werden durch die Matrix nämlich gezwungen, jede mögliche Beziehung zu überdenken. In Diagrammen wie in Bild 8-11 auf Seite 8-21 sind aber die gleichen Beziehungen dargestellt, wobei diese Art der Darstellung wesentlich leichter zu lesen ist.

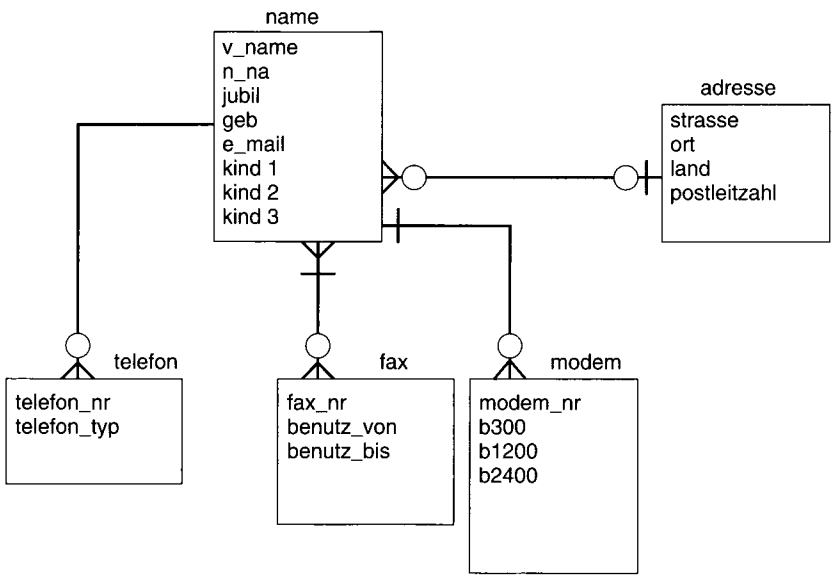



*Bild 8-11* ***Vorläufiges Entity-Relationship Diagramm unserer Beispieldatenbank***

### Und was kommt jetzt?

Im folgenden lernen Sie, wie Sie

- Entities, Relationships und Attribute in relationale Form bringen,
- das Entity-Relationship Modell reorganisieren,
- das Entity-Relationship Modell in *Normalform* bringen.

Im Kapitel 9 erfahren Sie, wie Sie Ihre Datenbank aus dem Entity-Relationship Modell aufbauen.

# Objekte des Entity-Relationship Modells in relationale Form bringen

Alle Datenobjekte, die Sie bisher kennengelernt habern - also Entities, Relationships, Attribute und einzelne Realisierungen von Entities - werden jetzt in SQL-Tabellen, Joins zwischen Tabellen, Spalten und Datensätze verwandelt. Tabellen, Spalten und Datensätze Ihrer Datenbank müssen den Regeln entsprechen, die im folgenden Abschnitt "Regeln für die Erstellung von Tabellen, Sätzen und Spalten" auf Seite 8-23 beschrieben sind.

Die Datenobjekte sollten diesen Regeln entsprechen, bevor Sie sie in *Normalform* bringen. Bei der Normalisierung analysieren Sie die Abhängigkeiten zwischen Ihren Entities, Beziehungen und Attributen. Mehr über Normalisierung erfahren Sie im Abschnitt "Das Datenmodell normalisieren" auf Seite 8-32.

Nach der Normalisierung können Sie Ihre Datenbank mit SQL-Anweisungen aufbauen. Dies wird im Kapitel 9 beschrieben. Dort wird außerdem das Datenbankschema für unser Beispiel dargestellt.

Jede Ihrer Entities steht für eine Tabelle in Ihrem Modell. Die Tabelle ist das *abstrakte* Konzept der Entity, während die einzelnen Datensätze jeweils eine *konkrete* Ausprägung der Entity darstellen. Jedes Attribut einer Entity entspricht einer Spalte in der Tabelle.

Die folgenden Ideen bilden die Grundlage von praktisch allen Methoden der relationalen Datenmodellierung, das Entity-Relationship Modell eingeschlossen. Wenn Sie sich beim Design Ihres Datenmodells an diese Regeln halten, werden Sie bei der Normalisierung Zeit und Mühe sparen.

## Regeln für die Erstellung von Tabellen, Sätzen und Spalten

Sie sind bereits mit dem Konzept einer *Tabelle*, die sich aus *Sätzen* und *Spalten* zusammensetzt, vertraut. Beim Festlegen der Tabellen für ein formales Datenmodell müssen Sie aber vier Regeln beachten:

- Sätze müssen alleine stehen.

  Jeder Satz einer Tabelle ist unabhängig; er hängt nicht von einem anderen Satz *derselben* Tabelle ab. Daraus folgt, daß die Reihenfolge der Sätze in einer Tabelle nicht wichtig für das Modell ist. Das Modell sollte selbst dann noch richtig sein, wenn alle Sätze der Tabelle in eine willkürliche Reihenfolge gebracht werden.

  Nachdem die Datenbank eingerichtet wurde, können Sie den Datenbankserver anweisen, die Sätze aus Gründen der Effizienz in einer bestimmten Reihenfolge zu speichern; dies hat jedoch keine Auswirkung auf das Modell.

- Sätze müssen eindeutig sein.

  In jedem Satz muß eine bestimmte Spalte einen eindeutigen Wert enthalten. Wenn keine einzelne Spalte diese Eigenschaft erfüllt, dann müssen die Werte einer bestimmten Gruppe von Spalten als Ganzes in jedem Satz unterschiedlich sein.

- Spalten müssen alleine stehen.

  Die Reihenfolge der Spalten in einer Tabelle hat für das Modell keine Bedeutung. Das Modell sollte selbst dann noch richtig sein, wenn die Spalten umgestellt werden.

  Nachdem die Datenbank eingerichtet wurde, hängt die Ausführung der Programme und gespeicherten Abfragen, die einen Stern (*) verwenden, der für *alle Spalten* steht, von der endgültigen Reihenfolge der Spalten ab; dies wirkt sich jedoch nicht auf das Modell aus.

- Spaltenwerte müssen einheitlich sein.

  Eine Spalte kann nur einzelne Werte enthalten, niemals Listen oder wiederholte Gruppen. Zusammengesetzte Werte müssen in verschiedene Spalten aufgeteilt werden. Wenn Sie z. B. den Vor- und Nachnamen einer Person als getrennte Werte behandeln - wie wir dies in diesem Beispiel auch tun - , müssen diese Werte in verschiedenen Spalten stehen und nicht in einer einzigen Spalte *name*.

- Jede Spalte muß einen eindeutigen Namen haben

  Zwei Spalten in derselben Tabellen können nicht denselben Namen haben. Beispielsweise enthält die Tabelle *name* in unserem Adreßbuch-

Beispiel Spalten für die Namen von Kindern. Sie können diese Spalten kind1, kind2, kind3 etc. nennen.

- Jede Spalte kann nur Daten eines einzigen Datentyps aufnehmen

  Eine Spalte darf immer nur Daten eines einzigen Datentyps enthalten. Eine Spalte vom Datentyp INTEGER darf beispielsweise nur numerische Daten enthalten, aber keine Buchstaben.

Falls Sie bisher nur mit Daten gearbeitet haben, die in Arrays oder sequentiellen Dateien abgelegt sind, dann kommen Ihnen diese Regeln vielleicht unnatürlich vor. Die Theorie über relationale Datenbanken zeigt jedoch, daß man durch die ausschließliche Verwendung von Tabellen, Sätzen und Spalten, die diesen Regeln entsprechen, alle Datentypen darstellen kann. Mit etwas Übung wendet man diese Regeln automatisch an.

## Spalten mit Constraints belegen

Wenn Sie mit der CREATE TABLE Anweisung Tabellen und Spalten erstellen, versehen Sie jede Spalte mit Constraints. Diese Constraints geben an, ob eine Spalte Zeichen und Ziffern enthalten darf, in welchem Format Daten eingegeben werden dürfen und anderes mehr. Ein Wertebereich gibt an, welche Werte in einer Spalte zulässig sind. Der Wertebereich einer Spalte beinhaltet die folgenden Punkte:

- Datentyp (INTEGER, CHAR, DATE usw.)
- Format (beispielsweise *tt/mm/jjjj* bei der Datumseingabe)
- Wertebereich (beispielsweise 1000-5000)
- Bedeutung (beispielsweise die Personalnummer)
- Zulässige Werte (beispielsweise nur *w* oder *m* für „männlich" oder „weiblich".)
- Eindeutigkeit
- Zulässigkeit von NULL-Werten
- Standardwerte
- Bedingungen zur Wahrung der referentiellen Integrität

Wertebereiche geben Sie an, wenn Sie die Tabelle erstellen. Diese Punkte werden im Kapitel 9 besprochen.

# Schlüssel für Tabellen festlegen

Spalten in Tabellen sind entweder *Schlüssel* oder *Deskriptoren*. Ein Schlüssel ist eine Spalte, über die ein Datensatz der Tabelle eindeutig definiert wird. Beispielsweise ist die Sozialversicherungsnummer ein eindeutiger Schlüssel für jeden Arbeitnehmer. Deskriptor-Spalten beinhalten die *nicht-eindeutigen* Informationen in einem Datensatz. Beispielsweise können zwei Angestellte den gleichen Vornamen Sue haben. Der Vorname Sue ist also ein nicht-eindeutiges Merkmal eines Angestellten. In Tabellen gibt es in erster Linie zwei Arten von Schlüsseln:

Primärschlüssel

Fremdschlüssel

Beim Erzeugen von Tabellen definieren Sie Primär- und Fremdschlüssel. Primär- und Fremdschlüssel schaffen eine physikalische Verbindung zwischen den einzelnen Tabellen. Dabei müssen Sie nach einer Spalte oder einer Kombination aus mehreren Spalten suchen, mit der jeder Datensatz eindeutig von jedem anderen unterschieden werden kann.

## Primärschlüssel

Die Spalte, deren Werte in jedem Satz unterschiedlich sind, stellt den *Primärschlüssel* einer Tabelle dar. Da die Werte unterschiedlich sind, machen sie jeden Satz eindeutig. Gibt es keine solche Spalte, so ist der Primärschlüssel aus zwei oder mehr Spalten *zusammengesetzt*, deren Werte (zusammengenommen) in jedem Satz unterschiedlich sind.

In einem Modell muß jede Tabelle über einen Primärschlüssel verfügen. Dies ergibt sich automatisch aus der Regel, daß alle Sätze eindeutig sein müssen. Falls erforderlich, ist der Primärschlüssel aus allen Spalten zusammengesetzt.

Der Primärschlüssel sollte entweder einen numerischen Datentyp aufweisen (INT der SMALLINT), SERIAL oder ein kurzes Feld vom Typ CHAR (wie man es für Codes vergibt) sein. Felder mit langen Zeichenketten sind für Primärschlüssel nicht zu empfehlen.

NULL-Werte sind in einer Primärschlüssel-Spalte niemals erlaubt. NULL-Werte sind nicht vergleichbar, d. h. es kann nicht gesagt werden, ob sie gleich oder unterschiedlich sind. Aus diesem Grund können sie einen Satz sich nicht eindeutig von den anderen Sätzen unterscheiden lassen. Wenn in einer Spalte NULL-Werte erlaubt sind, dann kann sie kein Teil eines Primärschlüssels sein.

### Was sind Null-Werte

Spalten können Null-Werte enthalten. Dies bedeutet, daß der Wert, der in die Spalte eingetragen werden soll, unbekannt ist oder nicht eingegeben werden kann. Beispielsweise kann die Spalte **kind1** Null-Werte aufnehmen; falls eine Person keine Kinder hat oder *man nicht weiß, ob Kinder vorhanden sind*, muß es die Möglichkeit geben, diese Tatsache in die Datenbank einzutragen.Verwechseln Sie den NULL-Wert nicht mit der Zahl 0 oder mit Leerzeichen.

Für einige Entities werden eigene Primärschlüssel erzeugt, beispielsweise die Artikelnummern in Katalogen, die außerhalb des Modells definiert werden. Sie sind benutzer-definiert (user-assigned).

Manchmal können mehr als eine Spalte bzw. mehr als eine Gruppe von Spalten als Primärschlüssel verwendet werden. Solche Spalten oder Gruppen werden *Kandidatenschlüssel* genannt.

Alle Kandidatenschlüssel sind eindeutig und damit von besonderer Bedeutung, denn diese Eigenschaft macht das Ergebnis einer SELECT-Operation vorhersehbar. Wenn man die Spalten eines Kandidatenschlüssels auswählt, weiß man, daß das Ergebnis keine doppelten Sätze enthalten kann. Manchmal ist es auch möglich, vorherzusagen, wieviele Sätze geliefert werden. Es bedeutet immer, daß das Ergebnis eine eigene richtige Tabelle sein kann, mit dem ausgewählten Kandidatenschlüssel als Primärschlüssel.

### Zusammengesetzte Schlüssel

Einige Entities haben keine Merkmale, die zuverlässig eindeutig sind. Beispielsweise können verschiedene Personen verschiedene Namen und verschiedene Bücher verschiedene Titel haben. In der Regel ist es möglich, eine Kombination aus Spalten zu finden, die als Primärschlüssel verwendet werden kann. So kommt es kaum vor, daß Personen mit dem gleichen Namen an der gleichen Adresse wohnen, oder daß verschiedene Bücher die gleiche Kombination aus Titel, Autoren und Erscheinungsjahr haben.

### Vom System zugewiesene Schlüssel

Ein vom System zugewiesener Primärschlüssel ist einem zusammengesetzten Schlüssel in der Regel vorzuziehen. Ein vom System zugewiesener Primärschlüssel ist eine Zahl (oder ein Code), die jeder Ausprägung eines Objekts zugeordnet wird, wenn sie zum ersten Mal in die Datenbank eingetragen wird. Typische Beispiele sind Bestell- oder Kundennummern. Am ein-

fachsten geht die Implementierung von Schlüsseln, die vom System zugewiesen werden sollen, mit SERIAL-Zahlen, denn diese kann das System automatisch erzeugen. Bei Informix existiert der Datentyp SERIAL, mit dem man leicht eine solche Spalte erzeugen kann. Es kann allerdings vorkommen, daß Datenbank-Benutzer einen einfachen numerischen Code ungern benutzen. Andere Codes könnten auf dem aktuellen Datum basieren; z. B. könnte die Identifikationsnummer eines Angestellten aus den Initialen der Person, kombiniert mit dem Eintrittsdatum, bestehen. In unserer Beispieldatenbank wird in der Tabelle **name** ein vom System generierten Primärschlüssel verwendet.

## Fremdschlüssel (Join-Spalten)

Ein *Fremdschlüssel* ist einfach eine Spalte (oder eine Gruppe von Spalten) in einer einzigen Tabelle, die Werte enthält, die mit dem *Primärschlüssel* einer anderen Tabelle übereinstimmen. Fremdschlüssel werden zur Verknüpfung von Tabellen verwendet; tatsächlich sind die meisten der *Join-Spalten*, die in diesem Handbuch bisher angesprochen wurden, Fremdschlüssel-Spalten. Bild 8-12 zeigt Primär- und Fremdschlüssel der Tabellen *customer* und *orders* aus der Beispieldatenbank stores6:

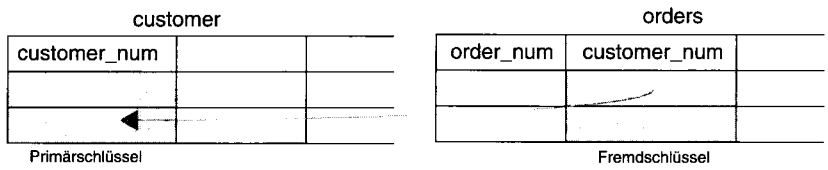



*Bild 8-12* ***Primär- und Fremdschlüssel in der Beziehung customer-orders***

Fremdschlüssel sollen bei jedem Auftreten im Modell vermerkt werden, weil ihr Vorhandensein die Möglichkeit einschränkt, Sätze aus Tabellen zu löschen. Bevor man einen Satz sicher löschen kann, muß man alle Sätze löschen, die mit dem Satz über Fremdschlüssel verbunden sind. Beispielsweise sollte es nicht möglich sein, in der Tabelle **customer** einen Kunden zu löschen, zu dem in der Tabelle **orders** bereits Aufträge existieren! Diese Aufträge würden dann nämlich ohne Besteller „ in der Luft hängen", da es zu der Kundennummer in den Aufträgen keine Kundendaten in der Tabelle **kunde** mehr gibt..

Mit der Anweisung ON DELETE CASCADE können Sie allerdings erreichen, daß *eine* Löschanweisung ausreicht, um den zu löschenden Datensatz und alle mit ihm über Fremdschlüssel verknüpften Datensätze zu löschen.

Die Datenbank verhindert Löschvorgänge, bei denen die referentielle Integrität verletzt wird. Man kann die referentielle Integrität immer aufrechterhalten, indem man alle Fremdschlüssel-Sätze löscht, bevor man den Primärschlüssel löscht, auf den sich die Fremdschlüssel beziehen. Wenn man bei der Erstellung der Datenbank mit dem Schlüsselwort REFERENCES Fremdschlüsselbeziehungen definiert, läßt es die Datenbank selbst nicht zu, Primärschlüssel zu löschen, für die es übereinstimmende Fremdschlüssel gibt. Dies verhindert auch, daß man einen Fremdschlüssel hinzufügt, der sich nicht auf einen bestehenden Wert eines Primärschlüssels bezieht. Es ist dann beispielsweise nicht möglich, eine Bestellung für einen Kunden aufzunehmen, solange der Kunde selbst nicht in die Datenbank eingegeben worden ist. Weiteres zum Thema referentielle Integrität erfahren Sie im Kapitel 4.

## Das Beispieldiagramm mit Schlüsseln versehen

Die erste Auswahl an Primär- und Sekundärschlüsseln wird in Bild 8-13 gezeigt. Dem Diagramm liegen einige wichtige Entscheidungen zugrunde.

In der Tabelle **name** wird die Spalte **satz_nr** als Primärschlüssel verwendet. Sie bekommt den Datentyp SERIAL und wird automatisch vom System erzeugt. Wenn Sie sich die anderen Spalten bzw. Attribute dieser Tabelle ansehen, werden Sie feststellen, daß diese häufig vom Typ CHAR und daher als Primärschlüssel nicht besonders geeignet sind. Und wenn Sie mehrere Spalten einer Tabelle zu einem zusammengesetzten Schlüssel kombinieren, wird dadurch nur das weitere Vorgehen kompliziert gemacht. Durch Verwendung des Datentyps SERIAL haben wir einen Primärschlüssel erzeugt, mit dem wir leicht andere Tabellen mit der Tabelle **name** verknüpfen können. (In den verknüpften Tabellen wird das Feld **satz_nr** als Fremdschlüssel verwendet. Hätten wir einen zusammengesetzten Primärschlüssel in der Tabelle name verwendet, wäre die Verknüpfung mit anderen Tabellen kompliziert geworden, da wir dann in all diesen Tabellen alle Felder des zusammengesetzten Schlüssels als Fremdschlüssel hätten aufnehmen müssen.)

Als weitere Primärschlüssel tauchen Telefonnummern in den Tabellen **telefon**, **fax** und **modem** auf. Die Tabelle **adresse** muß einen eigenen Primärschlüssel bekommen, da nach unseren Grundannahmen auch eine Adresse existieren kann, zu der man keinen Ansprechpartner hat. Wenn unsere Grundannahmen fordern würden, daß es keine Adresse ohne Ansprechpartner geben dürfte, dann könnte man die Tabelle **adresse** über den Fremdschlüssel **satz_nr** mit der Tabelle **name** verknüpfen.

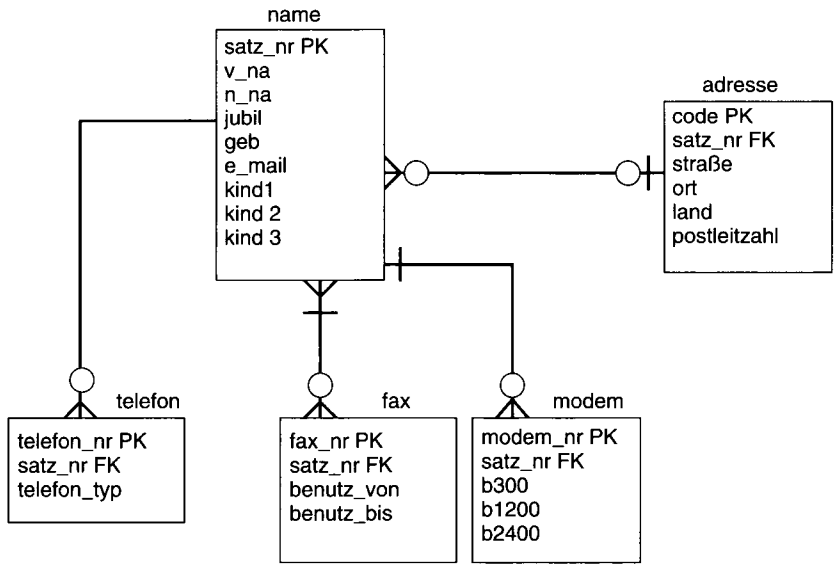



*Bild 8-13 Struktur unserer Beispieldatenbank mit Primär (PK)- und Fremdschlüsseln (FK)*

# Reorganisation der Beziehungen

Ein gutes Datenmodell soll eine Struktur liefern, die dem Datenbankserver einen schnellen Zugriff ermöglicht. Unser Modell kann weiter verbessert werden, indem die Beziehungen *reorganisiert* und das Datenmodell *normalisiert* wird. In diesem Abschnitt erfahren Sie alles zum Thema *Reorganisation*. Die Normalisierung wird im Abschnitt "Das Datenmodell normalisieren" auf Seite 8-32 besprochen.

## Reorganisation einer m:n Beziehung

*Viele-zu-Viele* Beziehungen machen Ihr Modell komplex und erzeugen Probleme bei der Anwendungsentwicklung. Das Problem ist nicht nur die Komplexität, sondern vor allem die Tatsache, daß man Daten mehrfach eingeben muß, was bei einem anderen Design nicht nötig wäre. Wenn beispielsweise mehrere Personen die gleiche Adresse haben, dann ist es besser, einen Adresscode zu vergeben und mit jeder Person nur den Adresscode zu speichern. Andernfalls muß für jede Person die volle, identische Adresse erfaßt werden. Es geht also darum, Redundanzen zu vermeiden. Der Ansatz, Viele-

zu-Viele Beziehungen aufzulösen, besteht darin, die beiden Entities zu isolieren und statt dessen zwei *Eins-zu-Viele* Beziehungen zwischen Ihnen und einer dritten *Schnitt-Entitiy* zu schaffen. Die Schnitt-Entity enthält in der Regel Attribute aus den beiden zu verknüpfenden Entities.

Um eine Viele-zu-Viele Beziehung aufzulösen, müssen Sie nochmal Ihre Grundannahmen studieren. Haben Sie die Beziehungen genau aufgezeichnet? In unserem Beispiel haben wir eine Viele-zu-Viele Beziehung zwischen den Entities *name* und *fax*, wie Sie in Bild 8-13 auf Seite 8-29 sehen. Um diese Beziehung aufzulösen, studieren wir noch einmal unsere Grundannahmen. Wir haben folgendes beschlossen: „Eine Person kann keine, eine oder mehrere Faxnummern haben. Eine Faxnummer kann mehreren Personen zugeordnet werden." Wir haben hier also tatsächlich eine Viele-zu-Viele Beziehung.

Ein weiteres Problem haben wir in der *fax* Entity: Die Telefonnummer, die wir als Primärschlüssel für die *telefon* Entity vorgesehen haben, kann dort mehr als einmal vorkommen; dadurch wird die referentielle Integrität verletzt. Wie Sie wissen, muß der Primärschlüssel immer eindeutig sein.

Wir reorganisieren diese Viele-zu-Viele Beziehung, indem wir zwischen den Entities *name* und *fax* eine Schnitt-Entity einführen. Die neue Schnitt-Entity mit dem Namen *faxname* enthält zwei Attribute, nämlich *fax_nr* und *satz_nr*. Der Primärschlüssel dieser Entity ist aus diesen beiden Attributen zusammengesetzt. Für sich gesehen ist jedes der beiden Attribute ein Fremdschlüssel: Zwischen den Tabellen *name* und *faxname* besteht eine Eins-zu-Viele Beziehung, da eine Name mit mehreren Faxnummern verknüpft werden kann. Umgekehrt kann jede Kombination in faxname mit einem **satz_nr** verbunden werden. Zwischen den Tabellen *fax* und *faxname* besteht eine Eins-zu-Viele Beziehung, da jede Nummer mit mehreren Kombinationen aus *faxname* verknüpft werden kann.

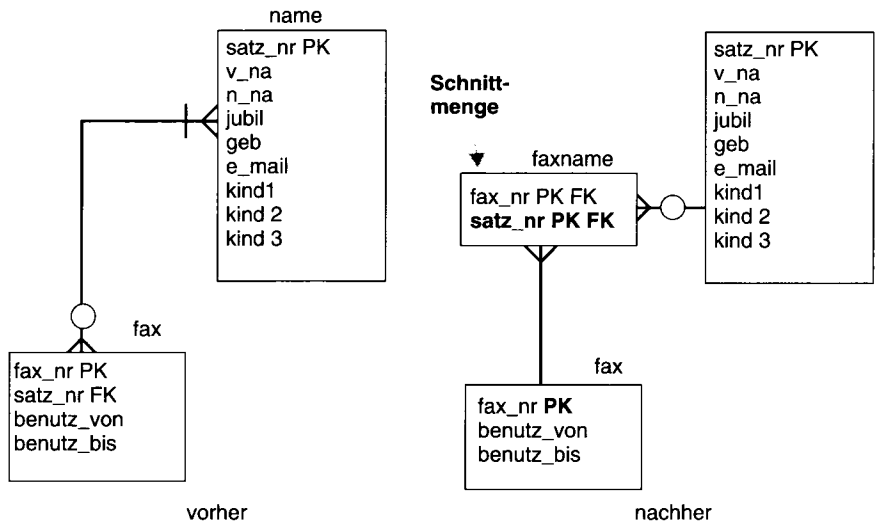



*Bild 8-14* *Auflösen einer Eins-zu-Viele Beziehung*

## Weitere besondere Beziehungen reorganisieren

Ihnen können weitere Spezialbeziehungen begegnen, die den schnellen Ablauf einer Datenbank behindern können. Dazu gehören:

- Komplexe Beziehungen
- Rekursive Beziehungen
- Redundante Beziehungen

Eine *komplexe* Beziehung besteht zwischen drei oder mehr Entities. Alle Entities müssen vorhanden sein, damit die Beziehung existiert. Um die Komplexität zu reduzieren, sollten Sie alle komplexen Beziehungen in eine Entity bringen, die binäre Beziehungen zu jeder der ursprünglichen Entities hat.

*Rekursive* Beziehungen bestehen zwischen Entities des gleichen Typs. Sie kommen nicht besonders häufig vor. Beispiele sind Stücklisten (Teile bestehen häufig aus mehreren anderen Teilen) oder Organisationen (Angestellte werden von anderen Angestellten geführt). Kapitel 5 enthält ein vertieftes Beispiel einer rekursiven Beziehung. Rekursive Beziehungen werden Sie möglicherweise nicht umstrukturieren.

Eine *redundante* Beziehung liegt dann vor, wenn zwei oder mehr Beziehungen dafür verwendet werden, ein und dasselbe Konzept darzustellen. Redundante Beziehungen machen das Modell komplexer und verleiten Entwickler unter Umständen dazu, Attribute an den falschen Stellen im Modell einzufügen. Redundante Beziehungen können als doppelte Einträge in Ihrem Entity-Relationship-Diagramm erscheinen. Beispielsweise könnten Sie zwei Entities mit den gleichen Attributen haben. Um redundante Beziehungen zu finden, sollten Sie nochmal Ihr Datenmodell durchsehen. Haben Sie mehr als eine Entity mit denselben Attributen? Möglicherweise müssen Sie eine Entity hinzufügen, um die Redundanz aufzulösen. In Kapitel 10 erhalten Sie weitere Information zum Thema „Redundanz" im Datenmodell.

# Das Datenmodell normalisieren

Der bisherige Stand unseres Beispiels scheint soweit in Ordnung zu sein. Es könnte in eine Datenbank umgesetzt werden; allerdings könnte es später bei der Entwicklung von Anwendungen und bei der Operationen mit der Datenbank Probleme geben. Die Normalisierung ist ein formaler Ansatz, bei dem einige Regeln auf Attribute und Entities angewendet werden.

Normalisierung Ihres Modells kann zu folgendem führen:

- Größere Flexibilität Ihres Designs
- Speicherung der Attribute in den richtigen Tabellen
- Verhindern von Redundanz
- Steigerung der Effizienz der Programmierer
- Verringerung der Kosten der Programmwartung
- Optimale Stabilität der Datenstruktur.

Die Normalisierung besteht aus mehreren Schritten, bei denen die Gestalt der Entities schrittweise optimiert wird. Diese Schritte heißen Normalisierungsregeln. Es gibt mehrere Stufen der *Normalform*. In diesem Kapitel werden die ersten drei Normalformen vorgestellt. Jede Stufe der Normalform führt zu einer strukturierteren Form des Datenmodells als ihre Vorgängerin. Daher müssen Sie zuerst die erste Stufe der Normalform erreichen, dann die zweite usw.

## Erste Stufe der Normalform

Eine Entity entspricht dann der ersten Normalisierungstufe, wenn sie keine sich wiederholenden Gruppen beinhaltet. Dies bedeutet, daß sie keine sich wiederholenden *Spalten* beinhalten darf. Diese machen Ihre Datenbank unflexibel, verschwenden Speicherplatz und machen Suchoperationen kompliziert. In unserem Beispiel enthält die Tabelle *name* sich wiederholende Spalten, nämlich *kind1*, *kind2* und *kind3*.

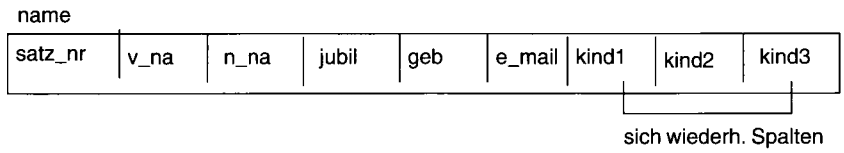



*Bild 8-15* ***Entity name vor der ersten Normalisierung***

In dieser Tabelle können einige Probleme auftreten. Auf der Platte wird jeweils Platz für drei Kinder reserviert, ganz gleich, ob die Person Kinder hat oder nicht. Die Höchstzahl an Kindern ist auf drei beschränkt; Einige Ihrer Bekannten haben aber vielleicht vier oder mehr Kinder. Und wenn Sie nach einem bestimmten Kind suchen, müssen Sie für jeden Datensatz alle drei Spalten durchsuchen.

Sie können die sich wiederholenden Spalten beseitigen und die erste Stufe der Normalform herstellen, indem Sie die Tabelle in zwei Tabellen aufteilen, wie Bild 8-16 auf Seite 8-33 zeigt. Die Verbindung zwischen den beiden Tabellen erfolgt über eine Primär- Fremdschlüsselbeziehung. Da es in der Tabelle *kind*keinen Eintrag ohne zugehörigen Datensatz in der Tabelle *name* geben kann, können wir die Verbindung über den Fremdschlüssel **satz_nr** in der Tabelle *kind* aufbauen.

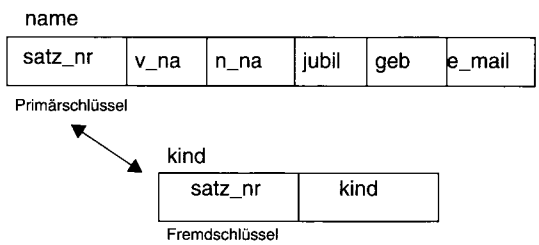



*Bild 8-16* ***Die Entity name in der ersten Normalisierungsstufe***

Dazu kommt noch, daß die Beziehung zwischen *name* und *modem* noch nicht in der ersten Normalisierungsstufe ist, da die Spalten b300, b1200 und b2400 sich wiederholen können. Diese Beziehung wird auf ähnliche Art und Weise normalisiert wie die *name-kind* Beziehung, wie man in Bild 8-17 sehen kann. Diese Abbildung zeigt das gesamte Datenmodel in der ersten Normalisierungsstufe.

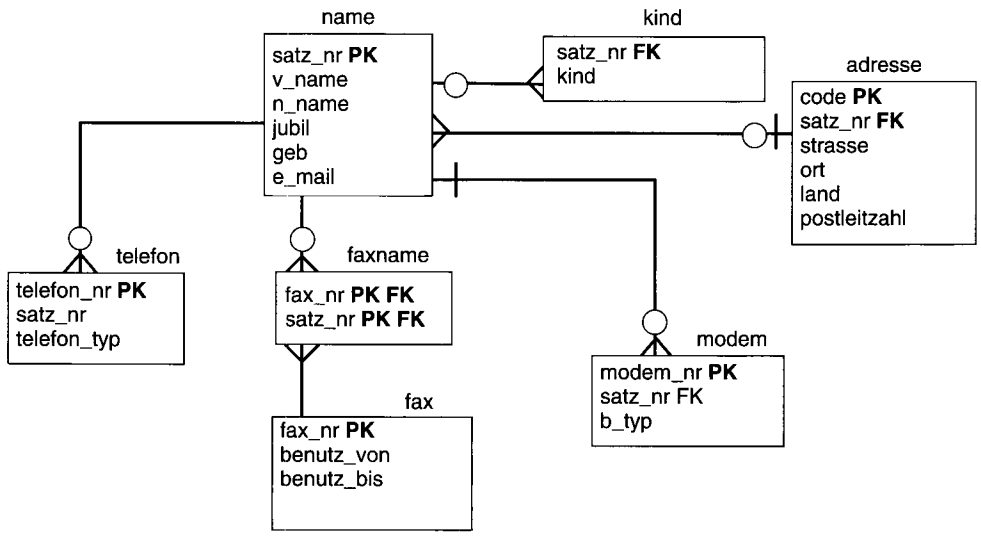



*Bild 8-17* ***Das Datenmodell unserer Beispieldatenbank (PK = Primärschlüssel, FK = Fremdschlüssel)***

## Zweite Stufe der Normalform

Eine Entity entspricht dann der zweiten Normalisierungsstufe, wenn die erste Stufe erreicht ist und außerdem alle Attribute in ihr vom Primärschlüssel abhängen. Dies bedeutet, daß alle Spalten in der Tabelle vom Primärschlüssel *funktional abhängig* sind.

Der Wert eines Attributes hängt dann von einer Spalte ab, wenn bei einem Wechsel des Spalteninhalts automatisch der Wert des Attributes wechselt. Das Attribut ist eine Funktion der Spalte.

- Wenn die Tabelle einen einspaltigen Primärschlüssel hat, muß das Attribut von diesem Schlüssel abhängen.
- Wenn die Tabelle einen zusammengesetzten Schlüssel hat, muß der Wert des Attributes von allen Spalten des zusammengesetzten Schlüssels und nicht nur von einigen abhängen.
- Wenn ein Attribut auch von anderen Spalten abhängt, müssen diese zu einem Kandidatenschlüssel gehören, der in jeder Spalte eindeutig ist.

Wenn Sie Ihr Modell nicht auf die zweite Stufe der Normalform bringen, führt dies zu Datenredundanz und Problemen beim Aktualisieren von Daten. Wandeln Sie Tabellen der ersten Normalisierungsstufe zu solchen der zweiten Normalisierungsstufe um, indem Sie Spalten entfernen, die nicht vom Primärschlüssel abhängen.

## Dritte Stufe der Normalform

Eine Entity ist dann auf der zweiten Stufe der Normalform, wenn die zweite Stufe erreicht ist und wenn all ihre Attribute nicht transitiv abhängig vom Primärschlüssel sind. *Transitiv abhängig* bedeutet, daß *Deskriptor-Spalten* (siehe Seite 8-25) nicht nur vom Primärschlüssel abhängen, sondern auch von anderen Deskriptor-Spalten, die wiederum vom Primärschlüssel abhängen. Im Klartext bedeutet dies, daß keine Spalte eine Tabelle von einer Deskriptor-Spalte abhängen darf, die vom Primärschlüssel abhängt.

Um diese Stufe zu erreichen, müssen Sie Spalten entfernen, die von anderen Deskriptor-Spalten abhängen.

## Zusammenfassung der Normalisierungsregeln

In diesem Kapitel wurden drei Normalisierungsregeln vorgestellt:

- Erste Stufe der Normalform: Eine Tabelle ist dann in der ersten Stufe der Normalform, wenn sie keine sich wiederholenden Spalten enthält.
- Zweite Stufe der Normalform. Eine Tabelle entspricht dann der zweiten Stufe der Normalform, wenn sie die Bedingungen der ersten Stufe erfüllt und nur Spalten enthält, die vom Primärschlüssel abhängen.
- Dritte Stufe der Normalform: Eine Tabelle entspricht dann der dritten Stufe der Normalform, wenn die Bedingungen der ersten beiden Stufen

erfüllt sind und wenn die Tabelle nur Spalten enthält, die nicht transitiv vom Primärschlüssel abhängen.

Wenn Sie diese Regeln befolgen, sind die Tabellen des Modells in der - wie es E. F. Codd, der Erfinder des relationalen Datenbanksystems nennt - dritten Normalform. Wenn sich Tabellen nicht in dieser Form befinden, befinden sich entweder redundante Daten in der Datenbank oder es wird Probleme geben, wenn die Daten geändert werden sollen.

Wenn es Ihnen nicht gelingt, einen Platz für ein Attribut gemäß dieser Regeln zu finden, haben Sie wahrscheinlich einen der folgenden Fehler begangen:

- Das Attribut ist nicht gut definiert.
- Das Attribut ist abgeleitet und nicht direkt.
- Das Attribut ist eigentlich eine Entity oder eine Beziehung.
- Im Modell fehlen einige Entities oder Beziehungen.

# Zusammenfassung

In diesem Kapitel wurden die folgenden Schritte des Entity-Relationship Modells dargestellt:

1. Identifizieren und definieren Sie ihre wesentlichen Datenobjekte, nämlich
   - Entities,
   - Beziehungen,
   - Attribute.
2. Skizzieren Sie Ihre Datenobjekte mit Hilfe des Entity-Relationship Diagrammansatzes.
3. Wandeln Sie Ihre Entity-Relationship Daten in relationale Konstrukte um.
   - Legen Sie für jede Entity Primärschlüssel und Fremdschlüssel fest.
4. Reorganisieren Sie Ihre Beziehungen, vor allem
   - Eins-zu-Eins Beziehungen,
   - m:n Beziehungen,
   - andere Sonderfälle von Beziehungen.
5. Normalisieren Sie Ihr Datenmodell:
   - Erste Normalform
   - Zweite Normalform
   - Dritte Normalform

Wenn Sie mehr über Datenbankdesign lernen wollen, empfehlen wir Ihnen die folgenden Titel:

- Database Modeling and Design, The Entity-Relationship Approach, Toby J. Teorey, Morgan Kauffman Publishers, Inc., 1990
- Handbook of Relational Database Design, by Candace C.Fleming und Barbara von Halle, Addison-Wesley Verlag, 1989

Kapitel

# Das Datenmodell implementieren



9

# Kapitelüberblick

Sobald das Datenmodell erstellt ist, muß es in Form einer Datenbank und in Form von Tabellen implementiert werden. Dieses Kapitel zeigt die Entscheidungen, die man bei der Implementierung des Modells treffen muß.

Der erste Schritt zur Implementierung ist die *Vervollständigung* des Datenmodells durch Bestimmung eines *Wertebereichs* für jede Spalte. Der zweite Schritt ist die Implementierung des Modells unter Verwendung von SQL-Anweisungen.

Der erste Abschnitt dieses Kapitels beschreibt im einzelnen, wie man den Wertebereich bestimmt. Der zweite Abschnitt zeigt, wie man eine Datenbank erzeugt (unter Verwendung der Kommandos CREATE DATABASE und CREATE TABLE) und wie man diese mit Daten füllt.

# Bestimmung der Wertebereiche

Um das in Kapitel 8 beschriebene Datenmodell zu implementieren, muß man für jede Spalte einen Wertebereich bestimmen. Der *Wertebereich einer Spalte* ergibt sich aus den Bedingungen, die für die Spalte bzw. das Attribut vergeben wurden sowie aus den zulässigen Eingabewerten.

Es muß also gewährleistet sein, daß die Daten die Wirklichkeit reflektieren. Falls ein Name dort eingetragen werden kann, wo eine Telefonnummer stehen sollte oder eine Zahl mit Nachkommastellen, wo eine ganze Zahl stehen sollte, dann ist die Integrität des Datenmodells gefährdet.

Bei der Bestimmung des Wertebereichs legt man zuerst die *Constraints* fest, denen ein Datenwert entsprechen muß, bevor er Bestandteil des Wertebereichs sein kann. Die Wertebereiche der Spalten werden unter Verwendung der folgenden Constraints festgelegt:

- Datentypen
- Standardwerte
- Prüf-Constraints

Zusätzlich kann man referentielle Constraints für Spalten vergeben, indem man sie als Primär- und Fremdschlüssel einer Tabelle festlegt. Wie diese Schlüssel bestimmt werden, wurde im Kapitel 8 dieses Handbuchs erläutert.

# Datentypen

Der wichtigste Constraint einer jeden Spalte ist der, der implizit im Datentyp einer Spalte enthalten ist. Bei der Auswahl eines Datentyps schränkt man die Eingabemöglichkeiten für die Spalte so ein, daß sie nur Werte enthalten kann, die durch diesen Typ repräsentiert werden.

Jeder Datentyp repräsentiert eine bestimmte Art von Daten. Der richtige Datentyp einer Spalte ist der, der all die Datenwerte repräsentiert, die für diese Spalte gedacht sind. Gleichzeitig sollte der Datentyp so wenig Werte wie möglich repräsentieren, die für diese Spalte nicht gedacht sind.

## Die Auswahl eines Datentyps

Jede Spalte einer Tabelle muß einen Datentyp haben. Dieser wird aus den Datentypen ausgewählt, die der Datenbankserver unterstützt. Die Auswahl des Datentyps ist aus mehreren Gründen wichtig:

- Er legt den zugrundeliegenden Wertebereich einer Spalte fest. Der Wertebereich ist die Menge der gültigen Datenelemente, die die Spalte speichern kann.
- Er legt die Art der Operationen fest, die man mit den Daten durchführen kann. Beispielsweise kann man Mengenfunktionen wie SUM nicht auf CHAR-Datentypen anwenden.
- Er legt fest, wieviel Platz jedes Datenelement auf der Platte belegt. Bei kleinen Tabellen ist dies nicht wichtig, aber wenn eine Tabelle zehn- oder hunderttausend Sätze enthält, kann der Unterschied zwischen einem 4-Byte- und einem 8-Byte-Datentyp entscheidend sein.

### Wahl der Datentypen bei Fremdschlüsseln

Primär- und Fremdschlüssel müssen fast immer vom gleichen Datentyp sein. Wenn der Primärschlüssel beispielsweise als CHAR definiert ist, muß auch der Fremdschlüssel als CHAR definiert sein. Wenn Sie allerdings einen Primärschlüssel vom Typ SERIAL haben, muß der dazugehörige Fremdschlüssel in der abhängigen Tabelle INTEGER sein. Die Kombination aus SERIAL und INTEGER ist der einzige Fall, bei dem Primär- und Fremdschlüssel von verschiedenem Datentyp sein dürfen. Die folgende Grafik in Bild 9-1 faßt die Möglichkeiten bei der Wahl von Datentypen zusammen.

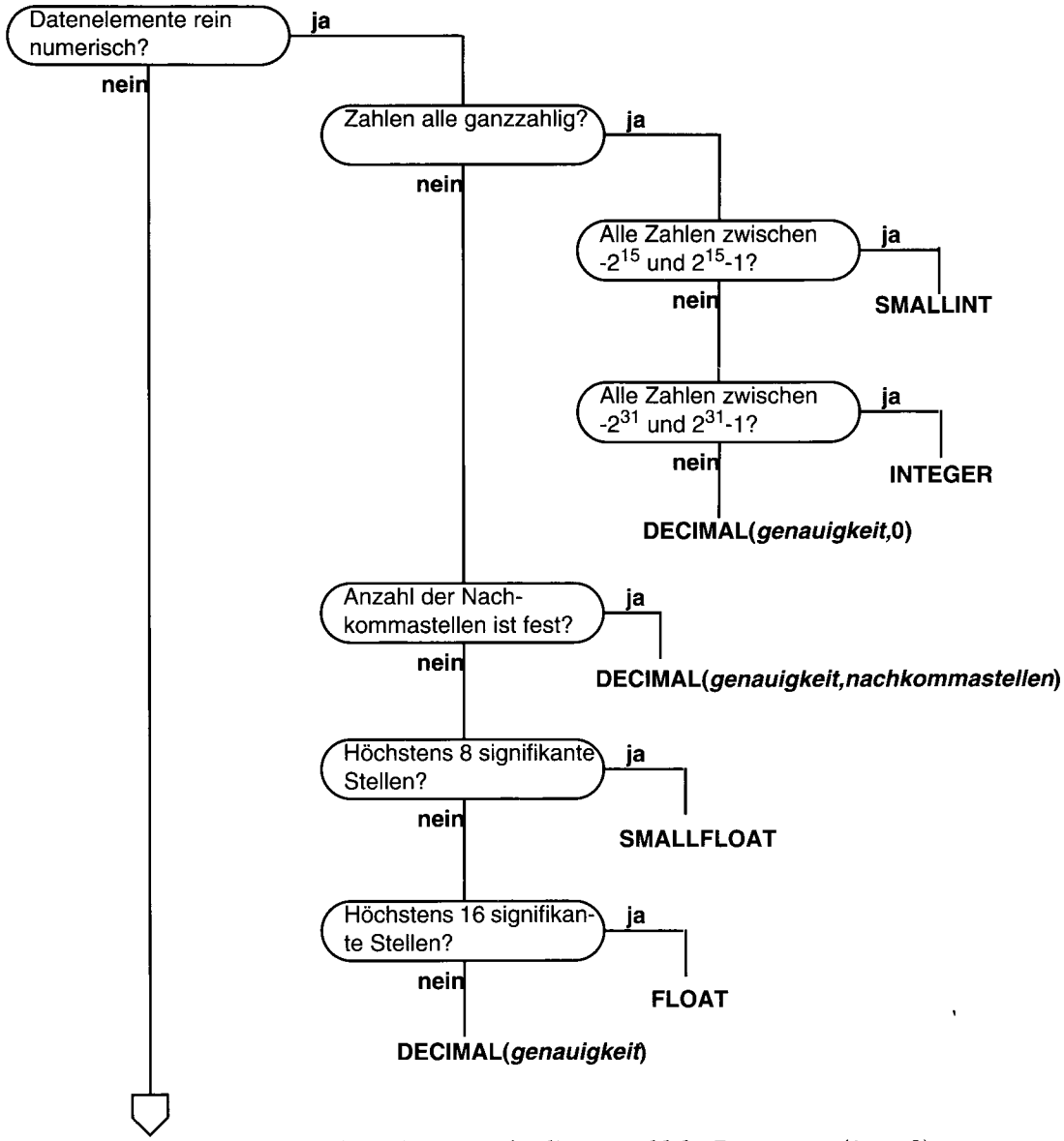



*Bild 9-1* *Ein Entscheidungsdiagramm für die Auswahl der Datentypen (1 von 2)*

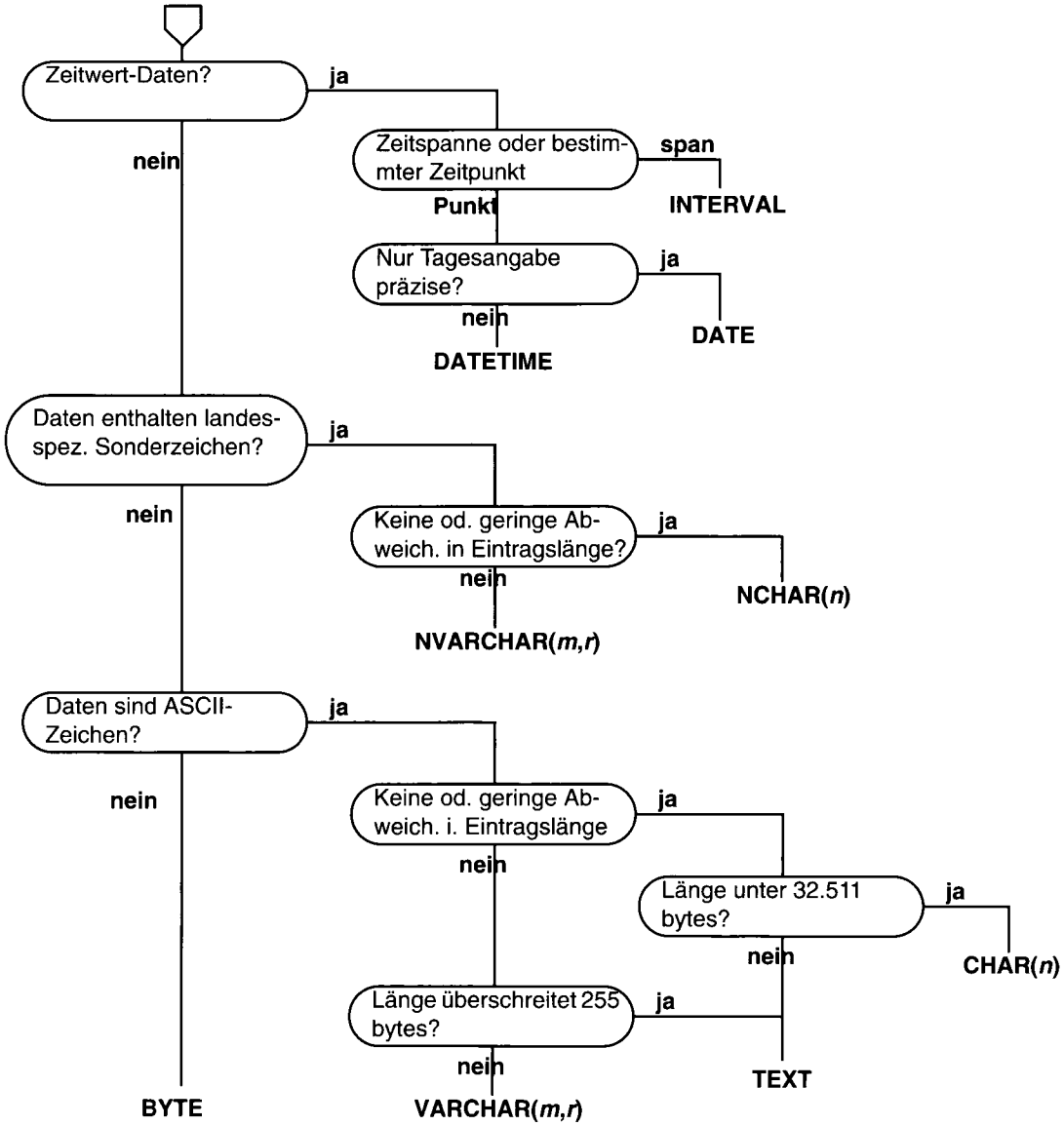



*Bild 9-1* *Ein Entscheidungsdiagramm für die Auswahl der Datentypen (2 von 2)*

# Numerische Datentypen

Die Informix-Datenbankserver unterstützen acht numerische Datentypen. Einige sind sehr gut geeignet für Zähler und Codes, einige für technische Größen und einige für Geldbeträge.

## Zähler und Codes: INTEGER und SMALLINT

Die Datentypen INTEGER und SMALLINT speichern kleine, ganze Zahlen. Sie sind für Spalten geeignet, die Zähler enthalten, fortlaufende Nummern, numerische Identifikationsnummern oder jeden beliebigen Bereich ganzer Zahlen, wenn man im voraus die kleinsten und größten zu speichernden Werte kennt.

Beide Typen werden als binäre Ganzzahlen mit Vorzeichen gespeichert. INTEGER-Werte bestehen aus 32 Bit und können ganze Zahlen von $-2^{31}$ bis $2^{31}-1$ darstellen; d. h. von –2.147.483.647 bis 2.147.483.647. (Die größte negative Zahl, –2.147.483.248, ist reserviert und kann nicht verwendet werden.)

SMALLINT-Werte bestehen nur aus 16 Bit. Sie können ganze Zahlen von –32.767 bis 32.767 darstellen. (Die größte negative Zahl, -32.768, ist reserviert und kann nicht verwendet werden.)

Diese Datentypen bieten zwei Vorteile:

- Sie benötigen wenig Platz (2 Byte pro SMALLINT-Wert und 4 Byte pro INTEGER-Wert).
- Arithmetische Ausdrücke wie SUM oder MAX und Sortiervergleiche können mit diesen Typen sehr effizient durchgeführt werden.

Der Nachteil bei der Verwendung von INTEGER und SMALLINT ist der begrenzte Wertebereich, den sie speichern können. Der Datenbankserver speichert keinen Wert, der die Aufnahmefähigkeit eines ganzzahligen Datentyps übersteigt. Wenn man die kleinsten und größten zu speichernden Werte kennt, stellt dies natürlich kein Problem dar.

## Automatische, fortlaufende Nummerierung: SERIAL

Der Datentyp SERIAL ist einfach ein INTEGER mit einer speziellen Funktion. Jedesmal, wenn in die Tabelle ein neuer Satz eingefügt wird, erzeugt der Datenbankserver für eine SERIAL-Spalte automatisch einen neuen Wert. Eine Tabelle kann jeweils nur eine einzige SERIAL-Spalte enthalten. Da der Datenbankserver diese Werte erzeugt, sind die fortlaufenden Werte in neuen Sätzen immer unterschiedlich; selbst dann, wenn mehrere Benutzer Sätze zur

gleichen Zeit hinzufügen. Dies ist eine nützliche Funktion, da es für gewöhnliche Programme ziemlich schwierig ist, unter diesen Bedingungen eindeutige numerische Codes zu erstellen.

### Wie viele fortlaufende Sätze?

**Nachdem $2^{31}$ Sätze in eine Tabelle eingefügt wurden, hat der Datenbankserver alle positiven fortlaufenden Zahlen verbraucht. Sollten Sie darüber besorgt sein? Wahrscheinlich nicht, da Sie, damit dieser Fall eintritt, 68 Jahre lang jede Sekunde einen Satz einfügen müßten. Falls dieser Fall jedoch eintreten würde, würde der Datenbankserver mit dem Erzeugen von Nummern fortfahren. Er würde die nächste, fortlaufende Zahl als eine Ganzzahl mit Vorzeichen behandeln. Da er nur positive Werte verwendet, würde er einfach wieder von vorn anfangen und die Erzeugung der ganzzahligen Werte mit 1 beginnen.**

Der Wert der erzeugten Nummern erhöht sich immer. Wenn Sätze aus der Tabelle gelöscht werden, wird deren fortlaufende Nummer *nicht* wiederverwendet. Dies bedeutet, daß Sätze, die nach einer SERIAL-Spalte sortiert sind, in der Reihenfolge geliefert werden, in der sie erstellt wurden. Dies gilt für keinen anderen Datentyp.

Den anfänglichen Wert einer SERIAL-Spalte können Sie in der CREATE TABLE-Anweisung festlegen. Dies macht es möglich, in verschiedenen Tabellen verschiedene Anfangswerte zu erzeugen. Die Datenbank **stores6** verwendet diese Technik. In **stores6** beginnt die Kundennummer (kunden_nr) mit 101, während die Auftragsnummer (**order_num**) mit 1001 beginnt. So lange wie diese Firma nicht mehr als 999 Kunden registriert, sind alle Kundennummern dreistellig und alle Auftragsnummern vierstellig.

Eine SERIAL-Spalte ist nicht automatisch eine eindeutige Spalte. Wenn man ganz sicher sein will, daß es keine doppelten fortlaufenden Nummern gibt, muß man einen Eindeutigkeits-Constraint verwenden (siehe "Die Verwendung von CREATE TABLE" auf Seite 9-31). Wenn Sie jedoch eine Tabelle unter Verwendung des interaktiven Schema-Editors von **DB-Access** bzw. **INFORMIX-SQL** erstellen, wird für jede SERIAL-Spalte automatisch ein Eindeutigkeits-Constraint verwendet.

Der Datentyp SERIAL bietet folgende Vorteile:

- Er stellt einen praktischen Weg für die Erzeugung der Schlüssel zur Verfügung, die vom System zugewiesen werden.
- Er erzeugt eindeutige numerische Codes, selbst wenn mehrere Benutzer die Tabelle aktualisieren.
- Unterschiedliche Tabellen können unterschiedliche Zahlenbereiche verwenden.

Dieser Datentyp hat folgende Nachteile:

- In einer Tabelle ist nur eine einzige SERIAL-Spalte erlaubt.
- Er kann nur willkürliche Nummern erzeugen (und willkürliche, also bedeutungslose numerische Codes könnten für manche Datenbank-Benutzer nicht akzeptabel sein).

### Die nächste SERIAL Nummer verändern

Der Anfangswert einer SERIAL-Spalte wird beim Erzeugen der Spalte gesetzt (siehe "Die Verwendung von CREATE TABLE" auf Seite 9-31). Später können Sie das Kommando ALTER TABLE verwenden, um den *nächsten* Wert neu zu setzen; der nächste Wert wird für den nächsten einzufügenden Satz verwendet.

Den *nächsten* Wert kann man nicht unterhalb des momentan größten Wertes der Spalte setzen, da dies den Datenbankserver veranlassen würde, doppelte Nummern zu erzeugen. Den nächsten Wert kann man jedoch auf jeden beliebigen Wert setzen, der über dem momentan größten Wert liegt; dies würde jedoch Lücken in der Reihenfolge erzeugen.

## Näherungswerte: FLOAT und SMALLFLOAT

Bei wissenschaftlichen, technischen und statistischen Anwendungen gibt es oft Zahlen bis zu einer Genauigkeit von ein paar Stellen; die Größe einer Zahl ist ebenso wichtig wie die genaue Stellenanzahl.

Die Gleitpunkt-Datentypen wurden für diese Anwendungen entwickelt. Man kann aus einem großen Bereich, der von kosmischen Größen bis zu mikroskopischen reicht, jede beliebige numerische Größe mit oder ohne Nachkommastellen darstellen. Man kann beispielsweise leicht sowohl die durchschnittliche Entfernung der Erde zur Sonne ($1{,}5 \times 10^{9}$ Meter) als auch die Planck'sche Konstante ($6{,}625 \times 10^{-27}$) darstellen. Ihre einzige Einschränkung ist die begrenzte Genauigkeit. Gleitpunkt-Zahlen behalten nur die signifikantesten Stellen der Werte. Wenn ein Wert nicht mehr Stellen hat, als eine Gleitpunkt-Zahl speichern kann, wird genau der Wert gespeichert. Wenn der Wert mehr Stellen hat, wird er als Näherungswert gespeichert; die letzten signifikanten Stellen werden als Nullen (0) behandelt.

Dieser Mangel an Genauigkeit ist für viele Verwendungen ausreichend, aber man sollte für die Aufzeichnung von Geldbeträgen niemals Gleitpunkt-Datentypen verwenden; dies gilt auch für andere Mengen, bei denen es ein Fehler ist, die letzten signifikanten Stellen in Null (0) umzuwandeln.

Bei den Gleitpunkt-Datentypen gibt es zwei Größen. Entsprechend der auf Ihrem Rechner vorliegenden Implementierung der Sprache C verarbeitet der Typ FLOAT binäre Gleitpunkt-Zahlen mit doppelter Genauigkeit. Eine Zahl belegt normalerweise 8 Byte. Der Typ SMALLFLOAT (auch als REAL bekannt) ist ein Typ mit einfacher Genauigkeit; diese binäre Gleitpunkt-Zahl belegt normalerweise 4 Byte. Der Hauptunterschied zwischen diesen zwei Datentypen liegt in der Genauigkeit. Eine FLOAT-Spalte speichert 16 Stellen der Werte, wohingegen eine SMALLFLOAT-Spalte nur 8 Stellen speichert.

Gleitpunkt-Zahlen bieten folgende Vorteile:

- Sie speichern sehr große und sehr kleine Zahlen, inklusive der Zahlen mit Nachkommastellen.
- Sie stellen Zahlen kompakt in 4 oder 8 Byte dar.
- Arithmetische Funktionen wie AVG und MIN und Sortiervergleiche sind bei diesen Datentypen sehr effektiv.

Der Hauptnachteil der Gleitpunkt-Zahlen ist, daß die Stellen, die außerhalb der Genauigkeitsbereichs liegen, als Nullen (0) behandelt werden.

### Veränderbare Genauigkeit des Gleitpunkts: DECIMAL(*genauigkeit*)

Der Datentyp DECIMAL(*genauigkeit*) ist ein Gleitpunkt-Typ, der den Typen FLOAT und SMALLFLOAT ähnlich ist. Der wichtige Unterschied ist, daß man die Anzahl der signifikanten Stellen bestimmen kann, die der Datentyp speichert. Die angegebene *Genauigkeit* kann von 1 bis 32 reichen; also von einer kleineren Genauigkeit als SMALLFLOAT bis zur doppelten Genauigkeit von FLOAT.

Die Größe einer DECIMAL(*genauigkeit*)-Zahl reicht von $10^{-128}$ bis $10^{126}$.

Dezimale Datentypen können einen leicht verwirren. Der hier erläuterte Datentyp ist DECIMAL(*genauigkeit*); dies meint den Datentyp DECIMAL, bei dem nur die Genauigkeit festgelegt ist. Die Größe der DECIMAL(*genauigkeit*) Zahlen hängt von der angegebenen *Genauigkeit* ab; sie belegen *1+genauigkeit/2* Byte (eventuell auf eine ganze Zahl aufgerundet).

DECIMAL(*genauigkeit*) hat gegenüber FLOAT folgende Vorteile:

- Die Genauigkeit kann für die Anwendung passend gesetzt werden, von einer hohen Näherung auf eine hohe Genauigkeit.
- Zahlen mit bis zu 32 Stellen können genau dargestellt werden.
- Der Speicher wird entsprechend der Genauigkeit der Zahl verwendet.

- Jeder Informix Datenbankserver unterstützt dieselbe Genauigkeit und dieselben Größenbereiche, unabhängig vom Host-Betriebssystem.

Der Datentyp DECIMAL(*genauigkeit*) hat folgende Nachteile:

- Arithmetische Funktionen und Sortierungen sind etwas langsamer als bei FLOAT-Zahlen.
- Viele Programmiersprachen unterstützen das DECIMAL(*genauigkeit*)-Datenformat nicht in gleicher Weise wie FLOAT und INTEGER. Wenn ein Programm der Datenbank einen DECIMAL(*genauigkeit*)-Wert entnimmt, muß es diesen für die Verarbeitung eventuell in ein anderes Format konvertieren. (**INFORMIX-4GL**-Programme können DECIMAL(*genauigkeit*)-Werte direkt verwenden.)

## Festpunktzahlen: DECIMAL und MONEY

Die meisten kaufmännischen Anwendungen müssen Zahlen speichern, die eine feste Anzahl von Vor- und Nachkommastellen haben. Geldbeträge sind das häufigste Beispiel. Beträge in U.S.- oder einer anderen Währung werden mit zwei Nachkommastellen geschrieben. Normalerweise kennt man auch in Abhängigkeit der zu speichernden Transaktion die Anzahl der Vorkommastellen – vielleicht 5 Stellen für ein persönliches Budget, 7 für ein kleines Geschäft und 12 oder 13 für ein nationales Budget.

Diese Zahlen sind *Festpunkt*-Zahlen, da das Dezimalkomma an einer bestimmte Stelle fixiert ist – unabhängig von dem Wert der Zahl. Der Datentyp DECIMAL(*genauigkeit, nachkommastellen*) wurde für die Speicherung solcher Werte entwickelt. Wenn Sie für eine Spalte diesen Typ angeben, geben Sie die *Genauigkeit* als die gesamte Anzahl der Stellen an, die gespeichert werden können; von 1 bis 32. Geben Sie die *Nachkommastellen* als die Anzahl der Stellen an, die rechts vom Komma stehen (Die Beziehung zwischen Genauigkeit und Nachkommastellen ist in Bild 9-2 dargestellt.). Die Nachkommastellen können Null (0) sein; dies bedeutet, daß nur ganze Zahlen gespeichert werden. Wenn man dies tut, dann unterstützt DECIMAL(*genauigkeit, nachkommastellen*) die Speicherung ganzzahliger Werte bis zu 32 Stellen.

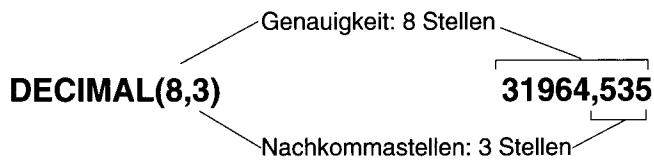



*Bild 9-2* ***Beziehung zwischen Genauigkeit und Nachkommastellen bei einer Festpunktzahl***

Wie der Datentyp DECIMAL(*genauigkeit*) belegt DECIMAL(*genauigkeit, nachkommastellen*) Speicherplatz entsprechend der Genauigkeit. Ein Wert belegt 1+*genauigkeit*/2 Byte, auf eine ganze Anzahl an Byte aufgerundet.

Der Datentyp MONEY ist identisch mit DECIMAL(*genauigkeit, nachkommastellen*) , verfügt aber über ein zusätzliches Merkmal. Jedesmal wenn der Datenbankserver einen MONEY-Wert für die Anzeige umwandelt, fügt er automatisch ein Währungssymbol hinzu.

Die Vorteile von DECIMAL(*genauigkeit, nachkommastellen*) gegebenüber INTEGER und FLOAT sind, daß eine viel größere Genauigkeit verfügbar ist (bis zu 32 Stellen im Vergleich zu 10 bei INTEGER und 16 bei FLOAT), wohingegen die Genauigkeit und die geforderte Speichermenge der Anwendung entsprechend angepaßt werden kann.

Die Nachteile sind, daß arithmetische Funktionen weniger effizient sind und daß viele Programmiersprachen Zahlen in dieser Form nicht unterstützen. Aus diesem Grund muß ein Programm eine Zahl normalerweise für die Verarbeitung in ein anderes numerisches Format umwandeln. (**INFORMIX-4GL**-Programme können DECIMAL(*genauigkeit, nachkommastellen*) - und MONEY-Werte direkt verwenden.)

### Wessen Geld? Auswahl des Währungsformats

Jede Nation schreibt Geldbeträge in seiner eigenen Weise. Wenn ein Informix Datenbankserver einen MONEY-Wert anzeigt, bezieht er sich auf das Währungsformat, das auf Betriebssystemebene definiert ist; diese Definition steht normalerweise in der Variablen DBMONEY. Ein Währungssymbol kann vor oder hinter dem Betrag stehen; das Tausendertrennzeichen kann auf einen Punkt, ein Komma oder ein anderes Zeichen gesetzt werden. Näheres hierzu siehe Kapitel 4 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen*.

## Zeitwert-Typen

Informix-Datenbankserver unterstützen für die Aufzeichnung von Zeitwerten drei Datentypen. Der Datentyp DATE speichert ein Kalenderdatum. DATETIME zeichnet einen Zeitpunkt mit einer beliebigen Genauigkeit auf, die von einem Jahr bis zu einem Bruchteil einer Sekunde reichen kann. Der Datentyp INTERVAL speichert eine Zeitspanne, d. h. eine Zeitdauer.

## Kalenderdaten: DATE

Der Datentyp DATE speichert ein Kalenderdatum. Ein DATE-Wert ist in Wirklichkeit ein ganzzahliger Wert mit Vorzeichen, dessen Inhalt als die Anzahl ganzer Tage interpretiert wird, die seit dem 31. Dezember 1899 vergangen sind. Meistens speichert er eine positive Zahl, die für Tage des aktuellen Jahrhunderts steht.

Das Format DATE verfügt über eine ausreichende Größe, um auch Kalenderdaten der fernen Zukunft (58.000 Jahrhunderte) speichern zu können. Negative DATE-Werte werden als Tage vor dem Ursprungsdatum interpretiert; d. h. der Wert -1 stellt den 30. Dezember 1899 dar.

Da DATE-Werte ganze Zahlen sind, lassen es Informix-Datenbankserver zu, diese in arithmetischen Ausdrücken zu verwenden. Man kann z. B. den Durchschnitt einer DATE-Spalte ermitteln oder man kann 7 oder 365 zu dem Wert einer DATE-Spalte addieren. Außerdem gibt es speziell für die Veränderung von DATE-Werten eine große Anzahl an Funktionen. (Siehe Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*)

Der Datentyp DATE benötigt für die Speicherung pro Element 4 Byte. Arithmetische Funktionen und Vergleiche werden mit einer DATE-Spalte schneller ausgeführt.

### M/D/Y? D-M-Y? Die Wahl eines Datumsformats

Die Datumskomponenten kann man auf viele Arten schreiben und anordnen. Wenn ein Informix Datenbankserver einen DATE-Wert anzeigt, bezieht er sich auf ein Format, das im Betriebssystem normalerweise in der Variablen DBDATE festgelegt ist. Die Werte für Tag, Monat und Jahr können in beliebiger Reihenfolge angezeigt werden, voneinander durch ein ausgewähltes Zeichen getrennt. Weiter Informationen erhalten Sie im Kapitel des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen.*

## Genaue Zeitpunkte: DATETIME

Der Datentyp DATETIME speichert jeden beliebigen Zeitpunkt seit dem Jahr 1. DATETIME ist in Wirklichkeit eine Familie aus 28 Datentypen, wobei jeder Typ über eine unterschiedliche Genauigkeit verfügt. Bei der Definition einer DATETIME-Spalte gibt man die Genauigkeit an. Die Genauigkeit kann jede beliebige Folge der Komponenten *year (Jahr), month (Monat), day (Tag), hour (Stunde), minute (Minute), second (Sekunde)* und *fraction(Sekundenbruchteil)* sein. Somit kann man eine DATETIME-Spalte definieren, die nur Jahre speichert, nur Monat und Tag oder nur Tag und Uhrzeit, die die Stunde oder

die Zeit bis auf Tausendstelsekunden genau angibt. Die Größe eines DATETIME-Wertes reicht von 2 bis 11 Byte; dies hängt, wie in Bild 9-3 gezeigt, von der Genauigkeit ab.

Der Vorteil des Datentyps DATETIME liegt darin, daß er Daten genauer als in ganzen Tagen speichern kann, und daß er Zeitwerte speichern kann. Der einzige Nachteil ist das unflexible Anzeigeformat, aber dies kann umgangen werden (siehe "Das Format für einen DATETIME- oder INTERVAL-Wert festlegen" auf Seite 9-19).

| Genauigkeit | Größe* | Genauigkeit | Größe* |
|---|---|---|---|
| year to year | 3 | day to hour | 3 |
| year to month | 4 | day to minute | 4 |
| year to day | 5 | day to second | 5 |
| year to hour | 6 | day to fraction(z) | 5+z/2 |
| year to minute | 7 | hour to hour | 2 |
| year to second | 8 | hour to minute | 3 |
| year to fraction(z) | 8+z/2 | hour to second | 4 |
| month to month | 2 | hour to fraction(z) | 4+z/2 |
| month to day | 3 | minute to minute | 2 |
| month to hour | 4 | minute to second | 3 |
| month to minute | 5 | minute to fraction(z) | 3+z/2 |
| month to second | 6 | second to second | 2 |
| month to fraction(z) | 6+z/2 | second to fraction(z) | 2+z/2 |
| day to day | 2 | fraction to fraction(z) | 1+z/2 |

* Wenn z ungerade ist, wird die Größe auf das nächste ganzzahlige Byte aufgerundet.

*Bild 9-3* ***Alle möglichen Genauigkeiten des Datentyps DATETIME, mit Größenangaben in Byte***

## Zeitdauer: INTERVAL

Der Datentyp INTERVAL speichert eine Zeitdauer, d. h. eine Zeitlänge. Die Differenz zwischen zwei DATETIME-Werten ist ein Intervall, das die Zeitspanne darstellt, die diese voneinander unterscheidet. Die folgenden Beispiele können dazu beitragen, die Unterschiede zu erläutern:

- Eine Angestellte fing am 21. Mai 1991 (entweder ein DATE- oder DATETIME-Wert) in einer Firma an.
- Sie arbeitete dort 254 Tage (ein INTERVAL, die Differenz zwischen der Funktion TODAY und dem anfänglichen DATE- oder DATETIME-Wert).
- Sie fing jeden Tag um 9.00 Uhr (ein DATETIME-Wert) an.
- Sie arbeitete 8 Stunden (ein INTERVAL) mit 45 Minuten Mittagspause (ein weiteres INTERVAL).
- Sie hat um 17.45 Uhr Feierabend (die Summe aus dem DATETIME-Wert, der ihrem Arbeitsbeginn entspricht, und den zwei INTERVAL-Werten).

Wie DATETIME ist auch INTERVAL eine Datentypfamilie mit unterschiedlichen Genauigkeiten. Ein INTERVAL-Wert kann eine Anzahl von Jahren und Monaten darstellen; oder er kann eine Anzahl von Tagen, Stunden, Minuten, Sekunden oder Sekundenbruchteilen darstellen: insgesamt gibt es 18 mögliche Genauigkeiten. Die Größe eines INTERVAL-Wertes reicht, wie in Bild 9-4 gezeigt, von 2 bis 8 Byte.

| Genauigkeit* | Größe** | Genauigkeit* | Größe** |
|---|---|---|---|
| year(*g*) to year | 1+*g*/2 | hour(*g*) to minute | 2+*g*/2 |
| year(*g*) to month | 2+*g*/2 | hour(*g*) to second | 3+*g*/2 |
| month(*g*) to month | 1+*g*/2 | hour(*g*) to fraction(z) | 4+(*g*+z)/2 |
| day(*g*) to day | 1+*g*/2 | minute(*g*) to minute | 1+*g*/2 |
| day(*g*) to hour | 2+ *g*/2 | minute(*g*) to second | 2+*g*/2 |
| day(*g*) to minute | 3+*g*/2 | minute(*g*) to fraction(z) | 3+(*g*+z)/2 |
| day(*g*) to second | 4+*g*/2 | second(*g*) to second | 1+*g*/2 |
| day(*g*) to fraction(z) | 5+(*g*+*f*)/2 | second(*g*) to fraction(z) | 2+(*g*+ z)/2 |
| hour(*g*) to hour | 1+*g*/2 | fraction to fraction(z) | 1+z/2 |

* *g = genauigkeit; z = ziffer*
** Gebrochene Größen auf die nächsthöhere Anzahl ganzer Byte aufrunden.

*Bild 9-4* ***Alle möglichen Genauigkeiten des Datentyps INTERVAL, mit Größenangaben in Byte***

INTERVAL-Werte können negativ und positiv sein. Man kann sie addieren oder subtrahieren und man kann sie durch multiplizieren oder dividieren mit einer Zahl vergrößern oder verkleinern. Dies gilt nicht für DATE- oder

DATETIME-Werte. Die Frage "Was ist die Hälfte der Anzahl der Tage bis zum 23. April?" ist sinnvoll, die Frage "Was ist die Hälfte des 23. April?" ist aber nicht sinnvoll.

## Zeichentypen

Alle Informix Datenbankserver unterstützen den Datentyp CHAR(*n*). Wenn NLS aktiviert ist, unterstützen die Informix Datenbankserver den Datentyp NCHAR(n). **INFORMIX-OnLine** unterstützt außerdem weitere, spezielle Zeichentypen.

### Zeichendaten: CHAR(n) und NCHAR(n)

Der Datentyp CHAR(*n*) enthält eine Folge von *n* ASCII-Zeichen. Die Länge von *n* muß im Bereich von 1 bis 32.767 liegen. (Wenn Sie den **INFORMIX-SE**-Datenbankserver verwenden betraägt die maximale Länge 32.511). Ist NLS aktiviert, enthält der Datentyp NCHAR(*n*) eine Folge von landesspezifischen Zeichen in derselben Länge wie CHAR.

Wann immer ein CHAR(*n*)- NCHAR(*n*)-Wert abgerufen oder gespeichert wird, werden genau *n* Byte übertragen. Ist ein eingefügter Wert kürzer, so wird er mit Leerzeichen bis zur Länge *n* aufgefüllt; ist ein eingefügter Wert zu lang, so wird er entsprechend abgeschnitten. Für Werte variabler Länge ist in diesem Format keine solche Operation vorgesehen.

Der Vorteil des Datentyps CHAR(*n*) bzw. NCHAR(*n*) liegt darin, daß er auf allen Datenbankservern verfügbar ist. NCHAR(*n*) ist auf allen Datenbankservern verfügbar, auf denen NLS aktiviert ist. Der einzige Nachteil ist, daß seine Länge fest ist - differiert die Länge der Datenwerte von Satz zu Satz sehr, wird Speicherplatz verschwendet.

### Zeichenketten variabler Länge: VARCHAR und NVARCHAR

Die Elemente in einer CHAR-Spalte sind oft unterschiedlich lang, d. h. viele Werte haben eine durchschnittlichen Länge und nur einige Werte haben die maximale Länge. Der Datentyp VARCHAR(*maximum, mindestlänge*) (beziehungsweise NVARCHAR(*maximum, mindestlänge*)bei NLS-Unterstützung) wurde entwickelt, um beim Speichern solcher Daten auf der Platte Platz zu sparen. Eine als VARCHAR(*maximum, mindestlänge*) bzw. mit NVARCHAR(*maximum, mindestlänge*) definierte Spalte wird wie eine als CHAR(*n*) definierte Spalte verwendet. Bei der Definition einer VARCHAR(*maximum, mindestlänge*) -bzw. einer NVARCHAR(*maximum, mindestlänge*)-Spalte legt man mit *maximum* die *maximale* Länge eines Datenelements fest. Auf der Platte wird nur

der tatsächliche Inhalt eines jeden Elements plus einem Byte, das die Länge anzeigt, gespeichert. Die oberste Grenze für *maximum* ist 255; d. h. ein VARCHAR- bzw. NVARCHAR -Wert kann bis zu 255 Zeichen speichern.

Wenn die Spalte indiziert ist, liegt die oberste Grenze für *maximum* bei 254.

### Das Format für einen DATETIME- oder INTERVAL-Wert festlegen

Der Datenbankserver zeigt die Komponenten eines INTERVAL- oder DATETIME-Wertes immer in der Reihenfolge *year-month-day hour:minute:second.fraction* an. Er bezieht sich nicht auf das Datumsformat, das auf Betriebssystemebene definiert ist, so wie er es beim Formatieren eines DATE-Wertes tut.

Man kann eine SELECT-Anweisung erstellen, die die Datumskomponenten eines DATETIME-Wertes in dem Format anzeigt, das im System definiert ist. Der Trick besteht darin, die Felder der Komponenten mit der Funktion EXTEND herauszukristallisieren und diese an die Funktion MDY() weiterzuleiten; diese Funktion wandelt die Komponenten in einen DATE-Wert um. Hier ein Ausschnitt aus einem Beispiel:

```
SELECT ... MDY (
      EXTEND (DATE_RECEIVED, MONTH TO
       MONTH),
      EXTEND (DATE_RECEIVED, DAY TO DAY),
      EXTEND (DATE_RECEIVED, YEAR TO YEAR))
      FROM RECEIPTS ...
```

Beim Entwickeln einer Liste mit **INFORMIX-4GL** oder **INFORMIX-SQL** haben Sie über die PRINT-Anweisung eine größere Flexibilität. Wählen Sie jede Komponente eines DATETIME- oder INTERVAL-Wertes unter Verwendung der Funktion EXTEND als einen Ausdruck aus. Geben Sie zur Vereinfachung jedem Ausdruck einen Alias-Namen. Kombinieren Sie in einem PRINT-Ausdruck die Komponenten mit der gewünschten Zeichensetzung.

```
SELECT ...
      EXTEND (START_TIME, HOUR TO HOUR) H,
      EXTEND (START_TIME, MINUTE TO MINUTE) M, ...
```

Anschließend in der Liste:

```
PRINT "Start work at ", H USING "&&", M USING "&&", "hours."
```

```
Start work at 0800 hours
```

Der zweite Parameter, *mindestlänge*, bezeichnet eine optionale Reservierungslänge; diese legt eine untere Grenze für die Länge eines Elements fest, wie es auf der Platte gespeichert werden soll. Ist ein Element kürzer als *mindestlänge*, so werden trotzdem *mindestlänge* Byte für die Speicherung zugewiesen. Der Zweck liegt darin, bei Aktualisierungen Zeit zu sparen. (Siehe "Die Ausführungszeit bei variabler Länge" auf Seite 9-21.)

Die Vorteile des Datentyps VARCHAR(*maximum, mindestlänge*) bzw. NVARCHAR(*maximum, mindestlänge*) gegenüber dem Datentyp CHAR(*n*) bzw. NCHAR(*n*) sind folgende:

- Er spart Speicherplatz, wenn die Längen der Datenelemente sehr unterschiedlich sind oder wenn nur einige Elemente länger als die durchschnittliche Länge sind.
- Abfragen bei kompakteren Tabellen können schneller sein.

Der Datentyp hat folgende Nachteile:

- Er läßt nicht mehr als 255 Zeichen zu.
- In einigen Fällen können Aktualisierungen einer Tabelle langsamer sein.
- Er ist nicht auf allen Informix Datenbankservern verfügbar.

## Die Ausführungszeit bei variabler Länge

Wenn der Datentyp VARCHAR(*maximum, mindestlänge*) bzw. NVARCHAR(*maximum, mindestlänge*) verwendet wird, dann haben die Sätze einer Tabelle unterschiedliche anstatt fester Längen. Dies wirkt sich auf die Geschwindigkeit der Datenbank-Operationen unterschiedlich aus.

Da auf eine Page mehr Sätze passen, kann der Datenbankserver eine Tabelle mit weniger Plattenoperationen durchsuchen, als wenn die Sätze eine feste Länge hätten. Daraus ergibt sich, daß Abfragen schneller ausgeführt werden können. Aus demselben Grund können Einfüge- und Löschvorgänge ein bißchen schneller sein.

Wenn man einen Satz aktualisiert, hängt der Arbeitsaufwand des Datenbankservers von der Länge des neuen Satzes ab, verglichen mit der Länge des alten Satzes. Wenn der neue Satz genauso lang oder kürzer ist, ist die Ausführungszeit nicht wesentlich anders als bei einem Satz mit fester Länge. Wenn der neue Satz aber länger als der alte ist, muß der Datenbankserver möglicherweise einige Plattenoperationen durchführen. Somit kann die Aktualisierung einer Tabelle, die VARCHAR(*maximum, mindestlänge*)- bzw. NVARCHAR(*maximum, mindestlänge*)-Daten verwendet, manchmal langsamer sein als bei Aktualisierungen mit Elementen fester Länge.

Diese Auswirkung kann man verringern, indem man *mindestlänge* als eine Größe angibt, die einen Großteil der Datenelemente beinhaltet. Die meisten Sätze verwenden dann die Reservierungslänge; für Auffüllungen wird nur wenig Platz vergeudet und Aktualisierungen sind nur dann langsam, wenn ein normaler Wert durch einen der seltenen, langen ersetzt wird.

### Große Zeichenobjekte: TEXT

Der Datentyp TEXT speichert Textblöcke. Er wurde entwickelt, um Dokumente zu speichern: Geschäftsformulare, Programm-Quellcode, ausführbare Programme oder Memos. Obwohl man in einem TEXT-Element beliebige Daten speichern kann, sollte dieser Datentyp auf druckbare ASCII-Zeichen beschränkt werden, weil Informix-Werkzeuge annehmen, daß ein TEXT-Element druckbar ist.

TEXT-Werte werden nicht mit den Sätzen zusammen gespeichert, von denen sie ein Bestandteil sind. Sie werden in ganzen Pages angefordert; normalerweise sind dies Bereiche, die von den Sätzen getrennt sind. (Siehe "Blob-Daten anlegen" auf Seite 10-21.).

### Wie BLOBs verwendet werden

Die Spalten des Typs TEXT und BYTE werden gemeinsam *Binary Large Objects* oder BLOBs genannt. Der Datenbankserver speichert sie nur und ruft sie nur ab. Normalerweise werden BLOB-Werte von Programmen geholt und gespeichert. Diese Programme sind in **INFORMIX-4GL** oder einer Sprache, die eingebettetes SQL unterstützt wie z. B. **INFORMIX-ESQL/C**, geschrieben. Mit einem solchen Programm kann man einen BLOB-Wert in ähnlicher Weise holen, einfügen oder aktualisieren, wie man eine sequentielle Datei liest oder schreibt.

In einer SQL-Anweisung (interaktiv oder programmiert) *kann* eine BLOB-Spalte auf folgende Arten *nicht* verwendet werden:

- In arithmetischen oder bool'schen Ausdrücken
- In einer GROUP BY- oder ORDER BY-Klausel
- In einer UNIQUE-Überprüfung
- Für die Indizierung, entweder auf sich selbst oder als Teil eines zusammengesetzten Index

In einer interaktiv eingegebenen SELECT-Anweisung, in einem Format oder einer Liste kann ein BLOB nur

- anhand des Namens ausgewählt werden; optional kann man einen Feldausschnitt angeben, um einen Teil auszuwählen
- seine Länge anhand der Auswahl LENGTH(*spalte*) zurückliefern
- mit dem Prädikat IS [NOT] NULL überprüft werden

In einer interaktiven INSERT-Anweisung kann man die VALUES-Klausel zum Einfügen eines BLOB-Wertes verwenden; der NULL-Wert ist aber der einzige, den man vergeben kann. Man kann jedoch das SELECT-Format der INSERT-Anweisung verwenden, um einen BLOB-Wert von einer anderen Tabelle zu kopieren.

In einer interaktiven UPDATE-Anweisung kann man eine BLOB-Spalte auf NULL setzen oder das Ergebnis einer Unterabfrage in ihr verwenden.

Der Vorteil des Datentyps TEXT gegenüber CHAR(*n*) und VARCHAR(*maximum, mindestlänge*) ist, daß die Größe eines TEXT-Datenelements nicht begrenzt ist, außer durch die Speicherkapazität der Platte. Der Datentyp TEXT weist folgende Nachteile auf:

- Er wird in ganzen Pages angefordert; ein kurzes Element vergeudet Platz.
- Es gibt Einschränkungen für die Verwendung einer TEXT-Spalte in einer SQL-Anweisung. (Siehe „BLOB-Daten anlegen" in Kapitel10)
- Er ist nicht auf allen Informix Datenbankservern verfügbar.

Man kann TEXT-Werte in Listen anzeigen lassen, die mit **INFORMIX-4GL**-Programmen oder dem **ACE** Listengenerator generiert wurden. Man kann TEXT-Werte auf einem Bildschirm anzeigen lassen und diese mit Hilfe von Bildschirmmasken editieren, die mit **INFORMIX-4GL**-Programmen oder mit dem Bildschirmmasken-Prozessor **PERFORM** generiert wurden.

### Binäre Objekte: BYTE

Der Datentyp BYTE wurde entwickelt, um beliebige Daten, die ein Programm erzeugen kann, zu speichern: Graphiken, Programm-Objektdateien und Dokumente, die von einem Textverarbeitungs- oder Kalkulationsprogramm gesichert wurden. Der Datenbankserver läßt in einer BYTE-Spalte jede beliebige Art von Daten mit beliebiger Länge zu.

Wie TEXT werden auch BYTE-Datenelemente in ganzen Pages gespeichert; die Pages liegen in Plattenbereichen, die von den normalen Datensätzen getrennt sind.

Im Gegensatz zu TEXT oder CHAR(*n*) liegt der Vorteil des Datentyps BYTE darin, daß er alle beliebigen Daten akzeptiert. Die Nachteile sind dieselben wie beim Datentyp TEXT.

## Den Datentyp ändern

Mit dem Kommando ALTER TABLE kann man den Datentyp ändern, der einer Spalte nach der Erstellung der Tabelle zugewiesen wurde. Obwohl die Änderung manchmal notwendig ist, sollte man sie aus folgenden Gründen vermeiden:

- Um einen Datentyp zu ändern, muß der Datenbankserver die Tabelle kopieren und neu aufbauen. Bei großen Tabellen kann dies viel Zeit und Speicherplatz benötigen.
- Einige Veränderungen des Datentyps können zu einem Informationsverlust führen. Wenn man beispielsweise eine Spalte von einem längeren in einen kürzeren Zeichentyp ändert, werden längere Werte abgeschnitten;

wenn man in einen weniger genauen numerischen Datentyp ändert, werden Nachkommastellen abgeschnitten.

- Vorhandene Programme, Masken, Listen und gespeicherte Abfragen müssen vielleicht auch geändert werden.

## Standardwerte

Ein Standardwert ist ein Wert, der in eine Spalte dann eingefügt wird, wenn ein expliziter Wert nicht angegeben ist. Ein Standardwert kann eine literale Zeichenkette sein, die von Ihnen oder durch einen der folgenden konstanten SQL-Ausdrücke festgelegt wurde:

- USER
- CURRENT
- TODAY
- DBSERVERNAME

Nicht alle Spalten benötigen Standardwerte, aber wenn Sie mit Ihrem Datenmodell arbeiten, entdecken Sie vielleicht Beispiele, bei denen die Verwendung eines Standardwertes bei der Dateneingabe Zeit spart oder Fehler bei der Dateneingabe verhindert. Beispielsweise enthält das Adreßbuch-Modell die Spalte **adresse**. Beim Betrachten der Werte dieser Spalte stellt man fest, daß mehr als 50% der Adressen in der Spalte **adresse** den Wert Bayern enthalten. Um Zeit zu sparen, gibt man die Zeichenkette "Bayern" als Standardwert für die Spalte **adresse** an.

Standardwerte sind auch dann zweckmäßig, wenn eine Spalte einen Schalter enthält. Das Adreßbuch-Modell verwendet den Schalter (Y/N), um anzuzeigen, ob ein Modem mit 300, 1200 oder 2400 Baud arbeitet. Man könnte für diese Spalte einen Standardwert ("N") festlegen. Dann müßte man künftig keine Werte für die Modems eintragen, die nicht mit bestimmten Baudraten arbeiten.

## Prüf-Constraints

Prüf-Constraints legen eine Bedingung oder Anforderung an Datenwerte fest, die ein Wert erfüllen muß, bevor er mit INSERT oder UPDATE eingefügt wird. Wenn ein Datensatz diese Bedingungen nicht erfüllt, wird er zurückgewiesen und eine Fehlermeldung wird ausgegeben. Sie geben einen Constraint in der CREATE TABLE oder der ALTER TABLE Anweisung an. Im folgenden Beispiel gibt es für die ganzzahligen Werte eine Einschränkung auf einen bestimmten Bereich:

```
customer_num >= 50000 AND customer_num <= 99999
```

Bei CHAR-Bereichen kann man Constraints auch unter Verwendung des MATCHES-Prädikats und eines syntaktisch zulässigen regulären Ausdrucks festlegen. Der folgende Constraint schränkt z. B. den Wertebereich für eine Telefonnummer entsprechend der Form einer lokalen amerikanischen Telefonnummer ein:

```
telefon_nr MATCHES "[2-9][2-9][0-9]-[0-9][0-9][0-9][0-9]"
```

Weitere Informationen zu diesem Thema finden Sie unter CREATE TABLE und ALTER TABLE im Handbuch *SQL-Sprachbeschreibung, Syntax.*

# Das Erzeugen der Datenbank

Nun kann man das Datenmodell in Form von Tabellen in einer Datenbank erzeugen. Man tun dies mit den Kommandos CREATE DATABASE, CREATE TABLE und CREATE INDEX. Die Syntax dieser Kommandos ist im Handbuch *SQL-Sprachbeschreibung, Syntax* detailliert beschrieben. Dieser Abschnitt erläutert die Verwendung der Kommandos CREATE DATABASE und CREATE TABLE bei der Implementierung eines Datenmodells. Die Verwendung des Kommandos CREATE INDEX ist im nächsten Kapitel enthalten.

Bedenken Sie, daß unser Adressbuch-Beispiel lediglich der Veranschaulichung dient. Aus Beispielgründen werden wir es in SQL-Anweisungen übersetzen.

Man muß dasselbe Modell möglicherweise mehr als einmal erzeugen. Die Kommandos zum Erzeugen des Modells können jedoch gespeichert und automatisch ausgeführt werden. Weitere Informationen zu diesem Thema erhalten Sie unter "Die Verwendung von Anweisungsdateien" auf Seite 9-33.

Sobald die Tabellen vorhanden sind, kann man diese mit Datensätzen füllen. Dies kann man manuell tun, mit einem Dienstprogramm oder mit einem Benutzerprogramm.

# Die Verwendung von CREATE DATABASE

Eine Datenbank ist ein Behälter, der alle Teile speichert, die zu einem Datenmodell gehören. Diese Teile umfassen natürlich die Tabellen, aber auch die Views, Indizes, Synonyme und Weiteres. Bevor man etwas anderes erzeugen kann, muß man die Datenbank erzeugen. Das Kommando CREATE DATABASE legt bei der Erzeugung der Datenbank die Art der zu verwendenden Transaktionsprotokollierung fest. Transaktionsprotokollierung ist im Kapitel 4 erläutert.

Der Datenbankserver **INFORMIX-OnLine Dynamic Server** unterscheidet sich von anderen Datenbankserver in der Art, wie er Datenbanken und Tabellen erzeugt. **INFORMIX-OnLine Dynamic Server** wird zuerst behandelt.

## Die Verwendung von CREATE DATABASE unter INFORMIX-OnLine Dynamic Server

Wenn der Datenbankserver **INFORMIX-OnLine Dynamic Server** eine Datenbank erzeugt, stellt er Records auf, die das Vorhandensein der Datenbank und die Art der Protokollierung anzeigen. Er verwaltet den Speicherplatz direkt, so daß diese Records für Betriebssystemkommandos nicht sichtbar sind.

### Namenskonflikte vermeiden

Normalerweise läuft **INFORMIX-OnLine Dynamic Server** auf einem Rechner nur einmal; **INFORMIX-OnLine Dynamic Server** verwaltet die Datenbanken, die den Benutzern dieses Rechners gehören. **INFORMIX-OnLine Dynamic Server** hält nur eine Liste mit den Datenbank-Namen. Der Name einer Datenbank muß sich von dem jeder anderen Datenbank unterscheiden, die von diesem Datenbankserver verwaltet wird. (Es ist möglich, mehr als einen Datenbankserver laufen zu haben. Dies wird manchmal gemacht, um z. B. getrennt von den aktuell benutzten Daten eine sichere Testumgebung zu schaffen.)

## Auswahl eines Dbspace

**INFORMIX-OnLine** bietet die Möglichkeit, eine Datenbank in einem bestimmten *Dbspace* zu erzeugen. Ein Dbspace ist ein benannter Bereich des Plattenspeichers. Fragen Sie den **INFORMIX-OnLine Dynamic Server** Administrator, ob Sie einen bestimmten Dbspace verwenden sollen. Der Administrator kann eine Datenbank in einem Dbspace unterbringen, um sie von anderen Datenbanken zu trennen, oder um sie in einer bestimmten Gerätedatei (Device) der Platte abzulegen (Kapitel 10 erläutert Dbspaces und deren Beziehung zu den Gerätedateien der Platte.).

Einige Dbspaces sind *gespiegelt* (die doppelte Speicherung auf zwei unterschiedlichen Devices erhöht die Zuverlässigkeit). Wenn der Inhalt einer Datenbank sehr wichtig ist, kann man die Datenbank in einem gespiegelten Dbspace unterbringen.

## Auswahl der Protokollierungsart

**INFORMIX-OnLine** bietet für die Transaktionsprotokollierung folgende vier Möglichkeiten:

- Überhaupt keine Protokollierung. Diese Möglichkeit ist nicht empfehlenswert; wenn die Datenbank auf Grund eines Hardware-Fehlers verloren geht, sind alle Datenänderungen seit der letzten Archivierung verloren.

```
CREATE DATABASE db_with_no_log
```

  Wenn Sie ohne Protokollierung arbeiten, sind die Anweisung BEGIN WORK und andere SQL-Anweisungen, die sich auf die Transaktionsverarbeitung beziehen, nicht erlaubt. Dies wirkt sich auf die Logik der Programme aus, die die Datenbank verwenden.

- Einfache (ungepufferte) Protokollierung. Dies ist bei den meisten Datenbanken die beste Wahl. Im Fehlerfall sind nur die nicht beendeten Transaktionen verloren.

```
CREATE DATABASE a_logged_db WITH LOG
```

- Gepufferte Protokollierung. Wenn die Datenbank verloren geht, sind nur einige der erst kürzlich erfolgten Änderungen verloren, möglicherweise

aber auch keine. Dem geringen Risiko steht eine leicht verbesserte Performance gegenüber.

```
CREATE DATABASE buf_log_db WITH BUFFERED LOG
```

Gepufferte Protokollierung ist bei häufig aktualisierten Datenbanken (bei denen die Geschwindigkeit der Aktualisierung wichtig ist) die beste Wahl. Im Falle eines Absturzes können die Aktualisierungen aus anderen Daten wieder nachvollzogen werden. Mit der Anweisung SET LOG kann man zwischen gepufferter und einfacher Protokollierung wechseln.

- ANSI-kompatible Protokollierung. Diese ist mit der einfachen Protokollierung gleichbedeutend, sie erfordert aber auch die ANSI-Regeln für die Transaktionsverarbeitung. (Siehe die Erörterung von ANSI SQL in Kapitel des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen.*)

```
CREATE DATABASE std_rules_db WITH LOG MODE ANSI
```

Die Entwicklung von ANSI SQL verbietet die Verwendung gepufferter Protokollierung.

Der Administrator kann die Transaktionsprotokollierung später ein- und ausschalten. Beispielsweise kann sie vor dem Einfügen vieler neuer Datensätze ausgeschaltet werden.

## Die Verwendung von CREATE DATABASE bei anderen Datenbankservern

Andere Informix-Datenbankserver erzeugen eine Datenbank in Form einer oder mehrere Dateien, die vom Betriebssystem verwaltet werden. Beispielsweise ist eine Datenbank unter dem Betriebssystem UNIX eine kleine Gruppe von Dateien, die in einem Dateiverzeichnis stehen. Der Name des Verzeichnisses ist der Name der Datenbank. (Im Handbuch Ihres Datenbankservers erhalten Sie detaillierte Informationen darüber, wie der Server Dateien verwendet.) Dies bedeutet, daß die Regeln für Datenbank-Namen dieselben sind wie die, die das Betriebssystem für Dateinamen hat.

### Auswahl der Protokollierungsart

Andere Datenbankserver bieten die folgenden drei Protokollierungsmöglichkeiten an:

- Überhaupt keine Protokollierung. Diese Möglichkeit ist nicht empfehlenswert. Wenn die Datenbank auf Grund eines Hardware-Fehlers verloren geht, sind alle Datenänderungen seit der letzten Archivierung verloren.

```
CREATE DATABASE not_logged_db
```

  Wenn Sie ohne Protokollierung arbeiten, sind die Anweisung BEGIN WORK und andere SQL-Anweisungen, die sich auf die Transaktionsverarbeitung beziehen, nicht erlaubt. Dies wirkt sich auf die Logik der Programme aus, die die Datenbank verwenden.

- Einfache Protokollierung. Dies ist bei den meisten Datenbanken die beste Wahl. Wenn die Datenbank verloren geht, ist nur die Änderung verloren, die beim Auftreten des Fehlers durchgeführt wurde.

```
CREATE DATABASE a_logged_db WITH LOG IN 'a-log-file'
```

  Sie müssen eine Datei festlegen, die das Transaktionprotokoll enthält. (Die Gestaltung des Dateinamens hängt von den Regeln Ihres Betriebssystems ab.) Die Datei wächst bei jeder Änderung der Datenbank. Bei jeder Archivierung der Datenbank-Dateien sollte man die Protokolldatei leeren, so daß sie nur die Transaktionen anzeigt, die nach der letzten Archivierung durchgeführt wurden.

- ANSI-kompatible Protokollierung. Diese stimmt mit der einfachen Protokollierung überein, sie ermöglicht aber auch die ANSI-Regeln für die Transaktionsverarbeitung. (Siehe die Erläuterung der ANSI-kompatiblen Datenbanken in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen.*)

```
CREATE DATABASE std_rules_db WITH LOG IN 'a-...   file' MODE
ANSI
```

Man kann mit der Anweisung START DATABASE einer nicht protokollier. Datenbank nachträglich ein Transaktionsprotokoll hinzufügen.

# Die Verwendung von CREATE TABLE

Mit der CREATE TABLE-Anweisung können Sie alle Tabellen erstellen, die Sie als Datenmodell entworfen haben. Für jede Spalte muß man folgende Informationen angeben:

- Den Namen der Spalte
- Den Datentyp (entsprechend der erstellten Wertebereichs-Liste)
- Wenn die Spalte (oder Spalten) ein Primärschlüssel ist (sind), den Constraint PRIMARY KEY
- Wenn die Spalte (oder Spalten) ein Fremdschlüssel ist (sind), den Constraint FOREIGN KEY
- Wenn die Spalte kein Primärschlüssel ist und keine NULL-Werte zulassen soll, den Constraint NOT NULL
- Wenn die Spalte kein Primärschlüssel ist und keine doppelten Werte zulassen soll, den Constraint UNIQUE
- Wenn die Spalte einen Standardwert hat, den Constraint DEFAULT
- Wenn die Spalte über einen Prüf-Constraint verfügt, den Constraint CHECK

Zusammengefaßt kann man sagen, daß die CREATE TABLE-Anweisung ein in Worte gefaßtes Abbild der Tabelle des Datenmodell-Diagramms ist. Bild 9-5 zeigt die Anweisungen für das Adreßbuch-Modell, dessen Diagramm Bild 8-17 auf Seite 8-34 zeigt.

```
CREATE TABLE name
        (
        satz_nr SERIAL PRIMARY KEY,
        v_name CHAR(20),
        n_name CHAR(20),
        geb DATE,
        jubil DATE,
        e_mail VARCHAR(25)
        );

CREATE TABLE kind
        (
        kind CHAR(20),
        satz_nr INT,
        FOREIGN KEY (satz_nr) REFERENCES name (satz_nr)
        );

CREATE TABLE adresse
        (
        code SERIAL PRIMARY KEY,
        satz_nr INT,
        strasse VARCHAR (50,20),
        ort VARCHAR (40,10),
        land CHAR(5) DEFAULT 'CA',
        postleitzahl CHAR(10),
        FOREIGN KEY (satz_nr) REFERENCES name (satz_nr)
        );

CREATE TABLE telefon
        (
        telefon_nr CHAR(13) PRIMARY KEY,
        telefon_typ CHAR(10),
        satz_nr INT,
        FOREIGN KEY (satz_nr) REFERENCES name (satz_nr)
        );

CREATE TABLE fax
        (
        fax_nr CHAR(13),
        benutz_von DATETIME HOUR TO MINUTE,
        benutz_bis DATETIME HOUR TO MINUTE,
        PRIMARY KEY (fax_nr)
        );

CREATE TABLE faxname
        (
        fax_nr CHAR(13),
        satz_nr INT,
        PRIMARY KEY (fax_nr, satz_nr),
        FOREIGN KEY (fax_nr) REFERENCES fax (fax_nr),
        FOREIGN KEY (satz_nr) REFERENCES name (satz_nr)
        );

CREATE TABLE modem
        (
        modem_nr CHAR(13) PRIMARY KEY,
        satz_nr INT,
        b_type CHAR(5),
        FOREIGN KEY (satz_nr) REFERENCES name (satz_nr)
        );
```

*Bild 9-5* *Die CREATE TABLE-Anweisungen für das Adreßbuch-Datenmodell*

# Die Verwendung von Anweisungsdateien

Man kann eine Datenbank und die Tabellen erstellen, indem man die Anweisungen interaktiv eingibt, beispielsweise in **INFORMIX-SQL** oder in **DB-Access**. Was aber, wenn man dies noch einmal tun muß, oder sogar mehrere Male?

Um Zeit zu sparen und um Fehlerquellen zu vermeiden, kann man alleAnweisungen für die Erstellung einer Datenbank in eine Datei schreiben und dieseAnweisungen automatisch bei Bedarf immer wieder ausführen.

## Die Anweisungsdatei „automatisch" erzeugen

Man kann die Anweisungen für die Implementierung des Modells selbst in eine Datei schreiben. Man kann dies jedoch auch von einem Programm machen lassen. Schlagen Sie im Handbuch *SQL-Sprachbeschreibung, Nachschlagen* im Kapitel 5 nach. Es dokumentiert das Dienstprogramm **dbschema**; dieses Programm untersucht die Struktur einer Datenbank und erzeugt all die SQL-Anweisungen, die für ihre Wiedererstellung erforderlich sind. Die erste Version der Datenbank können Sie *interaktiv* eingeben, beispielsweise in **INFORMIX-SQL** oder in **DB-Access**. Anschließend können Sie mit **dbschema** automatisch eine Anweisungsdatei erzeugen, die Sie dann nach Bedarf verändern und verbessern können.

## Die Datei ausführen

Die Programme **DB-Access** oder **INFORMIX-SQL**, die man für die interaktive Eingabe von SQL-Anweisungen verwendet, kann man aus einer Anweisungsdatei heraus aufrufen. Die Verwendung dieser Produkte auf Betriebssystem-Ebene ist in dem Handbuch **DB-Access** bzw. **INFORMIX-SQL** beschrieben. Man kann **DB-Access** oder **INFORMIX-SQL** aufrufen und diese eine Anweisungsdatei lesen und ausführen lassen. Die Anweisungsdatei kann von Ihnen oder von **dbschema** erstellt worden sein.

## Ein Beispiel

Die meisten Informix-Produkte werden mit der Beispiel-Datenbank **stores6** ausgeliefert (der Datenbank, die in den meisten Beispielen diese Handbuchs verwendet wird). Die Datenbank **stores**6 wird in Form einer Anweisungsdatei des Betriebssystems geliefert, die für die Erstellung der Datenbank die jeweiligen Informix-Produkte aufruft. Sie können diese kopieren und als Grundlage für die Automatisierung Ihres eigenen Datenmodells verwenden.

# Die Tabellen füllen

```
INSERT INTO hersteller VALUES ('MKL', 'Martin', 15)
```

Oder man kann die Eingabe der Sätze auch über ein mit **INFORMIX-4GL** oder in einer anderen Sprache geschriebenes Anwendungsprogramm durchführen.

Häufig können die ersten Sätze einer großen Tabelle aus Daten abgeleitet werden, die in Tabellen einer anderen Datenbank oder in Dateien des Betriebssystems gespeichert werden. Man kann die Daten in die neue Datenbank mit einer entsprechenden Anweisung komplett übertragen. Wenn sich die Daten in einer anderen Datenbank befinden, kann man diese auf mehrere Arten abrufen.

Wenn man **INFORMIX-OnLine** verwendet, kann man die gewünschten Daten aus der anderen Datenbank, die auf einem *anderen* Datenbankserver liegt, einfach auswählen, indem man diese als Teil einer INSERT-Anweisung in der Datenbank angibt. Beispielsweise könnten Sie Sätze aus der Tabelle **position** der Datenbank **stores6** in die neue Tabelle übernehmen.

```
INSERT INTO newtable
    SELECT position_nr, auftrag_nr, menge, artikel_nr,
        herst_kennz, ges_preis
    FROM stores6@otherserver:position
```

Bei **INFORMIX-SE** können Sie Daten aus jeder beliebigen Datenbank in die aktuelle Datenbank holen, solange beide Datenbanken auf dem gleichen Server liegen. Beispielsweise könnten Sie über eine temporäre Tabelle Information aus der Tabelle **katalog** der Datenbank **stores6** in eine neue Tabelle überspielen.

```
CONNECT TO 'sharky/db1';

SELECT catalog_num, stock_num, manu_code
    FROM catalog
    INTO temptable;

DISCONNECT;

CONNECT TO 'sharky/db2';

SELECT * from temptable
    INTO newsetable;
```

Wenn Sie unter **INFORMIX-SE** Daten aus einer anderen Datenbank, die auf einem anderen Datenbankserver liegt, in die aktuelle Datenbank importieren wollen, müssen Sie die Daten zunächst in eine Datei schreiben. Hierfür kann man das Kommando UNLOAD von **DB-Access, INFORMIX-SQL** und **INFORMIX-4GL** verwenden, oder man kann eine Liste über **ACE** oder **INFORMIX-4GL** erstellen und die Ausgabe in eine Datei leiten.

Wenn die Quelle eine andere Dateiart oder Datenbank ist, muß man die Daten in eine ASCII-Datei umwandeln; dies ist eine Datei mit druckbaren Daten, in der jede Zeile den Inhalt eines Satzes einer Tabelle repräsentiert.

Sobald die Daten in einer Datei stehen, kann man diese unter Verwendung des Dienstprogramms **dbload** in eine Tabelle einfügen (laden). Informationen über **dbload** erhalten Sie in Kapitel 5 des Handbuches *SQL-Sprachbeschreibung, Nachschlagen*. Das LOAD-Kommando von **DB-Access, INFORMIX-SQL** oder **INFORMIX-4GL** kann auch Sätze aus einer normalen ASCII-Datei laden. Über die Anweisungen LOAD und UNLOAD erfahren Sie mehr im Kapitel 1 des Handbuches *SQL-Sprachbeschreibung, Syntax*.

Das Einfügen hunderter oder tausender Sätze geht viel schneller, wenn man die Transaktionsprotokollierung ausschaltet. Es gibt keine Veranlassung, diese Einfügungen überhaupt zu protokollieren, weil man im Falle eines Fehlers das Verlorengegangene leicht wieder neu erzeugen kann. Hier die einzelnen Schritte für das komplette Laden einer großen Anzahl von Sätzen:

- Wenn die Möglichkeit besteht, daß andere Benutzer die Datenbank verwenden, dann schließen Sie diese mit der Anweisung DATABASE EXCLUSIVE aus.
- Wenn Sie **INFORMIX-OnLine** verwenden, dann bitten Sie den Administrator, die Protokollierung für die Datenbank auszuschalten.

  Die bisher vorhandenen Protokolle können für die Wiederherstellung der Datenbank in ihren aktuellen Zustand verwendet werden. Falls es beim „Masseneinfügen" Probleme gibt, kann man diesen Vorgang jederzeit leicht erneut ausführen. Man benötigt kein Protokoll, um diese Aktion zu wiederholen.
- Führen Sie die Anweisungen durch oder lassen Sie die Dienstprogramme laufen, die die Daten in die Tabellen einfügen.
- Archivieren Sie die neu geladene Datenbank.

  Wenn Sie **INFORMIX-OnLine** verwenden, bitten Sie entweder den Administrator um die Durchführung einer Komplett- oder Differenzsicherung oder verwenden Sie das Dienstprogramm **onunload**, um eine binäre Kopie Ihrer Datenbank anzufertigen.

  Wenn Sie andere Datenbankserver verwenden, dann verwenden Sie Betriebssystem-Kommandos für die Sicherung der Dateien, die die Datenbank repräsentieren.
- Stellen Sie die Transaktionsprotokollierung wieder her und heben Sie die Exklusiv-Sperre auf die Datenbank auf.

Die einzelnen Schritte zum Füllen einer Datenbank können Sie in einer Prozedur-Datei zusammenfassen. Man kann die **INFORMIX-OnLine** Verwaltung dadurch automatisieren, daß man die dem ON-Monitor entsprechenden Kommandos auf Betriebssystem-Ebene ausführt.

# Zusammenfassung

Dieses Kapitel beschrieb die Aufgaben, die man bei der Implementierung eines Datenmodells durchführen muß:

- Angabe der im Modell verwendeten Wertebereiche oder Constraints und Vervollständigung des Modells durch die Zuweisung von Constraints für jede Spalte.
- Verwendung von interaktivem SQL, um die Datenbank und die enthaltenen Tabellen zu erzeugen.
- Wenn man eine Datenbank voraussichtlich später noch einmal erstellen muß, schreibt man die SQL-Anweisungen hierzu in eine Anweisungsdatei.
- Füllen der Tabellen des Modells, zuerst unter Verwendung von interaktivem SQL und anschließend mit Anweisungen zur automatischen Einfügung von vielen Datensätzen.
- Möglicherweise schreibt man die Anweisungen zur automatischen Einfügung von vielen Datensätzen in eine Anweisungsdatei, um sie so leicht wiederholen zu können.

Jetzt können Sie Ihr Datenmodell verwenden und überprüfen. Falls es sehr große Tabellen enthält oder falls Sie Teile davon vor bestimmten Benutzerklassen schützen müssen, gibt es weitere Aufgaben zu bewältigen. Dies ist das Thema des nächsten Kapitels.

Kapitel

# 10

# Das Modell tunen

# Kapitelüberblick

Die vorangegangenen Kapitel haben die im Prinzip erforderlichen Schritte für die Erzeugung und die Implementierung eines Datenmodells zusammengefaßt. Normalerweise verfügt die daraus resultierende Datenbank über eine ausreichende Performance. Ist dies der Fall, so ist diese Arbeit beendet.

Manche Anwendungen haben jedoch strengere Anforderungen an die Performance. Einige Datenbanken enthalten extrem große Tabellen. Andere müssen von vielen Programmen gleichzeitig verwendbar sein oder von Programmen, die mit sehr hohen Anforderungen in Bezug auf die Antwortzeit arbeiten; und wieder andere müssen extrem zuverlässig sein.

Um solche Anforderungen zu erfüllen, kann man das Datenbank-Design tunen. In einigen Fällen muß man eventuell sogar die theoretische Korrektheit des Datenmodells opfern, um die gewünschte Performance zu erzielen.

Bevor Sie jedoch irgendeine Änderung vornehmen, sollten Sie sicher sein, daß die Abfragen so effektiv wie möglich sind. In Kapitel 13, "Abfragen optimieren," erfahren Sie, wie Sie die Performance verbessern können ohne das Datenmodell zu verändern.

Dieses Kapitel enthält Erörterungen zu folgenden Themen:

- Speichermethoden, die von **INFORMIX-OnLine** verwendet werden, sowie deren Auswirkung auf die Performance und Sicherheit
- Methoden zur Berechnung der Tabellen- und Indexgröße und die Verwendung dieser Werte, um die Ausführungszeit einer Abfrage abzuschätzen
- Nutzen und Aufwand durch Hinzufügen von Indizes
- Möglichkeiten zur "Denormalisierung" eines Datenmodells; gemeint ist damit die Zerstörung der theoretischen Korrektheit eines Datenmodells, also der Normalform, zugunsten einer verbesserten Performance
- Vorschläge zur Verbesserung der Parallelverarbeitung

# INFORMIX-OnLine Dynamic Server Plattenspeicher

**INFORMIX-OnLine Dynamic Server** verwaltet den Plattenspeicher direkt. Bei maximaler Ausnutzung der Möglichkeiten umgeht **OnLine** das Betriebssystem und arbeitet mit einem selbst verwalteten Plattenspeicher. (Die jeweils mögliche Stufe der Ausnutzung hängt vom Betriebssystem ab.)

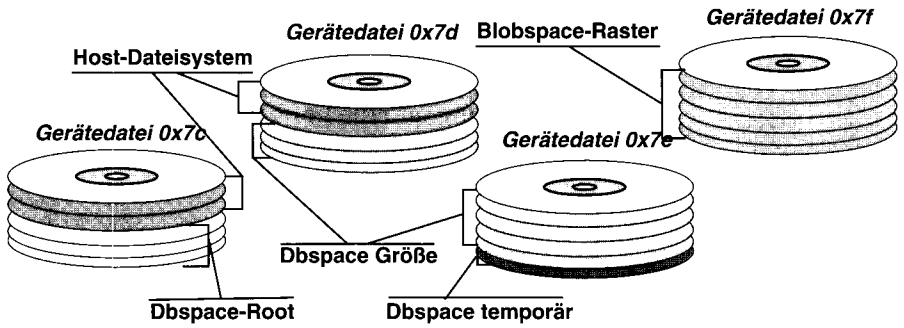



*Bild 10-1* *Die Beziehung zwischen Chunks und Dbspaces*

Die Methoden des Datenbankservers sind nicht kompliziert. Es werden jedoch mehrere unterschiedliche Begriffe für verschiedene Bereiche in unterschiedlichen Zusammenhängen verwendet: *Chunk, Dbspace* (bzw. *Blobspace*), *Page, Extent* und *Tblspace.*

## Chunks und Pages

Die zugrunde liegende Einheit der physikalischen Speicherung ist ein *Chunk.* Ein Chunk ist eine Einheit des Plattenspeichers, die für **INFORMIX-OnLine Dynamic Server** reserviert ist. Gewöhnlich ist dies ein physikalisches Gerät, d. h. ein Plattenlaufwerk (Device). Ein Chunk kann jedoch auch ein Teil einer Platte oder nur eine Datei sein. Ein Chunk ist auf jeden Fall eine Bereichseinheit, die vom Betriebssystem als eine Einheit erkannt wird und die unter der exklusiven Kontrolle von **INFORMIX-OnLine Dynamic Server** steht.

Eine *Page* ist die Grundeinheit für den Platten-Input/-Output (I/O). Der gesamte Platz eines Chunk wird in Pages eingeteilt. Alle I/O werden in Einheiten von ganzen Pages durchgeführt. Die Größe einer Page ist in allen Chunks gleich. Sie wird bei der Installation des **INFORMIX-OnLine**-Systems gesetzt.

## Dbspaces und Blobspaces

Ein *Space (Bereich)* umfaßt immer einen oder mehrere vollständige Chunks. Häufig enthält ein Space genau einen Chunk, so daß der Space und der Chunk identisch sind. Wenn jedoch ein Space mehr als einen Chunk umfaßt, läßt **INFORMIX-OnLine** mehrere Chunks wie eine einzige Folge von Pages erscheinen.

Ein Space ist entweder ein *Dbspace* oder ein *Blobspace*; dies ist von der Verwendung abhängig. Wenn ein Space Datenbanken und Tabellen enthält, ist es ein Dbspace. Wenn er für die Speicherung von Binary Large Objects (den BLOB-Datentypen TEXT und BYTE) vorgesehen ist, ist er ein Blobspace. Chunks werden einem Space bei dessen Erzeugung zugewiesen. Chunks können einem Space aber auch nachträglich hinzugefügt werden.

Ein bestimmter Dbspace, der *Root* -Dbspace, wird zuerst erzeugt. Der Root-Dbspace ist der wichtigste Dbspace, weil er die Kontrollinformationen speichert, die alle anderen Dbpaces und Chunks beschreiben.

Teile einer einzigen Datenbank können in zwei oder mehr Dbspaces vorkommen. Eine einzelne Tabelle ist jedoch immer vollständig in einem Dbspace enthalten.

## Plattenspiegelung

Einzelne Spaces können *gespiegelt* werden. Die Chunks eines gespiegelten Bereichs sind paarweise mit anderen Chunks gleicher Größe verbunden. Jedesmal, wenn eine Page in einen der Chunks geschrieben wird, wird sie auch in den gespiegelten Chunk geschrieben. Wenn ein Chunk wegen eines Hardwarefehlers ausfällt, verwendet der Datenbankserver die Spiegelungen, um die Verarbeitung ohne Unterbrechung fortsetzen zu können. Wenn der ausgefallene Chunk wieder betriebsbereit ist (oder wenn ein Ersatz-Chunk zugewiesen wurde), setzt **INFORMIX-OnLine** diesen automatisch auf den gleichen Stand wie den arbeitenden Chunk und setzt die Verarbeitung fort.

Wenn Dbspaces gespiegelt werden, dann sollte auch der Root-Dbspace gespiegelt werden. Wenn er auf Grund eines Hardwareausfalls verloren gegangen ist, sind ansonsten alle **INFORMIX-OnLine**-Daten unbrauchbar, unabhängig von den Spiegelungen.

Wenn Ihre Datenbank in Bezug auf einen Hardwareausfall extreme Anforderungen an die Zuverlässigkeit stellt, sollten Sie diese in einem gespiegelten Dbspace unterbringen. Wie in den folgenden Abschnitten gezeigt, ist es möglich, einzelne Tabellen in bestimmten Dbspaces einzurichten. Man kann also einige Tabellen in gespiegelten Bereichen und andere Tabellen in normalen Bereichen einrichten.

# Datenbanken

Eine Datenbank befindet sich anfangs in einem einzigen Dbspace. Sie wird dort über einen Parameter der CREATE DATABASE-Anweisung angelegt. Im folgenden Beispiel wird eine Datenbank in dem Dbspace **dev0x2d** erzeugt:

```
CREATE DATABASE reliable IN dev0x2d WITH BUFFERED LOG
```

Wenn kein Dbspace angegeben ist, wird die Datenbank im Root-Dbspace angelegt. Wenn sich eine Datenbank in einem Dbspace befindet, bedeutet dies nur, daß

- die Systemtabellen in diesem Dbspace gespeichert werden
- dieser Dbspace der Standardplatz für Tabellen ist, die nicht explizit in einem anderen Dbspace erstellt wurden.

# Tabellen und Bereiche

Eine Tabelle befindet sich vollständig in einem einzigen Dbspace (Die zugehörigen Blob-Werte können in separaten Blobspaces oder in einem optischen Speichersystem liegen.). Wenn kein Dbspace angegeben ist, befindet sich eine Tabelle in dem Dbspace, in dem sich auch die Datenbank befindet. In dem folgenden Beispiel wird eine Tabelle im Dbspace **misctabs** erzeugt:

```
CREATE TABLE taxrates (…column specifications…) IN misctabs
```

Durch das Ablegen einer Tabelle in einem bestimmten Dbspace kann man eine Reihe von unterschiedlichen Zielen erreichen. Einige dieser Ziele werden in den folgenden Abschnitten erläutert.

## Die Spiegelung nutzen

Legen Sie alle Tabellen, die von äußerst wichtigen Anwendungen verwendet werden, in einem gespiegelten Dbspace an. Alternativ hierzu kann man die Datenbank in einem gespiegelten Dbspace erzeugen und die wichtigen Tabellen standardmäßig dort anlegen. Die Tabellen, die zwar Bestandteil der Datenbank sind, aber nicht von wichtigen Programmen verwendet werden, kann man in Bereichen (Spaces) anlegen, die nicht gespiegelt werden.

### Temporären Bereich gemeinsam nutzen

Wenn Datenbanken sehr groß sind und der Platz auf der Platte knapp ist, dann könnte in den normalen Dbspaces nicht genug Platz für große temporäre Tabellen sein, die für die Optimierung der Abfrage-Performance sinnvoll sein können. Eine temporäre Tabelle ist jedoch nur während der Laufzeit des Programmes vorhanden, das diese Tabelle erzeugt hat – oder sogar kürzer, wenn das Programm die Tabelle sofort löscht. Man sollte einen einzelnen Dbspace für temporäre Tabellen einrichten. Dieser kann von vielen Programmen immer wieder verwendet werden.

### Zugeordnete Hardware festlegen

Ein Dbspace kann einem Chunk entsprechen und ein Chunk kann einer Gerätedatei entsprechen. Somit kann man eine Tabelle in einer Platten-Gerätedatei anlegen, die dieser zur Benutzung zugeordnet wurde. Wenn Plattenlaufwerke unterschiedliche Performance-Stufen aufweisen, kann man die am häufigsten benutzten Tabellen in den schnellsten Laufwerken unterbringen.

Indem man einer häufig verwendeten Tabelle einen Schreib-/Lesekopf zuordnet, kann man Konflikte mit Anwendungen verringern, die andere Tabellen verwenden. Dies verringert aber nicht die Konflikte zwischen Programmen, die dieselben Tabellen verwenden; dies geschieht erst, wenn sich die Tabelle über mehrere Gerätedateien erstreckt, so wie es im nächsten Abschnitt vorgestellt wird.

### Konflikte beim gemeinsamen Zugriff mehrerer Programme reduzieren

Ein Dbspace kann mehrere Chunks umfassen und jeder Chunk kann eine unterschiedliche Platte repräsentieren. Deswegen kann man einer einzigen Tabelle mehrere Schreib-/Leseköpfe zuweisen. Bild 10-2 zeigt hierzu ein Beispiel.

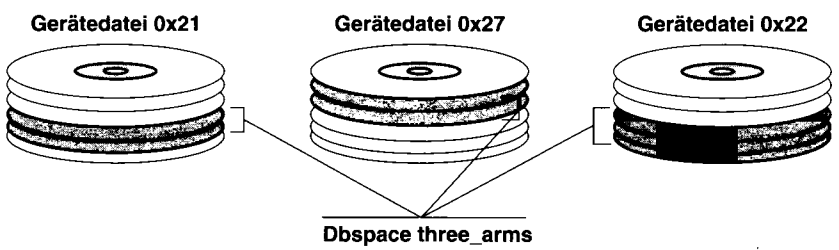



*Bild 10-2* *Ein Dbspace, der über drei Platten verteilt ist*

Der Dbspace **three_arms** besteht aus Chunks auf drei verschiedenen Platten. Die Größe eines jeden Chunk beträgt ungefähr ein Drittel der Größe einer wichtigen, häufig verwendeten Tabelle. (Der dritte Chunk ist größer, um eine Erweiterung zu ermöglichen.) Fragen mehrere Programme die Tabelle ab, so werden deren I/O-Operationen über die drei Platten verteilt; dies verringert die Konflikte bei der Verwendung der Hardware.

In einer Anordnung wie dieser können die Platten untereinander als Spiegel für eine andere Platte fungieren. Beispielsweise kann die Gerätedatei **0x21** auf **0x27** gespiegelt werden, **0x27** auf **0x22** und **0x22** auf **0x21**. Solange zwei Gerätedateien betriebsbereit sind, ist der Dbspace verwendbar. (Wenn jedoch ein einzelnes Hardwareteil, wie z. B. ein Plattencontroller, zu allen drei Platten gehört, ist diese Art der Spiegelanordnung immer noch ungeschützt, wenn diese eine Komponente ausfällt.)

# Tblspaces

Der gesamte Plattenbereich, der einer Tabelle zugeteilt wird, ist der *Tblspace* der Tabelle. Der Tblspace beinhaltet die Pages, die den Datensätzen zugeteilt wurden und die Pages, die den Indizes zugeteilt wurden. Er beinhaltet auch die Pages, die von den Blob-Spalten verwendet werden. Er beinhaltet aber keine Pages, die von Blob-Daten in einem separaten Blobspace verwendet werden oder die von Blob-Daten in einem optischen Speichersystem gespeichert werden. (Diese Möglichkeiten werden im Abschnitt "Blob-Daten anlegen" auf Seite 10-21 erläutert.)

Der Tblspace ist nur ein Aufzeichnungsobjekt; er entspricht keinem bestimmten Teil eines Dbspace. Die Indizes und Daten-Extents, die eine Tabelle bilden, können im Dbspace verteilt sein.

Das Dienstprogramm **oncheck** (mit der Option **-pt**) liefert Informationen über den Zustand der Tblspaces zurück; diese Informationen beinhalten die Anzahl der zugeteilten Pages und deren Verwendung. Man kann diese Informationen verwenden, um das Wachstum einer Tabelle innerhalb einer bestimmte Zeit zu überwachen.

# Extents

Sobald man in eine Tabelle Sätze einfügt, weist **INFORMIX-OnLine Dynamic Server** dieser Tabelle Plattenbereiche in Einheiten zu, die *Extents* genannt werden. Jeder Extent ist ein Block physikalisch aufeinanderfolgender Pages in dem Dbspace. Selbst wenn ein Dbspace mehr als einen Chunk umfaßt, werden Extents immer vollständig in einem einzigen Chunk zugewiesen, damit sie aufeinanderfolgend bleiben.

Die unmittelbare Nachbarschaft ist für die Performance wichtig. Wenn die Daten-Pages aufeinanderfolgen, sind die Bewegungen des Schreib-/Lesekopfes beim sequentiellen Lesen der Sätze minimal. Der Extent-Mechanismus ist ein Kompromiß zwischen den folgenden konkurrierenden Anforderungen:

- Die meisten Dbspaces werden von mehreren Tabellen gemeinsam benutzt.
- Die Größe einiger Tabellen ist im voraus nicht bekannt.
- Tabellen können zu unterschiedlichen Zeiten und mit unterschiedlicher Geschwindigkeit wachsen.
- Alle Pages einer Tabelle sollten aus Gründen der Performance nebeneinander liegen.

Da die Tabellengrößen nicht bekannt sind, kann der Platz einer Tabelle nicht im voraus zugeteilt werden. Aus diesem Grund werden Extents nur dann hinzugefügt, wenn sie benötigt werden. Wegen der besseren Performance sind aber in jedem Extent alle Pages aufeinanderfolgend.

## Die Extent-Größen auswählen

Beim Erzeugen einer Tabelle kann man die Größe des ersten Extent ebenso bestimmen wie die Größe derjenigen Extents, die beim Wachsen der Tabelle hinzugefügt werden. Die Größe der hinzugefügten Extents kann man nachträglich mit der Anweisung ALTER TABLE ändern. Im folgenden Beispiel wird eine Tabelle erzeugt, bei der die Größe des Anfangs-Extents 512 KByte und die der hinzugefügten Extents 100 KByte beträgt:

```
CREATE TABLE big_one (…column specifications…)
    IN big_space
    EXTENT SIZE 512
    NEXT SIZE 100
```

Im folgenden Beispiel wird die Größe des Folge-Extents auf 50 KByte geändert. Dies hat keine Auswirkung auf bereits vorhandene Extents.

```
ALTER TABLE big_one MODIFY NEXT SIZE 50
```

Die Größen der Folge-Extents sind bei folgenden Tabellenarten für die Performance unbedeutend:

- Eine kleine Tabelle hat nur einen Extent (ansonsten wäre sie nicht klein). Wenn auf sie häufig zugegriffen wird, befinden sich große Teile der Tabelle im Speicher.
- Eine Tabelle, auf die selten zugegriffen wird, ist nicht wichtig für die Performance; dabei spielt die Größe keine Rolle.
- Eine Tabelle, die in einem zugewiesenen Dbspace liegt, erhält immer neue Extents, die neben den alten liegen. Die Größe dieser Extents ist nicht wichtig, weil sie – da sie nebeneinander liegen – wie ein großer Extent erscheinen.

Wenn man einer dieser Tabellenarten eine Extent-Größe zuweist, muß man nur darauf achten, daß man die Erstellung zu vieler Extents vermeidet. Eine große Anzahl an Extents zwingt den Datenbankserver dazu, zusätzlich Zeit für die Verwaltung aufzuwenden. Außerdem gibt es für die Anzahl der zulässigen Extents eine obere Grenze. (Dies wird im Abschnitt "Obere Grenze für Extents" auf Seite 10-11 beschrieben.)

Die Größen der Folge-Extents werden dann wichtig, wenn zwei oder mehr große, wachsende Tabellen einen Dbspace gemeinsam nutzen. Weil die Extents, die den Tabellen hinzugefügt werden, dazwischengeschoben werden, verlängert sich mit jedem neuen Extent die Zugriffszeit beim sequentiellen Lesen. Dies erhöht auch den Gesamtweg, den der Schreib-/Lesekopf beim nicht sequentiellen Lesen zurücklegen muß.

Die Größe eines Extents ist nur durch die Größe des Dbspace begrenzt. Wenn man die Endgröße einer Tabelle kennt (oder diese mit einer Genauigkeit von 25% voraussagen kann), wird der gesamte Bereich dem Anfangs-Extent (Initial-Extent) zugewiesen. Wenn Tabellen auf eine unbekannte Größe anwachsen, weist man diesen für die Folge-Extents Größen zu, die so gewählt sind, daß die Tabellen den Dbspace jeweils mit einer möglichst niedrigen Extent-Anzahl gemeinsam nutzen. Hier ist ein möglicher Ansatz:

- Entscheiden Sie sich für ein Verhältnis, in dem Tabellen ein Dbspace gemeinsam nutzen sollen. Beispielsweise kann man ein Dbspace unter drei Tabellen im Verhältnis 0,4 : 0,2 : 0,3 (10% sind für kleine und allgemeine Tabellen reserviert) aufteilen.
- Weisen Sie jeder Tabelle ein Viertel des gemeinsam benutzten Bereichs vom Dbspace als Anfangs-Extent zu.
- Weisen Sie jeder Tabelle ein Achtel des gemeinsam benutzten Bereichs vom Dbspace als Größe des Folge-Extent zu.

Kontrollieren Sie regelmäßig mit **oncheck** die Größe der Tabellen.

Was passiert, wenn es bei zunehmender Auffüllung des Dbspace nicht genügend aufeinanderfolgenden Platz gibt, um einen Extent in der von Ihnen gewünschten Größe zu erzeugen? In diesem Fall weist **INFORMIX-OnLine** den größtmöglichen durchgehenden Extent zu.

## Obere Grenze für Extents

Es sollte keiner Tabelle möglich sein, sich eine große Anzahl an Extents zuzulegen. Aus diesem Grund gibt es für die Anzahl der zulässigen Extents eine obere Grenze. Wenn nach Erreichung der Grenze versucht wird, einen Extent hinzuzufügen (um eine INSERT-Anweisung auszuführen), verursacht dies den **ISAM**-Fehler 136 ( `keine Extents mehr`).

Die obere Grenze hängt von der Page-Größe und der Tabellendefinition ab. Um die obere Grenze für Extents für eine bestimmte Tabelle zu ermitteln , errechnet man eine Reihe von Werten wie folgt:

| | |
|---|---|
| *pagesize* = | die Größe der Page (wird bei **onstat -c** unter der Überschrift BUFFSIZE angezeigt) |
| *colspace*= | 4 × die Anzahl der Spalten der Tabelle |
| *ixspace*= | 12 × die Anzahl der Indizes der Tabelle |
| *ixparts*= | 4 × die Anzahl der in jedem Index genannten Spalten |
| *extspace*= | *pagesize* - (*colspace* + *ixspace* + *ixparts* + 84) |

Eine Tabelle kann nicht mehr Extents haben als *extspace* /8. Wenn die Größe einer Page 2.048 ist, dann kann die Tabelle **customer** der Beispiel-Datenbank, wie in Bild 10-3 berechnet, nicht mehr als 234 Extents haben.

Um sicher zu sein, daß die Grenze nicht überschritten ist, überprüft **INFORMIX-OnLine** jedesmal bei der Neuerstellung eines Extent die Anzahl der Extents. Wenn der erzeugte Extent der 16. oder ein Vielfaches von 16 ist, dann wird die Größe des Folge-Extent bei dieser Tabelle automatisch verdoppelt.

| Variable | Wert | Basis |
|---|---|---|
| *pagesize* | 2,048 | Ausgabe von **onstat** |
| *colspace* | 40 | zehn Spalten |
| *ixspace* | 36 | drei Indizes |
| *ixparts* | 12 | eine Spalte pro Index |
| *extspace* | 1,876 | 2,048 - (40 + 36 + 12 + 84) |
| *limit* | 234 | *extspace*/8 |

*Bild 10-3* ***Die verwendeten Werte für die Bestimmung der obersten Grenze von Extents***

# Tabellen reorganisieren

Wenn einer Tabelle viele Extents hinzugefügt werden, kann die Performance bei Abfragen nachlassen. Dies ist dann der Fall, wenn zwei oder mehr große, wachsende Tabellen ein Dbspace gemeinsam nutzen. Wenn Tabellen gleichzeitig wachsen, werden deren neue Extents und Index-Pages dazwischengeschoben; dabei entstehen große Lücken zwischen den Extents einer Tabelle. Diese Situation wird in Bild 10-4 veranschaulicht.


*Bild 10-4* ***Ein fragmentierter Dbspace***

Das Bild zeigt auch Lücken unbenutzter Bereiche, die vielleicht dadurch entstanden sind, daß bei der Zuweisung der Extents Tabellen vorhanden waren, die inzwischen gelöscht wurden.

Ein Durcheinander, so wie in Bild 10-4 beschrieben, verschlechtert die Performance auf zwei Arten. Für einen sequentiellen Zugriff auf eine Tabelle sind lange Zugriffszeiten erforderlich. Bei einem nicht sequentiellen Zugriff auf

eine Tabelle muß die Platte eventuell über den gesamten Dbspace durchsucht werden. Sie können einen Dbspace jedoch neu aufbauen, so daß die Tabellen wieder kompakt vorliegen; dies wird in Bild 10-5 gezeigt. Die jeweilige Reihenfolge der reorganisierten Tabellen innerhalb des Dbspace ist nicht wichtig; das einzig Wichtige ist, daß die Pages jeder Tabelle zusammen liegen. Wenn eine Tabelle sequentiell gelesen wird, gibt es auf diese Weise keine langen Zugriffszeiten. Wenn eine Tabelle nicht sequentiell gelesen wird, muß der Schreib-/Lesekopf nur den Bereich überstreichen, der von der Tabelle belegt ist.


*Bild 10-5* *Einen Dbspace reorganisieren, um die Tabellen kompakter zu machen*

Gehen Sie bei der Reorganisation eines Dbspace folgendermaßen vor:

- Kopieren Sie die Tabellen des Dbspace einzeln auf ein Band; verwenden Sie hierfür das Dienstprogramm **onunload**.
- Löschen Sie alle Tabellen im Dbspace.
- Erzeugen Sie die Tabellen wieder; verwenden Sie dafür das Dienstprogramm **onload**.

Das Dienstprogramm **onload** erzeugt die Tabellen wieder mit den gleichen Eigenschaften, die sie vorher hatten; hierzu gehören auch die gleichen Extent-Größen. Wenn ein neuer Extent neben dem vorherigen Extent erzeugt wird, dann werden die beiden wie ein einziger Extent behandelt.

Man kann eine Tabelle auch mit dem UNLOAD-Kommando von **DB-Access**, **INFORMIX-SQL** oder **INFORMIX-4GL** entladen, und die Tabelle mit dem Kommando LOAD oder dem Dienstprogramm **dbload** zurückladen. Diese Operationen wandeln eine Tabelle jedoch in ein CHAR-Format um, wohingegen **onload** und **onunload** mit Binär-Kopien der Pages arbeiten.

Es gibt zwei weitere Arten, um eine Tabelle zu reorganisieren. Wenn man das Kommando ALTER TABLE verwendet, um eine Spalte hinzuzufügen oder zu löschen oder um einen Datentyp zu ändern, wird die Tabelle kopiert und wieder aufgebaut. Wenn man einen Cluster-Index erzeugt oder einen Index in einen Cluster-Index umwandelt, wird die Tabelle sortiert und neu geschrieben (Siehe "Cluster-Indizes" auf Seite 10-29.). In beiden Fällen wird die

Tabelle in andere Bereiche des Dbspace geschrieben. Wenn sich jedoch andere Tabellen in dem Dbspace befinden, gibt es keine Garantie dafür, daß alle neuen Extents nebeneinander liegen.

# Die Größe von Tabellen berechnen

Dieser Abschnitt erläutert Methoden zur Berechnung der ungefähren Tabellen- und Indexgröße, ausgedrückt in Pages. Wie bei den vorherigen Abschnitten gilt auch dies nur für **INFORMIX-OnLine Dynamic Server**.

Diese Berechnungen haben das Ziel, die Anzahl der Pages zu schätzen, die für die Speicherung der Daten einer Tabelle verwendet werden. Wenn die Tabelle bereits vorhanden ist, oder wenn man eine Beispieltabelle unter Verwendung simulierter Daten mit realistischer Größe erstellen kann, dann muß man keine Schätzungen machen. In diesem Fall können die genauen Zahlen mit dem Dienstprogramm **oncheck** ermittelt werden.

Man kann eine Schätzung über die Berechnung einer Reihe von Werten erhalten. Bei allen Schätzungen braucht man die unten aufgeführten Werte, die wie folgt vereinbart sind:

*estrows=* die geschätzte Anzahl der Sätze, die eine Tabelle beim Erreichen ihrer normalen Größe hat.

*pagesize=* die Größe einer Page (diese wird bei **onstat -c** unter der Überschrift BUFFSIZE angezeigt).

*pageuse=* *pagesize* - 28 (der Platz, der auf einer Page für Daten verfügbar ist).

In den Berechnungen dieses Kapitels bezeichnet `floor`(*x*) den größten ganzzahligen Wert, der kleiner als *x* ist, wohingegen `ceiling`(*x*) den kleinsten ganzzahligen Wert bezeichnet, der größer als *x* ist.

## Sätze mit fester Länge schätzen

Wenn eine Tabelle keine VARCHAR-Spalten enthält, haben alle Sätze dieselbe Größe. Berechnen Sie die folgenden zusätzlichen Variablen. Die Werte sind wie folgt:

*rowsize* = die Summe aus den Größen aller Spalten der Tabelle, plus 4 Byte. Die Größen verschiedener Datentypen sind in Kapitel 9 erläutert. Nehmen Sie bei TEXT- und BYTE-Spalten eine Größe von 56 Byte an.

*homerow*= `if` (*rowsize* ≤ *pageuse*) then *rowsize*
else 4 + Rest von (*rowsize/pageuse*)

Falls ein Satz länger als eine Page ist, werden soviele ganze Seiten wie möglich am Ende des Satzes entfernt und in separaten Erweiterungs-Pages gespeichert. Weisen Sie *homerow* die Größe des führenden Teils zu, der in der Home-Page aufbewahrt ist. Der Wert *homerow* ist einfach *rowsize*, wenn ein Satz in eine Page paßt.

*overpage* = `if` (*rowsize* ≤ *pageuse*) then 0
else floor (*rowsize* / *pageuse*)

Weisen Sie *overpage* die Anzahl der für jeden Satz benötigten Erweiterungs-Pages zu; wenn ein Satz in eine Page paßt, ist die Anzahl Null (0).

*datrows* = min(255,floor (*pageuse* / *homerow*))

Weisen Sie *datrows* die Anzahl der Sätze (oder erste Teile der Sätze) zu, die in eine Page passen. Es gibt die obere Grenze von 255 Sätzen pro Page, selbst wenn die Sätze sehr kurz sind.

*datpages* = ceiling (*estrows* / *datrows*)

*expages* = *estrows* × *overpage*

Eine Tabelle braucht ungefähr *datpages* + *expages* Pages für die Sätze (Hinweise zur Abschätzung zusätzlicher Pages für TEXT- und BYTE-Spalten werden im Abschnitt "Schätzen der Blobpages" auf Seite 10-20 gegeben.). Eine Schätzung der Tabelle **customer** wird in Bild 10-7 gezeigt.

| Variable | Schätzung | Grundlage der Schätzung | | | |
|---|---|---|---|---|---|
| *estrows* | 1.500 | Geschäftsplan | | | |
| *pagesize* | 2.048 | Ausgabe von **onstat** | | | |
| *pageuse* | 2.020 | *pagesize* - 28 | | | |
| *rowsize* | 138 | Summe aus 4+ | **Spalte** | **Datentyp** | **Größe** |
| | | | customer_num | SERIAL | 4 |
| | | | fname | CHAR(15) | 15 |
| | | | lname | CHAR(15) | 15 |
| | | | company | CHAR(20) | 20 |
| | | | address1 | CHAR(20) | 20 |
| | | | address2 | CHAR(20) | 20 |
| | | | city | CHAR(15) | 15 |
| | | | state | CHAR(2) | 2 |
| | | | zipcode | CHAR(5) | 5 |
| | | | phone | CHAR(18) | 18 |
| *homerow* | 134 | *rowsize* ≤ *pageuse* | | | |
| *overpage* | 0 | *rowsize* ≤ *pageuse* | | | |
| *datrows* | 14 | floor (*pageuse*/*homerow*) | | | |
| *datpages* | 108 | ceiling (*estrows*/*datrows*) | | | |
| *expages* | 0 | *estrows* × *overpage* | | | |

***Bild 10-6*** ***Schätzung der Größe der Tabelle*** **customer**

## Sätze mit variabler Länge schätzen

Wenn eine Tabelle eine oder mehrere VARCHAR-Spalten enthält ( oder NVARCHAR-Spalten, wenn NLS aktiviert ist), sind die Sätze unterschiedlich lang. Ausgehend von den realen Daten muß man die typischen Größe der einzelnen VARCHAR-Spalte schätzen.

Wenn **INFORMIX-OnLine Dynamic Server** Sätzen von unterschiedlicher Größe einen Bereich zuweist, so werden Pages dann als voll betrachtet, wenn der Platz für einen weiteren Satz mit *maximaler* Größe nicht ausreicht. Dies kann die Zuweisung der Sätze an Pages begrenzen. Berechnen Sie folgende Variablen:

*maxrow* = die Summe der maximalen Größen aller Spalten der Tabelle, plus 4 Byte. Behandeln Sie TEXT- und BYTE-Spalten so, als hätten Sie eine Größe von 56 Byte.

*typrow* = die Summe der geschätzten typischen Größen aller Spalten einer Tabelle, plus 4 Byte.

*homerow* = `if` (*typrow* ≤ *pageuse*) `then` *typrow*
else 4 + Rest von (*typrow*/*pageuse*)

Weisen Sie *homerow* die Menge eines typischen Satzes zu, die in die Original-Page paßt; wenn eine Page ausreichend ist, ist dies *typrow*.

*overpage* = `if` (*typrow* ≤ *pageuse*) then *0*
else floor (*typrow*/*pageuse*)

Weisen Sie *overpage* die Anzahl der Erweiterungs-Pages zu, die für einen typischen Satz benötigt werden; dies ist Null (0), wenn ein typischer Satz auf eine einzige Page paßt.

*homemax* = if (*maxrow* ≤ *pageuse*) then *maxrow*
else 4 + Rest von (*maxrow*/*pageuse*)

Weisen Sie *homemax* den Reservierungswunsch des Datenbankservers zu.

*datrows* = floor ((*pageuse* – *homemax*)/*typrow*)

Weisen Sie *datrows* die Anzahl der typischen Sätze zu, die in eine Page passen, bevor der Datenbankserver entscheidet, daß die Page voll ist.

*datpages* = ceiling (*estrows*/*datrows*)

*expages* = *estrows* × *overpage*

Die Tabelle erfordert ungefähr *datpages* + *expages* Pages für die Sätze. (Hinweise zur Abschätzung zusätzlicher Pages für TEXT- und BYTE-Spalten werden im Abschnitt "Schätzen der Blobpages" auf Seite 10-20 gegeben.) Eine Schätzung der Tabelle **catalog** wird in Bild 10-7 gezeigt.

| Variable | Schätzung | Grundlage der Schätzung |
|---|---|---|
| *estrows* | 5.000 | Geschäftsplan. |
| *pagesize* | 2.048 | Ausgabe von **tbstat** |
| *pageuse* | 2.020 | *pagesize* - 28 |
| *maxrow* | 381 | Summe aus 4+: |

| Spalte | Datentyp | max Größe | typ. |
|---|---|---|---|
| catalog_num | SERIAL | 4 | 4 |
| stock_num | SMALLINT | 2 | 2 |
| manu_code | CHAR(3) | 3 | 3 |
| cat_descr | TEXT | 56 | 56 |
| cat_picture | BYTE | 56 | 56 |
| cat_adv | VARCHAR(255,65) | 256 | 66 |

| | | |
|---|---|---|
| *typrow* | 191 | |
| *homerow* | 191 | *typrow* ≤ *pageuse* |
| *overpage* | 0 | *typrow* ≤ *pageuse* |
| *homemax* | 381 | *maxrow* ≤ *pageuse* |
| *datrows* | 8 | floor ((*pageuse* - *homemax*)/*typrow*) |
| *datpages* | 625 | ceiling (*estrows*/*datrows*) |
| *expages* | 0 | *estrows* × *overpage* |

*Bild 10-7* ***Schätzung der Größe der Tabelle* catalog**

# Schätzung der Index-Pages

Der Tblspace enthält die Index-Pages ebenso wie Daten-Pages. Die Index-Pages können einen bedeutenden Teil der gesamten Schätzung ausmachen.

Ein Indexeintrag besteht aus einem *Schlüssel* und einem *Zeiger* (*Pointer*). Der Schlüssel ist eine Kopie der indizierten Spalte bzw. der indizierten Spalten eines Datensatzes. Der Zeiger ist ein Wert, der 4 Byte lang ist; er wird verwendet, um den Satz ausfindig zu machen, der den Schlüsselwert enthält.

Ein Unique-Index enthält je einen Eintrag für jeden Satz der Tabelle. Er enthält auch zusätzliche Zeiger-Pages, die eine B+ Baumstruktur erzeugen. (Die Indexstruktur ist im Abschnitt "Zeitbedarf bei einem Netzwerkzugriff" auf Seite 13-21 beschrieben.) Wenn ein Index doppelte Werte zuläßt, enthält er weniger Schlüssel als Zeiger; d. h., daß auf einen Schlüssel eine Liste von Zeigern folgen kann.

Wenn dies alles wäre, was bei Indizes zu beachten ist, dann wäre die Schätzung des erforderlichen Platzes einfach. Eine Index-Leaf-Page verwendet jedoch eine Technik, die *Schlüsselkomprimierung* genannt wird; dies ermöglicht es dem Datenbankserver, in eine Page mehr Schlüssel einzufügen, als normalerweise hineinpassen würden. Der Platz, der auf Grund der Schlüsselkomprimierung gespart wird, verändert sich mit dem Inhalt der Schlüssel, von Page zu Page und von Index zu Index.

Um die Größe eines kompakten Index zu schätzen (unter Nichtbeachtung des Platzes, der durch die Schlüsselkomprimierung gespart wurde), sind die Werte wie folgt anzunehmen:

*keysize* = die gesamte Größe der indizierten Spalte bzw. Spalten.

*pctuniq* = die Anzahl der einmaligen (unique) Einträge, die Sie in diesem Index erwarten, geteilt durch *estrows*.

Verwenden Sie 1,0 bei einem Unique-Index oder bei einem Index, bei dem nur gelegentlich doppelte Werte vorkommen. Wenn doppelte Werte bei wichtigen Zahlen vorkommen, weisen Sie *pctuniq* einen Wert kleiner als 1,0 zu.

*entsize* = (`keysize` + (5 / *pctuniq*)

*pagents* = `floor` (*pageuse*/*entsize*)

Es gibt pro Index-Page ungefähr *pagents* Einträge.

*leaves* = `ceiling` (*estrows*/*pagents*)

Es gibt in einem Index ungefähr *leaves* Leaf-Pages.

*branches* = ceiling (*leaves*/*pagents*)

Es gibt ungefähr *branches* Pages, die keine Leaf-Pages sind.

Der Index enthält ungefähr *leaves* + *branches* Pages, wenn er kompakt ist. Schätzungen über zwei Indizes der Tabelle **customer** sind in Bild 10-7 auf Seite 10-18 dargestellt.

Da Sätze gelöscht und neue Sätze eingefügt werden, kann ein Index kleiner werden. Dies bedeutet, daß die Leaf-Pages eventuell nicht mehr voll sind. Auf der anderen Seite kann ein Index kleiner sein, wenn die Schlüsselkomprimierung effektiv ist. Die hier vorgestellte Methode sollte für eine vorsichtige Schätzung der ersten Indizes genügen. Wenn der Platz eines Index wichtig ist, kann man unter Verwendung realer Daten einen großen Index zum Testen erstellen und die Größe des Index mit dem Dienstprogramm **oncheck** prüfen.

| Variable | Schätzung | Basis für die Schätzung |
|---|---|---|
| *estrows* | 1.500 | Geschäftsplan |
| *pagesize* | 2.048 | Ausgabe von **onstat** |
| *pageuse* | 2.016 | *pagesize* |
| | | |
| *keysize* | 4 | `customer_num` ist 4 Byte lang |
| *pctuniq* | 1,0 | Unique-Index |
| *entsize* | 12 | 4 x 1,0 + 8 |
| *pagents* | 168 | `floor` (2.016/12) |
| *leaves* | 9 | `ceiling` (1.500/168) |
| *branches* | 1 | `ceiling` (9/168) |
| | | |
| *keysize* | 2 | `state` ist CHAR(2) |
| *pctuniq* | 0,033 | 50 Staaten unter 1.500 Sätzen |
| *entsize* | 8,066 | 2 x 0,033 + 8 |
| *pagents* | 249 | `floor` (2.016/8.066) |
| *leaves* | 7 | `ceiling` (1.500/249) |
| *branches* | 1 | `ceiling` (7/168) |

*Bild 10-8* ***Schätzung der Größe zweier Indizes der Tabelle* customer**

# Schätzen der Blobpages

BYTE- und TEXT-Datenelemente werden auf der Platte in separaten Pages gespeichert. Diese sind entweder über die Daten- und Index-Pages im Tblspace verteilt oder sie sind in einem separaten Blobspace (Diese Erläuterung bezieht sich nicht auf die Blob-Daten, die auf optischen Medien gespeichert werden.). Jedes Blob-Datenelement belegt eine ganze Anzahl an Pages. Für jede Blob-Spalte sind die Werte wie folgt anzunehmen:

*typage* = die Anzahl der ganzen Pages, die für die Speicherung eines Elements von typischer Größe erforderlich ist.

*nnpart* = der Anteil der Sätze, bei denen es in dieser Spalte keinen NULL-Wert gibt.

*totpage*= *typage* × *estrows* × *nnpart*.

Die Werte der Spalte belegen ungefähr *totpage* Pages. Die Tabelle **catalog** der Beispiel-Datenbank enthält zwei Blobpages (siehe Bild 10-7 auf Seite 10-18). Ihre geschätzten Größen kann man sich wie folgt vorstellen:

In der Spalte **cat_descr** ist der Text einer Beschreibung meistens zwei Pages lang; dies sind 250 Wörter oder ungefähr 1.500 Zeichen; daraus folgt *typage* = 1. Es gibt für jeden Eintrag eine Beschreibung, so daß *nnpart* = 1,0 ist. Folglich ist *totpage* = 1 × 5.000 × 1,0 = 5.000 Daten-Pages.

Die Spalte **cat_picture** enthält Liniengraphiken in Computer-lesbarer Form. Die Untersuchung einiger dieser Bilder zeigte, daß die Größe stark voneinander abweicht, aber daß eine typische Datei etwa 75.000 Byte enthält. Aus diesem Grund ist *typage* = 38. Die Marketingabteilung schätzt, daß sie ein Bild mit einem Eintrag in vier Pages speichern werden: *nnpart* = 0,25. Folglich ist *totpage* = 38 × 5.000 × 0,25 = 47.500 Datenpages.

Nachdem eine Tabelle erstellt und geladen wurde, kann man die Verwendung der Blobpages mit dem Dienstprogramm **oncheck** und der Option **-ptT** prüfen.

## Blob-Daten anlegen

Beim Erzeugen einer Spalte des Typs BYTE oder TEXT kann man wählen, ob man die Daten der Spalte im Tblspace oder in einem Blobspace anlegt. (Diese Erläuterung gilt nicht für Blob-Daten, die auf optischen Medien gespeichert sind). Im folgenden Beispiel wird ein TEXT-Wert im Tblspace angelegt und ein BYTE-Wert in dem Blobspace **rasters**.

```
CREATE TABLE examptab
    (
    pic_id SERIAL,
    pic_desc TEXT IN TABLE,
    pic_raster BYTE IN rasters
    )
```

Ein TEXT- oder BYTE-Wert wird nie zusammen mit den Sätzen einer Tabelle gespeichert. Mit dem Satz wird nur ein Deskriptor gespeichert, der 56 Byte lang ist. Der Wert selbst belegt jedoch mindestens eine Page.

Wenn Blob-Werte im Tblspace gespeichert werden, werden ihre Pages zwischen den Pages, die Sätze enthalten, verteilt. Dies führt zu einer Vergrößerung der Tabelle. Die Blobpages trennen die Pages, die Sätze enthalten, und verteilen sie auf der Platte. Wenn der Datenbankserver nun nur die Sätze und nicht die Blob-Daten liest, muß sich der Schreib-/Lesekopf weiter bewegen als er dies tun müßte, wenn die Blobpages separat gespeichert wären. Der Datenbankserver sucht bei jeder SELECT-Operation, die keine Blob-Spalten abruft, nur die Pages ab, die die Sätze enthalten; zum Überprüfen der Sätze verwendet er Filterausdrücke.

Ein weiterer Gesichtspunkt ist, daß die Platten-I/O zu und von einem Dbspace gepuffert werden. Pages werden für den Fall, daß sie bald wieder benötigt werden, im Speicher gehalten; wenn Pages geschrieben werden kann ein Programm fortfahren, bevor das Schreiben der aktuellen Daten auf der Platte stattfindet.

Jedoch sind Platten-Ein-/Ausgaben zu und von Blobspaces nicht gepuffert. Blobspace-Pages werden nicht in Puffern gehalten, damit sie nochmals gelesen werden können. Einem Programm ist es damit nicht möglich, fortzufahren, bis die Ausgaben an die Blobspace-Pages vollständig sind. Der Grund dafür ist, daß Blobspace-Ein- und -Ausgaben erfahrungsgemäß sehr umfangreich sind. Wenn diese Pages an normale Puffermechanismen weitergeleitet würden, könnten sie die Puffer in Beschlag nehmen und dabei die Index-Pages und weitere Pages, die für eine gute Performance nützlich sind, aus den Puffern verdrängen.

Unter folgenden Umständen sollte man eine TEXT- oder BYTE-Spalte in einem Blobspace anlegen, um so eine gute Performance zu erzielen:

- Wenn einzelne Datenelemente jeweils größer als ein oder zwei Pages sind; wenn sie im Dbspace gespeichert würden, verringert deren Transfer die Wirkung der Page-Puffer.
- Wenn die Anzahl der Blob-Daten-Pages mehr als halb so groß ist wie die Anzahl der Pages, die Datensätze enthalten; wenn sie im Dbspace gespeichert werden, ist die Tabelle aufgebläht und Abfragen der Tabelle werden verlangsamt.

Für eine Tabelle, die relativ klein und nicht temporär ist, kann man die Wirkung eines zugewiesenen Blobspace folgendermaßen erreichen: Laden Sie die ganze Tabelle mit Sätzen, in denen die Blob-Spalten NULL-Werte enthalten. Erzeugen Sie alle Indizes. Satz-Pages und Index-Pages sind nun aufeinanderfolgend. Aktualisieren Sie alle Sätze mit Blob-Daten. Die Blob-Pages stehen dann im Tblspace nach den Satz- und Index-Pages.

# Indexverwaltung

Ein Index ist für eine Spalte erforderlich (oder eine Zusammensetzung von Spalten), die *eindeutig* sein muß. Wie bereits in Kapitel 13 erläutert, kann ein Index die Abfragen-Optimierung auf verschiedene Arten beschleunigen. Der Optimierer kann einen Index wie folgt verwenden:

- Wiederholtes sequentielles Durchsuchen einer Tabelle durch nicht sequentiellen Zugriff ersetzen
- Das Lesen von Sätzen überhaupt vermeiden, wenn zu verarbeitende Ausdrücke nur indizierte Spalten angeben
- Das Sortieren vermeiden (inklusive der Erstellung einer temporären Tabelle), wenn GROUP BY- und ORDER BY-Klauseln ausgeführt werden

Daraus folgt, daß ein Index auf der richtigen Spalte tausende, zehntausende oder in extremen Fällen sogar Millionen von Plattenzugriffen während einer Abfrage einsparen kann. Mit Indizierung ist jedoch ein gewisser Aufwand verbunden.

## Der Platzbedarf von Indizes

Der erste Nachteil eines Index ist sein Platzbedarf auf der Platte. Eine Methode zu Schätzung wurde im Abschnitt "Schätzung der Index-Pages" auf Seite 10-18 vorgestellt. Grob gesagt enthält ein Index eine Kopie eines jeden eindeutigen Daten-Wertes der indizierten Spalte und zusätzlich einen 4 Byte langen Zeiger für jeden Satz in der Tabelle. Diese Informationen können viele Pages zu dem Platz hinzufügen, der für die Tabelle angefordert ist; es ist möglich, daß man genau soviel Index-Pages wie Daten-Pages erhält.

## Der Zeitbedarf von Indizes

Der zweite Nachteil eines Index besteht in seinem Zeitbedarf bei der Veränderung einer Tabelle. In den folgenden Beschreibungen wird angenommen, daß bei der Suche nach einem Index-Eintrag ungefähr zwei Pages gelesen werden müssen. Dies ist der Fall, wenn der Index aus einer Root-Page, einer Zwischenstufe und Leaf-Pages besteht, und sich die Root-Page bereits in einem Puffer befindet. Der Index einer sehr großen Tabelle verfügt über zwei Zwischenstufen, so daß beim Suchen eines Eintrags ungefähr drei Pages gelesen werden müssen.

Nehmen wir an, daß ein Index verwendet wurde, um den zu ändernden Satz ausfindig zu machen. Die Index-Pages werden in den Page-Puffern gefunden. Die Pages anderer Indizes, die geändert werden müssen, müssen von der Platte gelesen werden.

Unter diesen Annahmen kommt es bei folgenden Änderungen zu zusätzlichem Zeitaufwand für die Index-Verwaltung:

- Wenn ein Satz aus einer Tabelle gelöscht wird, müssen die Einträge aus allen Indizes gelöscht werden.

  Der Eintrag des gelöschten Satzes muß gesucht werden (zwei oder drei Pages) und die Leaf-Page muß zurückgeschrieben werden (eine Page); dies ergibt einen gesamten Zugriff von drei bzw. vier Pages pro Index.

- Wenn ein Satz eingefügt wird, müssen die Einträge in alle Indizes eingefügt werden.

  Es muß der Platz für den Eintrag des eingefügten Satzes gefunden werden (zwei oder drei Pages) und zurückgeschrieben werden (eine Page); dies ergibt einen gesamten Zugriff von drei bzw. vier Pages pro Index.

- Wenn ein Satz aktualisiert wird, müssen die Einträge in jedem Index gesucht werden, der sich auf eine geänderte Spalte bezieht (zwei oder drei Pages). Die Leaf-Page muß zurückgeschrieben werden, um den alten Eintrag zu entfernen (eine Page); anschließend muß die neue Spalte im selben Index gesucht werden (zwei oder drei weitere Pages), und der Satz muß eingefügt werden (eine weitere Page).

Einfügungen und Löschungen ändern die Anzahl der Einträge einer Leaf-Page. Bei fast jeder *pagents* Operation muß zusätzliche Arbeit getan werden, weil eine Leaf-Page entweder voll ist oder geleert wurde. Da *pagents* jedoch gewöhnlich größer als 100 ist, kommt dies bei weniger als 1% vor und kann bei der Schätzung vernachlässigt werden.

Kurz gesagt, wenn ein Satz eingefügt oder gelöscht wird, werden pro Index drei bzw. vier zusätzliche Page-I/O-Operationen durchgeführt. Wenn ein Satz aktualisiert wird, werden für jeden Index, der eine geänderte Spalte betrifft, sechs bis acht Page-I/O-Operationen durchgeführt. Bedenken Sie auch, daß beim Zurücksetzen einer Transaktion dies alles rückgängig gemacht werden muß. Aus diesem Grund kann das Zurücksetzen einer Transaktion sehr lange dauern.

Da für die Änderung eines Satzes selbst nur zwei Page-I/O-Operationen erforderlich sind, ist die Verwaltung des Index der zeitintensivste Teil der Datenänderung. Eine Möglichkeit, um diesen Aufwand zu verringern, wird im Abschnitt "Indizes löschen" auf Seite 10-28 erläutert.

# Die Auswahl der Indizes

Indizes sind bei Spalten erforderlich, die eindeutig sein müssen und die nicht als Primärschlüssel angegeben sind. Zusätzlich sollte man einen Index in drei weiteren Fällen hinzufügen:

- Spalten, die bei Joins verwendet werden und die nicht als Fremdschlüssel festgelegt sind
- Spalten, die häufig in Filterausdrücken verwendet werden
- Spalten, die häufig zum Sortieren und Gruppieren verwendet werden

## Join-Spalten

Wie in Kapitel 13 erläutert, sollte mindestens eine der Spalten, die in einem Join-Ausdruck angegeben ist, einen Index haben. Falls kein Index vorhanden ist, erstellt der Datenbankserver normalerweise vor dem Join einen temporären Index und löscht diesen anschließend wieder. Dies ist fast immer schneller, als einen Join mit wiederholtem sequentiellen Durchsuchen einer Tabelle durchzuführen.

Wenn beide Spalten eines Join-Ausdrucks indiziert sind, verfügt der Optimierer über mehr Möglichkeiten zur Erstellung eines Abfrageplans. Im allgemeinen gilt, daß man jede Spalte indizieren sollte, die in einem Join-Ausdruck häufiger verwendet wird und die kein Primär- oder Fremdschlüssel ist.

## Selektive Filter-Spalten in großen Tabellen

Wenn eine Spalte häufig verwendet wird, um Sätze aus einer großen Tabelle herauszufiltern, sollte man diese Spalte indizieren. Der Optimierer kann den Index für das Heraussuchen der gewünschten Spalten verwenden und somit das sequentielle Durchsuchen der ganzen Tabelle vermeiden. Ein Beispiel könnte eine Tabelle sein, die eine sehr lange Liste von Postleitzahlen enthält. Wenn man entdeckt, daß die Spalte mit den Postleitzahlen häufig für die Auswahl einer Teilmenge der Sätze verwendet wird, sollte man eine Indizierung dieser Spalte in Erwägung ziehen, obwohl sie nicht in Joins verwendet wird.

Diese Strategie erzielt letztendlich jedoch nur dann Zeiteinsparungen, wenn die Selektivität der Spalte hoch ist; d. h. wenn nur ein kleiner Teil der Sätze einen der indizierten Werte enthält. Ein nicht sequentieller Zugriff anhand eines Index benötigt viel mehr Festplatten-I/O-Operationen als ein sequentieller Zugriff; aus diesem Grund könnte der Datenbankserver eine Tabelle

ebenso gut sequentiell lesen, wenn dem Filter-Ausdruck einer Spalte mehr als ein Viertel der Sätze entsprechen. Als Regel gilt, daß die Indizierung einer Filter-Spalte in folgenden Fällen Zeit spart:

- Die Spalte wird in den Filter-Ausdrücken von vielen Abfragen bzw. von langsamen Abfragen verwendet.
- Die Spalte enthält mindestens 100 eindeutige Werte.
- Die meisten Spalten-Werte erscheinen in weniger als 10% der Sätze.

### ORDER-BY- und GROUP-BY-Spalten

Wenn eine große Anzahl von Sätzen sortiert oder gruppiert werden muß, dann muß der Datenbankserver die Sätze sortieren. Hierfür schreibt der Datenbankserver alle Sätze in eine temporäre Tabelle und sortiert diese Tabelle. Falls aber (wie in Kapitel 13 erläutert) die Sortier-Spalten indiziert sind, liest der Optimierer die Sätze anhand des Index in sortierter Reihenfolge. Dies vermeidet eine anschließende Sortierung.

Da die Schlüssel in einem Index in sortierter Reihenfolge vorliegen, stellt der Index wirklich das Ergebnis der Sortierung einer Tabelle dar. Dadurch, daß man die Sortier-Spalte bzw. Spalten indiziert, kann man viele Sortierungen während der Abfragen durch eine einzige Sortierung ersetzen; diese erfolgt bei der Erzeugung des Index.

## Mehrfach vorkommende Schlüssel verlangsamen Index-Veränderungen

Wenn in einem Index mehrfach vorkommende Schlüssel erlaubt sind, dann werden die einzelnen Einträge in Listen zu Gruppen zusammengefaßt. Wenn die Selektivität einer Spalte hoch ist, sind diese Listen im allgemeinen kurz. Wenn es aber nur einige eindeutige Werte gibt, werden die Listen ziemlich lang und sie können tatsächlich mehrere Leaf-Pages überschreiten.

Beispielsweise sind bei einer indizierten Spalte, die nur die Werte V für verheiratet und L für ledig enthält, alle indizierten Einträge in genau zwei Listen für mehrfach vorhandene Werte enthalten. Solch ein Index ist nicht sehr nützlich, aber man kann über ihn Abfragen durchführen; der Datenbankserver kann die Liste der Sätze lesen, die den einen oder den anderen Wert haben.

Wenn ein Eintrag aus der Liste der mehrfach vorhandenen Werte gelöscht wird, dann muß der Datenbankserver die ganze Liste lesen und einen Teil erneut schreiben. Wenn man einen Eintrag hinzufügt, setzt der Datenbankserver den neuen Satz an das Ende der Liste. Wenn die Liste – wie im

Normalfall – kurz ist, ist keine Operation problematisch. Wenn sich eine Liste aber über mehrere Pages erstreckt, muß der Datenbankserver alle Sätze lesen, um das Ende zu finden. Wenn er einen Eintrag löscht, muß er normalerweise die Hälfte der Pages der Liste aktualisieren und erneut schreiben.

Folglich kann eine indizierte Spalte, die eine kleine Anzahl verschiedener Werte enthält, in einer Tabelle, die eine große Anzahl an Sätzen enthält, die Geschwindigkeit einer Aktualisierung drastisch reduzieren. Ein Beispiel hierzu ist eine Spalte, die die Namen oder Abkürzungen der Staaten bzw. Bezirke enthält. Wenn es in einer Mailing-Liste, die 100.000 Sätze umfaßt, 50 eindeutige Werte gibt, dann gibt es im Durchschnitt 2.000 mal den gleichen Wert. Aber reale Daten sind niemals so gut verteilt; in einer solchen Tabelle ist es wahrscheinlicher, daß 10.000 mal oder öfter der gleiche Wert vorkommt, und daß dadurch die Listen eine Länge von 50 Pages erreichen können.

Wenn der Datenbankserver einen Eintrag in einer solchen Liste einfügt oder löscht, ist er für längere Zeit beschäftigt. Nachteilig ist, daß er während der Ausführungszeit alle betroffenen Index-Pages sperren muß; dies verringert die parallelen Zugriffe auf die Tabelle sehr.

Man kann dieses Problem auf ziemlich einfache Weise vermeiden; hierfür braucht man etwas Platz auf der Platte. Man muß wissen, daß der Datenbankserver die führende Spalte eines zusammengesetzten Index genauso wie einen Index einer einzelnen Spalte verwendet. Anstatt einen Index für eine Spalte zu erzeugen, die nur einige eindeutige Werte enthält, sollte man einen zusammengesetzten Index für diese und eine nachfolgende Spalte erzeugen, wobei die Werte der zweiten Spalte gut verteilt sind.

Beispielsweise kann man den Index einer Spalte, die nur die Werte V und L enthält, in einen zusammengesetzten Index umwandeln, der diese Spalte und eine Spalte mit dem Geburtsdatum enthält. Man kann jede beliebige zweite Spalte verwenden, um die Schlüsselwerte zu verteilen, solange sich die Werte nicht verändern oder Änderungen zusammen mit dem wirklichen Schlüssel auftreten. Je kürzer die zweite Spalte ist, desto besser, da deren Werte in den Index kopiert werden und die Größe des Index erhöhen.

## Indizes löschen

Bei einigen Anwendungen kann der Großteil der Aktualisierungen einer Tabelle auf eine bestimmte Zeitperiode begrenzt werden. Alle Aktualisierungen werden vielleicht über Nacht oder an bestimmten Tagen durchgeführt.

Wenn dies der Fall ist, sollte man in Betracht ziehen, die nicht eindeutigen Indizes während der Durchführung von Aktualisierungen zu löschen, und nach der Aktualisierung neue Indizes zu erzeugen. Dies kann zwei positive Auswirkungen haben.

Zum einen kann das Aktualisierungsprogramm schneller ablaufen, da weniger Indizes aktualisiert werden müssen. Häufig ist die gesamte Zeit, die man zum Löschen der Indizes, der Aktualisierung ohne die Indizes und der anschließenden Wiedererzeugung der Indizes braucht, geringer als die Zeit, die man für die Aktualisierung mit den Indizes braucht (Der Zeitbedarf für die Aktualisierung der Indizes ist im Abschnitt "Der Zeitbedarf von Indizes" auf Seite 10-23 erläutert.).

Zum anderen sind erst neu erstellte Indizes die effektivsten. Häufige Aktualisierungen führen gewöhnlich zu einer schlechten Indexstruktur; dies bewirkt, daß der Index viele nur zum Teil gefüllte Leaf-Pages enthält. Dadurch verringert sich die Wirksamkeit eines Index und es wird Platz auf der Platte verschwendet.

Als weitere zeiteinsparende Maßnahme sollte man sicherstellen, daß ein Batch-Aktualisierungs-Programm Sätze in der Reihenfolge aufruft, die durch den Primärschlüssel-Index definiert ist. Diese Reihenfolge bewirkt, daß die Pages des Primärschlüssel-Index der Reihe nach gelesen werden und jede Page nur einmal.

Wenn man die LOAD-Anweisung oder das Dienstprogramm **dbload** verwendet, verlangsamen vorhandene Indizes auch das Laden der Tabellen. Eine Tabelle zu laden, die überhaupt keine Indizes hat, ist ein sehr schneller Vorgang (er dauert etwas länger als eine sequentiellen Kopie von Platte zu Platte), aber die Aktualisierung von Indizes ist ein viel größerer Aufwand.

Die schnellste Möglichkeit, eine Tabelle zu laden, ist folgende:

1. Die Tabelle löschen (falls sie existiert).
2. Die Tabelle erzeugen, ohne dabei irgendwelche Unique-Constraints festzulegen.
3. Alle Sätze in die Tabelle laden.
4. Die Tabelle um Unique-Constraints ergänzen.
5. Indizes erzeugen, die nicht eindeutig sind.

Wenn nicht garantiert ist, daß die geladenen Daten allen Unique-Constraints genügen, muß man vor dem Laden der Sätze Unique-Indizes erzeugen. Es spart Zeit, wenn die Sätze zumindest für einen der Indizes in der richtigen Reihenfolge vorliegen (wenn man die Wahl hat, sollte dies der Index mit dem längsten Schlüssel sein). Dies verringert die Anzahl der Leaf--Pages, die gelesen und geschrieben werden müssen.

## Cluster-Indizes

Der Begriff *Cluster-Index* ist eine unzutreffende Bezeichnung. An diesem Index ist überhaupt nicht besonders; es ist die Tabelle, die geändert wird, so daß die Sätze physikalisch so sortiert sind, daß sie mit der Reihenfolge der Indexeinträge übereinstimmen. (Verwechseln Sie einen Cluster-Index nicht mit einem *optischen Cluster*; dies ist eine Methode für die Speicherung von Blobs auf einem optischen Medium, die logisch miteinander verbunden sind. Informationen über optische Cluster erhalten Sie im Handbuch *INFORMIX-On-Line/Optical Benutzerhandbuc*h).

Wenn man weiß, daß eine Tabelle nach einem bestimmten Index sortiert ist, kann man dieses Wissen benutzen, um das Sortieren zu vermeiden. Wenn eine Tabelle nach dieser Spalte durchsucht wird, kann man sich sicher sein, daß diese Spalte nur einmal in sequentieller Reihenfolge gelesen wird.

Bei der Datenbank **stores6** hat die Tabelle **orders** den Index **zip_ix** für die Spalte **zip_code**. Das folgende Kommando veranlaßt den Datenbankserver, die Sätze der Tabelle **customer** in absteigender Reihenfolge nach den Werten der Spalte **zip_code** zu sortieren:

```
ALTER INDEX zip_ix TO CLUSTER
```

Um eine Tabelle nach einer nicht indizierten Spalte zu sortieren, muß man einen Index erzeugen. Das folgende Kommando sortiert die Tabelle **orders** neu nach den Werten der Spalte **order_date**:

```
CREATE CLUSTERED INDEX o_date_ix ON orders (order_date ASC)
```

Um eine Tabelle neu sortieren zu können, muß der Datenbankserver die Tabelle kopieren. Im vorangegangenen Beispiel liest der Datenbankserver alle Sätze der Tabelle und baut einen Index auf. Anschließend liest er die Indexeinträge der Reihe nach. Für jeden Eintrag liest er den passenden Satz der Tabelle und kopiert ihn in eine neue Tabelle. Die Sätze der neuen Tabelle sind in der gewünschten Reihenfolge. Diese neue Tabelle ersetzt die alte Tabelle.

Bei Änderung einer Tabelle wird die Neusortierung nicht aufrechterhalten. Wenn man neue Sätze einfügt, werden sie physikalisch am Ende der Tabelle gespeichert, ohne Rücksicht auf deren Inhalt. Wenn man Sätze aktualisiert und den Wert der geclusterten Spalte verändert, werden die Sätze trotzdem an ihren ursprünglichen Platz in der Tabelle zurückgeschrieben.

Wenn die Sortierung auf Grund von Aktualisierungen nicht mehr stimmt, kann man sie wiederherstellen. Das folgende Kommando sortiert die Tabelle neu, um die physikalische Reihenfolge wiederherzustellen:

```
ALTER INDEX o_date_ix TO CLUSTER
```

Die Wiederherstellung der physikalischen Reihenfolge geht gewöhnlich schneller als die erstmalige Sortierung; der Grund dafür ist, daß das Lesen einer Tabelle, die sich fast in pysikalischer Reihenfolge befindet, einem sequentiellen Durchsuchen beinahe gleichkommt.

Das Sortieren in physikalische Reihenfolge und die Wiederherstellung der Reihenfolge braucht viel Zeit und Platz. Man kann einige dieser Sortierungen vermeiden, indem man die Tabelle in der gewünschten Reihenfolge aufbaut. Die physikalische Reihenfolge der Sätze entspricht der Reihenfolge der Eingabe; wenn man in die Tabelle von Anfang an sortierte Daten lädt, braucht man keine Sortierung nach der physikalischen Reihenfolge durchführen.

# Denormalisierung

Das Entity-Relationship Modell, also die Methode zur Datenmodellierung, die in Kapitel 8 beschrieben ist, erzeugt Tabellen, die keine redundanten oder berechneten Daten enthalten.

Um besonders hohen Anforderungen an die Performance zu genügen, muß man manchmal das Datenmodell auf eine Art abändern, die der theoretischen Betrachtungsweise des Modells zuwiderläuft. Dieser Abschnitt beschreibt einige Änderungen und den damit verbundenen Aufwand.

## Kürzere Sätze für schnellere Abfragen

Ein allgemeiner Grundsatz ist, daß Tabellen mit kürzeren Sätzen eine bessere Performance erzielen als Tabellen mit längeren Sätzen. Dies liegt daran, daß Festplatten-I/O's in Pages und nicht in Sätzen durchgeführt werden. Je kürzer die Sätze einer Tabelle sind, desto mehr Sätze sind auf einer Page enthalten. Je mehr Sätze auf einer Page sind, desto weniger I/O-Operationen sind

zum sequentiellen Lesen der Tabelle nötig, und desto wahrscheinlicher ist es, daß ein nicht sequentieller Zugriff aus einem Puffer durchgeführt werden kann.

Das Entity-Relationship Modell hat Sie veranlaßt, alle Attribute einer Entity in eine einzelne Tabelle zu schreiben. Bei einigen Entities kann dies jedoch zu Sätzen mit ungünstiger Länge führen. Es gibt einige Möglichkeiten, diese zu kürzen. Sobald die Sätze kürzer werden, sollte sich die Abfrage-Performance verbessern.

# Lange Zeichenketten ausschließen

Zumeist handelt es sich bei den langen Attributen um Zeichenketten. Wenn man diese aus der ganzen Tabelle entfernen kann, werden die Sätze kürzer.

## Die Verwendung von VARCHAR-Zeichenketten

Eine bestehende Datenbank hat möglicherweise CHAR-Spalten, die sinnvollerweise in VARCHAR-Spalten konvertiert werden sollten. VARCHAR-Spalten kürzen den Satz, wenn der durchschnittliche Wert in der CHAR-Spalte um mindestens 2 Byte kürzer ist als die vorhandene feste Größe der Spalte. (Bei aktiviertem NLS können NCHAR-Spalten CHAR-Spalten und NVARCHAR-Spalten VARCHAR-Spalten ersetzen.)

Dieses Ersetzen beeinträchtigt nicht die theoretischen Eigenschaften des Modells. Außerdem sind VARCHAR-Daten sofort mit den meisten vorhandenen Programmen, Masken und Listen kompatibel (Masken müssen erneut kompiliert werden).

## Lange Zeichenketten in TEXT ändern

Wenn eine typische Zeichenkette eine halbe Page oder mehr füllt, sollte man erwägen, diese in eine TEXT-Spalte mit einem separaten Blobspace umzuwandeln. Die Spalte in der Satz-Page ist nur 56-Byte lang; dies sollte viele weitere Sätze in einer Page zulassen. Der Datentyp TEXT ist jedoch nicht automatisch mit vorhandenen Programmen kompatibel. Der Code, um einen TEXT-Wert in ein Programm zu holen, ist viel komplizierter als der Code zum Holen eines CHAR-Wertes.

### Eine Symboltabelle für wiederholte Zeichenketten erstellen

Wenn eine Spalte Zeichenketten enthält, die nicht für jeden Satz eindeutig sind, kann man diese Zeichenketten in eine Tabelle übertragen, in der nur eindeutige Kopien gespeichert werden.

Z. B. enthält die Spalte **customer.city** in der Beispiel-Datenbank die Namen von Städten. Einige Städtenamen werden in der Spalte wiederholt und bei den meisten Sätzen sind in diesem Feld einige Leerzeichen enthalten. Die Verwendung des Datentyps VARCHAR entfernt diese Leerzeichen, aber nicht die mehrfach vorkommenden Werte.

Man kann die Tabelle **cities** folgendermaßen erstellen:

```
CREATE TABLE cities
    (
    city_num SERIAL PRIMARY KEY,
    city_name VARCHAR(40) UNIQUE
    )
```

Anschließend kann man die Definition der Tabelle **customer** ändern, so daß die Spalte **city** ein Fremdschlüssel wird, der sich auf die Spalte **city_num** in der Tabelle **cities** bezieht.

Man muß jedes Programm, das einen neuen Satz in die Tabelle **customer** einfügt, dahingehend ändern, daß es die Stadt des neuen Kunden in die Tabelle **cities** einträgt. Der Datenbankserver kann in SQLCODE einen Wert zurückgeben, der anzeigt, ob die Einfügung auf Grund eines doppelten Schlüssels erfolgreich war oder nicht. Dies ist kein logischer Fehler; er bedeutet nur, daß bereits ein Kunde in dieser Stadt wohnt (Ein **4GL**-Programm muß jedoch das Kommando WHENEVER benutzen, um die Fehler abzufangen; ansonsten beendet der negative Wert in SQLCODE das Programm.).

Neben der Änderung der Programme, die Daten einfügen, muß man auch alle Programme und gespeicherten Abfragen ändern, die den Städtenamen abrufen. Man muß einen Join auf die neue Tabelle **cities** verwenden, um die Daten zu erhalten. Mit der Aufgabe der theoretischen Korrektheit des Modells handelt man sich nachteilig ein, daß Programme, die Sätze einfügen sowie einige Abfragen besonderes kompliziert werden. Bevor man Änderungen durchführt, sollte man sicher sein, daß diese zu vernünftigen Einsparungen in bezug auf Plattenspeicher und Ausführungszeit führen.

### Zeichenketten in eine Begleittabelle übertragen

Zeichenketten, die kürzer als eine halbe Page sind, vergeuden Platz auf der Platte, wenn man diese als TEXT behandelt; man kann diese aber von der Haupttabelle in eine Begleittabelle übertragen. Die Verwendung von Begleittabellen ist das Thema des nächsten Abschnitts.

## Umfangreiche Tabellen aufteilen

Betrachten Sie alle Attribute einer Entitiy, deren Sätze für eine gute Performance zu umfangreich sind. Suchen Sie nach einigen Gemeinsamkeiten oder Prinzipien, die die Aufteilung in zwei Gruppen ermöglichen. Teilen Sie die Tabelle in eine Primär- und in eine Begleittabelle auf und wiederholen Sie in jeder Tabelle den Primärschlüssel. Die kürzeren Sätze ermöglichen es, daß jede Tabelle schneller abgefragt oder aktualisiert werden kann.

### Teilung anhand der Größe

Ein Grundsatz, nach dem man eine Entity-Tabelle teilen kann, ist die Größe: übertragen Sie die Attribute, die normalerweise Zeichenketten sind, in die Begleittabelle. Belassen Sie die numerischen und andere kleine Attribute in der Primärtabelle. Bei der Beispiel-Datenbank kann man die Spalte **ship_instruct** von der Tabelle **orders** abspalten. Die Begleittabelle kann man **orders_ship** nennen. Die Begleittabelle hat zwei Spalten, einen Primärschlüssel, der eine Kopie von **orders.order_num** ist und die Originalspalte **ship_instruct**.

### Teilung aufgrund häufiger Verwendung

Ein anderer Grundsatz für die Teilung einer Entity ist die Häufigkeit der Verwendung. Wenn einige Attribute selten abgefragt werden, kann man sie in eine Begleittabelle übertragen. Bei der Beispiel-Datenbank könnte es sein, daß die Spalten **ship_instruct**, **ship_weight** und **ship_charge** nur in einem einzigen Programm abgefragt werden. In diesem Fall kann man diese in eine Begleittabelle übertragen.

### Teilung auf Grund der Aktualisierungshäufigkeit

Aktualisierungen nehmen mehr Zeit in Anspruch als Abfragen. Da die Aktualisierungsprogramme Index-Pages und Datensätze sperren, werden die Programme langsamer, als die, die nur Abfragen durchführen. Wenn bestimmte Attribute einer Entity häufig aktualisiert werden, während viele andere nur selten oder überhaupt nicht verändert werden – *und* wenn viele Ab-

fragen diese zuletzt genannten auswählen und die häufig aktualisierten nicht brauchen – kann man die häufig geänderten Spalten in eine Begleittabelle übertragen. Die Aktualisierungsprogramme verwenden diese Tabelle, während die Abfrageprogramme die Primärtabelle verwenden.

### Aufwand für Begleittabellen

Das Teilen einer Tabelle verbraucht zusätzlichen Speicherplatz. Für jeden Satz gibt es zwei Kopien des Primärschlüssels, eine Kopie in jedem Satz. Außerdem gibt es auch zwei Primärschlüssel-Indizes. Um die Anzahl der zusätzlichen Pages zu schätzen, kann man die Methoden verwenden, die in den vorherigen Abschnitten beschrieben wurden.

Vorhandene Programme, Listen und Masken, die SELECT * verwenden, muß man verändern, weil weniger Spalten zurückgeliefert werden. Programme, Listen und Masken, die Attribute von beiden Tabellen verwenden, müssen einen Join durchführen, um diese zu verbinden.

Wenn man einen Satz einfügt oder löscht, müssen zwei Tabellen anstatt einer geändert werden. Falls man die Änderung der beiden Tabellen nicht koordiniert (z. B. indem man die Änderungen in einer einzigen Transaktion durchführt), geht die semantische Integrität verloren.

## Große Tabellen teilen

Eine große Tabelle wirft große Verwaltungsprobleme auf. Die Abfrage einer Tabelle, die 400 MByte umfaßt, nimmt natürlich viel Zeit in Anspruch; aber auch die Sicherung oder Wiederherstellung braucht viel Zeit. Das Sortieren wird sehr langsam, weil die Sortierung in drei oder mehr Sortierstufen durchgeführt wird. Eine Sortierung kann sogar unmöglich werden, weil zu viel Speicherplatz erforderlich wäre. Indizes werden weniger effektiv, weil die Indexstruktur drei oder mehr Zwischenstufen enthält.

Ziehen Sie es in Betracht, eine solche Tabelle in Segmente aufzuteilen. Jedes Segment verfügt über dieselbe Spalten-, Index- und Constraint-Anordnung, aber über unterschiedliche Sätze. Die Teilung in Gruppen von Sätzen sollte nach einem Wert eines Attributs erfolgen, der in den meisten Abfragen vorkommt (wahrscheinlich der Primärschlüssel), so daß das Programm oder der Benutzer leicht feststellen kann, zu welcher Untertabelle ein Satz gehört. Wenn z. B. der Primärschlüssel eine Zahl ist, könnten Sätze über bestimmte, durchschnittliche Zahlenwerte auf zehn Untertabellen verteilt werden.

Wenn Sätze ein Zeit- oder Datumsattribut beinhalten, kann es sinnvoll sein, eine Tabelle anhand des Alters in Segmente aufzuteilen; besonders dann, wenn neuere Datensätze viel häufiger verwendet werden als ältere.

Die Vorteile beim Aufteilen einer großen Tabelle sind, daß man die Segmente separat kopieren, sortieren, indizieren, archivieren und wiederherstellen kann. Alle diese Operationen werden einfacher. Solange die meisten Abfragen auf dem Attribut basieren, das für die Segmentierung verwendet wird, sollten die Abfragen viel schneller sein.

Der Nachteil beim Aufteilen einer Tabelle ist, daß Operationen, die sich auf die gesamte Tabelle beziehen, viel komplizierter werden. Wenn eine Abfrage nicht auf eine einzige Untertabelle gerichtet werden kann, muß sie in der Form eines UNION mit vielen Unterabfragen geschrieben werden; dabei bezieht sich jede Unterabfrage auf eine andere Untertabelle. Dies macht die Entwicklung von Listen und Masken viel komplizierter.

# Redundante und abgeleitete Daten

Das Datenmodell, das nach den Methoden von Kapitel 8 entwickelt wurde, enthält keine redundanten Daten (jedes Attribut kommt in genau einer Tabelle vor) und keine abgeleiteten Daten.

Diese Eigenschaften minimieren die Größe des verwendeten Speicherplatzes und machen Aktualisierungen so einfach wie möglich. Sie können jedoch dadurch gezwungen sein, häufig Joins und Mengenfunktionen zu verwenden; dies kann mehr Zeit in Anspruch nehmen, als akzeptabel ist.

Als Alternative hierzu kann man neue Spalten einfügen, die redundante oder abgeleitete Daten enthalten – unter der Voraussetzung, daß man die Risiken kennt.

## Abgeleitete Daten hinzufügen

In der Datenbank **stores6** gibt es in der Tabelle **orders** keine Spalte für den Gesamtpreis eines Auftrags. Der Grund dafür ist, daß diese Informationen aus den Sätzen der Tabelle **items** berechnet werden können. Ein Programm, das das Datum und den Gesamtpreis der Auftragsnummer (**order_num**) 1009 auflisten will, kann dies mit folgender Abfrage erreichen:

```
SELECT order_date, SUM (total_price)
    FROM orders, items
    WHERE orders.order_num = 1009
        AND orders.order_num = items.order_num
    GROUP BY orders.order_num, orders.order_date
```

Obwohl beim Join zur Tabelle **items** nur drei oder vier zusätzliche Pages gelesen werden, kann dies für ein interaktives Programm zu lange dauern. Eine Lösung wäre, der Tabelle **orders** eine Spalte mit dem Gesamtpreis hinzuzufügen.

Der Aufwand für berechnete Daten (also abgeleitete Daten) ergibt sich aus dem Speicherplatz, der Komplexität und der Datenintegrität.

Der für eine Gesamtpreis-Spalte verwendete Speicherplatz ist verschwendet, weil dieselben Informationen zweimal gespeichert werden. Außerdem macht diese Spalte die Sätze der Tabelle länger; auf eine Page passen weniger Sätze und die Abfrage einer Tabelle wird langsamer. Am wichtigsten ist, daß jedes Programm, das die Basis-Attribute ändert, dahingehend geändert werden muß, daß es auch die abgeleitete Spalte aktualisiert. Zwangsläufig kann es vorkommen, daß die abgeleitete Spalte falsch ist, daß sie nicht den richtig berechneten Wert enthält.

Berechnete Daten sind keine zuverlässigen Daten. Beispielsweise kann bei der Eingabe eines Auftrags die Errechnung des Auftragswerts ungenau sein. Während die Sätze in die Tabelle **items** eingefügt werden, passiert es, daß der Gesamtpreis des Auftrags in der Tabelle **orders** nicht gleich der Summe der entsprechenden Sätze in der Tabelle **items** ist. Allgemein gilt, daß man die Genauigkeit berechneter Daten so sorgfältig wie möglich festlegen muß, bevor man diese einer Datenbank hinzufügt. Man sollte die Bedingungen, unter denen eine berechnete Spalte *un*zuverlässig ist, so genau wie möglich festlegen.

### Redundante Daten hinzufügen

Ein korrektes Datenmodell vermeidet Redundanz, indem es ein Attribut nur in der Tabelle der Entity hält, welche es beschreibt. Wenn die Attribut-Daten in unterschiedlichen Zusammenhängen gebraucht werden, stellt man die Verknüpfung anhand von Tabellen-Joins her. Eine Verknüpfung braucht aber Zeit. Falls eine häufig verwendete Verknüpfung die Performance verringert, kann man dies vermeiden, indem man die verknüpften Daten in eine andere Tabelle kopiert.

In der Beispiel-Datenbank enthält die Tabelle **manufact** die Namen der Hersteller und deren Lieferzeiten (In einer wirklichen Datenbank würde die Tabelle viele weitere Attribute eines Lieferanten enthalten, wie die Adresse, den Namen des Verkäufers usw.).

Der Inhalt der Tabelle **manufact** ist in erster Linie eine Ergänzung zur Tabelle **stock**. Angenommen, eine zeitkritische Anwendung ruft häufig die Lieferzeit eines bestimmten Produkts ab, aber keine andere Spalte der Tabelle **manufact**. Für eine solche Abfrage muß der Datenbankserver bei der Durchführung der Suche zwei oder drei Daten-Pages lesen.

Man kann in die Tabelle **stock** eine neue Spalte, **lead_time**, einfügen und diese mit einer Kopie der entsprechenden Sätze der Spalte **lead_time** der Tabelle **manufact** füllen. Dies vermeidet die Suche und beschleunigt die Anwendung.

Genauso wie abgeleitete Daten brauchen auch redundante Daten Speicherplatz und stellen somit ein Integritätsrisiko dar. In diesem Beispiel gibt es nicht nur eine zusätzliche Kopie der Lieferzeit eines jeden Herstellers, sondern viele (Jeder Hersteller kann in der Tabelle **stock** mehrmals vorkommen.). Die Programme, die einen Satz in die Tabelle **manufac** einfügen oder einen Satz aktualisieren, müssen dahingehend geändert werden, daß mehrere Sätze der Tabelle **stock** ebenso aktualisiert werden.

Die Integritätsgefahr besteht darin, daß redundante Kopien der Daten eventuell nicht genau sind. Wenn sich in der Tabelle **manufact** die Lieferzeit ändert, ist die Spalte in der Tabelle **stock** solange nicht aktuell, bis sie ebenfalls abgeändert wird. Wie bei berechneten Daten sollte man sich die Mühe machen, die Bedingungen, unter denen die redundanten Daten falsch sein könnten, festzulegen.

# Parallelbearbeitung maximieren

Einige Datenbanken werden zu einer bestimmten Zeit immer nur von einem Programm verwendet; andere werden von mehreren Programmen gleichzeitig verwendet. Zwei Faktoren machen parallel zugreifende Programme von Natur aus langsamer, als wenn der Zugriff nacheinander ausgeführt würde.

- Mehrere Programme stören sich gegenseitig bei der Verwendung der Puffer und Laufwerke. Die Pages, die für ein Programm gelesen wurden, könnten von der Abfrage des nächsten Programms aus dem Puffer gelöscht werden und müssen eventuell erneut gelesen werden. Der Festplatten-I/O einer Abfrage könnte den Schreib-/Lesekopf an eine andere Stelle versetzen und dadurch den sequentiellen Zugriff eines anderen Programms verlangsamen.
- Programme, die Daten verändern, sperren Pages; dies hält alle anderen Programme auf, die dieselben Daten verwenden.

## Konkurrierende Zugriffe vermindern

Konkurrenz zwischen Programmen, die dieselben Betriebsmittel benutzen, ist unvermeidbar. Man kann damit auf drei Arten umgehen:

1. Die Programme veranlassen, weniger Betriebsmittel zu verwenden; entweder dadurch, daß man sie weniger Arbeit ausführen läßt, oder dadurch, daß man ihre Arbeit effektiver macht.
2. Die Betriebsmittel besser verwenden; z. B. indem man Tabellen einem Dbspace zuweist, um Konkurrenz zu vermindern.
3. Mehr Betriebsmittel zur Verfügung stellen: mehr Speicher, mehr und schnellere Laufwerke, mehr und schnellere Computer.

Der erste Punkt entspricht dem Thema von Kapitel 13; dies handelt davon, wie man Abfragen schneller macht. Manchmal muß man auch in Betracht ziehen, Abfragen weniger ausführen zu lassen; d. h. die Funktionen zu verringern, die für Online-Benutzer zur Verfügung stehen. Man könnte nach Transaktionen suchen, die das Durchsuchen einer gesamten, großen Tabelle erforderlich machen: alle Arten von Aufsummierung, Ermittlungen von Durchschnittswerten und Erstellungen von Verwaltungslisten, besonders dann, wenn diese einen Join erfordern. Ziehen Sie in Betracht, solche Transaktionen aus Programmen zu löschen, die online laufen. Bieten Sie stattdessen neue Werkzeuge an, die die Interaktion erst außerhalb der Spitzenzeiten durchführen und bei denen die Ausgaben erst am nächsten Tag erfolgen.

## Änderungen zu einer bestimmten Zeit durchführen

Um die größtmögliche Wirkung zu erzielen, sollte man die Änderungen zu Zeiten planen, zu denen keine Benutzer interaktiv mit der Datenbank arbeiten. Ein Grund dafür ist, daß konkurrierende Änderungen auch bei allen Indizes durchgeführt werden müssen; d. h., daß sie jeden bisher erwähnten Zeitbedarf haben. Ein anderer Grund ist, daß die Änderungen nicht nur Sätze der Tabelle sperren, sondern auch Index-Pages. Dies erhöht die Anzahl der verwendeten Sperren und die Anzahl der Verzögerungen durch Sperren für alle Benutzer.

Ob Änderungen zu einer bestimmten Zeit möglich sind, hängt völlig von der jeweiligen Anwendung ab.

## Verwendung eines Aktualisierungsjournals

Anstatt Aktualisierungen zu der Zeit durchzuführen, zu der die Daten verfügbar sind, sollte man die Erstellung eines *Aktualisierungsjournals* in Betracht ziehen. Dies ist eine Tabelle, deren Sätze anstehende Aktualisierungen darstellen. Die Sätze enthalten folgende Informationen:

- Datenelemente, die in einer Basis-Tabelle zu ändern sind, wobei der Wert NULL keine Änderung bedeutet
- Protokoll-Informationen, die es ermöglichen, eine fehlerhafte Aktualisierung bis zu der Transaktion zurückzuverfolgen, die den Satz eingefügt hat

Einen Satz in ein Aktualisierungsjournal einzufügen, geht gewöhnlich schneller, als die Aktualisierung in einer bzw. mehreren Basistabellen durchzuführen. Zum einen, weil das Journal in den meisten Fällen für die Aktualisierung über genau einen Index verfügt, während die Basis-Tabellen gewöhnlich mehrere haben. Zum anderen, weil Sperr-Verzögerungen fast nicht vorhanden sind, da beim Einfügen eines Satzes in eine Tabelle, die keine Indizes aufweist, nur eine Sperre auf den eingefügten Satz erfordert. Ein anderes Programm kann einen Satz sogar einfügen, bevor das erste Programm seine Transaktion abgeschlossen hat.

Außerdem werden die Pages der Primärtabellen nicht gesperrt, da keine Aktualisierungen durchgeführt werden. Dies ermöglicht es Abfrageprogrammen, ohne Verzögerung abzulaufen.

Nach den Hauptbelastungszeiten (eventuell nach Geschäftsschluß) ruft man ein Batch-Programm auf, um die Aktualisierungen gültig und verfügbar zu machen. Die Aktualisierungen werden Satz für Satz in der Reihenfolge der Eingabe ausgeführt. Unabhängig von möglichen System-Fehlern sollten Sie unbedingt überprüfen, daß keine Korrektur vergessen oder zweimal ausgeführt wurde.

Eine Möglichkeit, um eine Basis-Tabelle zu aktualisieren, ist, zwei Cursor zu verwenden. Der erste Cursor ist ein permanenter Cursor; er wird verwendet, um die Sätze eines Journals zu durchsuchen. (Siehe "Permanenter Cursor: „Hold Cursor"" auf Seite 7-21.) Für jeden Satz des Journals führt das Programm folgende Schritte durch:

1. Das Kommando BEGIN WORK erteilen.
2. Die Sätze mit einem Update-Cursor aus den zu aktualisierenden Tabellen holen (Es werden nur diese Sätze gesperrt.).
3. Die Aktualisierungsdaten aus dem Journal mit den Daten der Zielsätze abgleichen.

4. Die Aktualisierungen in den Zieltabellen durchführen.
5. Den Journalsatz auf eine Weise aktualisieren, daß er als abgeschlossen markiert ist.
6. Das Kommando COMMIT WORK erteilen (oder das Kommando ROLLBACK WORK, wenn ein Fehler entdeckt wurde).

Man läßt ein anderes Programm zum Löschen und Wiederaufbauen der Journaltabelle nur dann ablaufen, nachdem jeder Satz des Journals für gültig erklärt, angewendet und markiert wurde.

Der offensichtliche Nachteil eines Aktualisierungsjournals ist, daß die Basistabellen nicht die aktuellsten Daten anzeigen. Wenn es notwendig ist, daß die Aktualisierungen sofort sichtbar sind, kann man das Journal nicht verwenden.

Die großen Vorteile von verringerten I/O's und verringerten Verzögerungen während der Hauptauslastungszeit sprechen zugunsten eines Aktualisierungsjournals. Zeitverschobene Aktualisierungen werden in vielen Anwendungen akzeptiert. Beispielsweise gibt keine Bank einen Kontostand genauer an als den Stand, der am Vortag bei Geschäftsschluß vorlag; und zwar aus diesem Grund: die Bank zeichnet die Kontobewegungen während des Tages in einem Journal auf und führt die Aktualisierungen über Nacht durch.

## Aktualisierungen isolieren und auflösen

Falls Aktualisierungen wirklich während der Hauptauslastungszeit interaktiv durchgeführt werden müssen, muß man eine Möglichkeit finden, die Abfragen von den Aktualisierungen zu isolieren.

### Tabellen aufteilen, um stark benutzte Spalten zu isolieren

Dieses Konzepte ist bereits in einem früheren Abschnitt behandelt worden. Wenn es erforderlich ist, Aktualisierungen während der Hauptauslastungszeiten durchzuführen, sollte man die Struktur der Tabellen untersuchen. Kann man die Spalten einteilen in solche, die ständig geändert werden und in statische, die selten aktualisiert werden? Falls dies der Fall ist, sollte man erwägen, die Tabellen so aufzuteilen, daß die häufig verwendeten Spalten in einer Begleittabelle isoliert werden.

Abhängig von den Operationen und von den Prioritäten der Benutzer kann man anschließend entweder die statische oder die häufig verwendete Untertabelle auf das schnellste Laufwerk übertragen.

### Engpäße, die von Tabellen verursacht sind, auflösen

Kleine Tabellen werden manchmal für Aufsummierungen, Arbeitssätze und Protokollierungen verwendet. Beispielsweise kann ein interaktives Programm eine Tabelle pflegen, die für jeden berechtigten Benutzer einen Satz enthält. Jedesmal, wenn ein Benutzer ein Programm aufruft oder beendet, aktualisiert das interaktive Programm den Benutzersatz, um die Start- und Endzeit, die Anzahl der Transaktionen oder um andere Daten zur Arbeitsüberwachung aufzuzeigen.

Kleine Tabellen, die nur gelesen werden, verursachen keine Performanceprobleme; aber kleine Tabellen, die parallel aktualisiert werden, verursachen welche. Eine kleine Tabelle, die bei jeder Transaktion aktualisiert werden muß, kann zu einem Engpaß werden, wobei jedes aktive Programm in eine Warteschlange eingereiht wird und warten muß.

Um das Problem zu lösen, kann man entweder ein Journal verwenden, das Aktualisierungen außerhalb der Hauptauslastungszeit durchführt (wie bereits beschrieben); oder man kann den Engpaß auflösen, indem man mehrere Tabellen erzeugt. Um die Benutzer zu überwachen, kann man eine Tabelle erzeugen, die für jeden Benutzer einen Satz enthält.

## Zusammenfassung

Wenn Tabellen eine mittlere Größe haben und immer nur ein Benutzer auf eine Datenbank zugreift, dann ist eine sorgfältig angewandte relationale Theorie ausreichend, um eine gute Performance zu erzielen.

Wenn sowohl die Anzahl der Tabellen, als auch die Anzahl der Benutzer größer werden und die Performance absinkt, muß man praktikable Lösungen finden.

Zuerst einmal muß man die Werkzeuge, die **INFORMIX-OnLine** bietet, kennen und nutzen können. **INFORMIX-OnLine** erlaubt es, die Tabellen hardwareseitig mit größt möglichem Nutzen anzuordnen. Dann kann man langsam nach und nach damit beginnen, die Struktur des Datenmodells und die Programme zu vervollständigen.

Kapitel

# 11

# Zugriff auf Datenbanken regeln

# Kapitelüberblick

Bei einigen Datenbanken stehen sämtliche Daten allen Benutzern zur Verfügung. Bei anderen Datenbanken ist dies nicht der Fall; einigen Benutzern wird der Zugriff auf einige oder alle Daten verwehrt. Man kann den Zugriff auf Daten auf den folgenden fünf Ebenen einschränken:

- Wenn die Datenbank auf Betriebssystemebene in Dateien gespeichert wird, dann kann man auf Betriebssystemebene den Zugriff auf Dateien verwehren.
- Diese Ebene steht nicht zur Verfügung, wenn die Datenbank mit **INFORMIX-OnLine Dynamic Server** gehalten wird. Dieser Datenbankserver führt seine eigene Festplattenverwaltung durch. Die Möglichkeiten des Betriebssystems können nicht zur Wirkung kommen.
- Man kann die Anweisungen GRANT und REVOKE verwenden, um den Zugriff auf die Datenbank oder bestimmte Tabellen zu erlauben oder zu verwehren.
- Mit der Anweisung CREATE PROCEDURE kann man eine in der Datenbank zu speichernde Prozedur schreiben (und anschließend kompilieren). Mit einer solchen Prozedur kann kontrolliert und überwacht werden, welche Benutzer Datenbank-Tabellen lesen, ändern oder erzeugen dürfen.
- Mit der Anweisung CREATE VIEW kann man die Sicht auf die Daten einschränken bzw. ändern. Die Einschränkung kann vertikal sein und dabei bestimmte Spalten ausschließen oder sie kann horizontal sein und dabei bestimmte Sätze ausschließen; sie kann aber auch beides sein.
- Man kann die Anweisungen GRANT und CREATE VIEW auch miteinander kombinieren und dadurch präzise Kontrolle darüber erhalten, welche Teile einer Tabelle und welche Daten ein Benutzer modifizieren kann.

Diese Themen werden in diesem Kapitel behandelt.

# Den Zugriff auf eine Datenbank kontrollieren

Der normale Mechanismus für Berechtigungen auf eine Datenbank basiert auf den Anweisungen GRANT und REVOKE. Diese werden im Abschnitt "Berechtigungen erteilen" auf Seite 11-6 behandelt. Man kann sich jedoch auch mit den Mitteln des Betriebssystems zusätzliche Möglichkeiten zur Kontrolle des Datenbankzugriffs verschaffen.

## Datenbankdateien absichern

Andere Datenbankserver als **INFORMIX-OnLine Dynamic Server** speichern eine Datenbank in Betriebssystem-Dateien. Typischerweise wird eine Datenbank durch eine Anzahl von Dateien repräsentiert: je eine für eine Tabelle, je eine für die Indizes einer Tabelle und mögliche weitere Dateien. Die Dateien werden in einem Dateiverzeichnis gehalten. Das Dateiverzeichnis repräsentiert die Datenbank als Ganzes.

### Mehrbenutzer-Systeme

Man kann den Zugriff auf eine Datenbank dadurch untersagen, daß man den Zugriff auf das Dateiverzeichnis der Datenbank untersagt. Auf welche Weise Sie dies durchführen können, hängt von Ihrem Betriebssystem und Ihrer Hardware ab. Mehrbenutzer-Systeme haben Software-Werkzeuge wie z. B. Access Control List von VMS oder die Berechtigungen auf Dateiebene von UNIX.

***Hinweis:*** *Bei UNIX wird dem Datenbankverzeichnis der Gruppenname* ***informix*** *zugeordnet. Und auch der Datenbankserver läuft immer unter dem Gruppennamen* ***informix****. Deshalb kann man den Gruppennamen nicht dazu verwenden, um einer bestimmten Gruppe von Benutzern den Zugriff zu verweigern. Man kann lediglich alle Gruppenrechte entziehen (Dateimodus 700), und dadurch jedem anderen außer dem Eigentümer den Zugriff auf das Dateiverzeichnis verwehren.*

Man kann auf diese Weise auch den Zugriff auf einzelne Tabellen verwehren; man kann z. B. bestimmten Benutzern den Zugriff auf diejenigen Dateien verwehren, die solche Tabellen repräsentieren, und ihnen den Zugriff auf den Rest der Dateien ermöglichen. Jedoch sind die Datenbankserver nicht dafür entworfen worden, mit Tricks dieser Art umgehen zu können. Sollte eine nicht autorisierte Person versuchen, in einer dieser Tabellen eine Abfrage auszuführen, dann liefert der Datenbankserver lediglich eine Fehlermeldung darüber, daß er nicht in der Lage ist, die Datei zu finden. Dies könnte die Benutzer irritieren.

### Einbenutzer-Systeme

Typische Einbenutzer-Systeme haben geringe softwaremäßige Kontrollen über Dateizugriffe. Man kann den Zugriff auf eine Datenbank lediglich dadurch unmöglich machen, daß man sie auf eine Platte oder Diskette schreibt, die man aus dem Rechner herausnehmen und unter Verschluß bringen kann.

Keine dieser Techniken greifen, wenn Sie mit dem **INFORMIX-OnLine** Datenbankserver arbeiten. Dieser Server hat selbst die Kontrolle über die Festplatte und umgeht dadurch die Mechanismen für Zugriffe auf Dateien auf Betriebssystemebene.

## Vertrauliche Daten absichern

Unabhängig davon, welche spezifischen Methoden zur Zugriffsverweigerung Ihr Betriebssystem bietet, sollten Sie keinesfalls den Inhalt einer höchst sensiblen Datenbank auf einer Platte belassen, die auf Ihrem Rechner allgemein zugänglich ist. Normale Software-Kontrollen lassen sich umgehen; auch bei Daten, die geschützt sein sollten.

Man kann eine Datenbank auf verschiedene Arten mit unterschiedlichen Schwierigkeitsgraden dem öffentlichen Zugriff verweigern:

- Das physikalische Medium aus dem Rechner ausbauen und forttragen. Ist die Festplatte selbst nicht entfernbar, dann könnte man vielleicht das Laufwerk entfernen.
- Die Datenbank auf ein Band kopieren.
- Die Datenbank-Dateien mit einem Dienstprogramm zur Verschlüsselung kopieren. Nur die verschlüsselte Version aufbewahren.

In den letzten beiden Fällen darf man nach dem Kopieren nicht vergessen, die Original-Datenbankdatei mit einem geeigneten Programm vollständig zu löschen bzw. zu leeren.

Anstatt das gesamte Datenbankverzeichnis zu löschen, kann man auch nur diejenigen Dateien kopieren und löschen, die die individuellen Tabellen repräsentieren. Man darf dabei jedoch nicht die Tatsache übersehen, daß Indexdateien Kopien der Daten der jeweils indizierten Spalte bzw. der Spalten enthalten. Die Indexdateien sind auf die gleiche Weise zu löschen.

# Berechtigungen erteilen

Das Recht, auf eine Datenbank zuzugreifen nennt man ein *Berechtigung*. Beispielsweise bezeichnet man das Recht zur Benutzung einer Datenbank als CONNECT-Berechtigung, wohingegen das Recht zur Neuaufnahme von Datensätzen in eine Tabelle als INSERT-Berechtigung bezeichnet wird. Man kann den Umgang mit einer Datenbank dadurch kontrollieren, daß man anderen Benutzern diese Berechtigungen erteilt oder sie ihnen entzieht.

Berechtigungen für Benutzer sind in zwei Gruppen aufgeteilt: eine Gruppe berührt die Datenbank als Ganzes und die andere Gruppe bezieht sich auf einzelne Tabellen. Außerdem gibt es prozedurbezogene Berechtigungen, die festlegen, wer eine Prozedur ausführen darf.

# Berechtigungen auf Datenbank-Ebene

Die drei Ebenen der Berechtigungen auf Datenbanken geben umfassende Kontrollmöglichkeiten darüber, wer Zugriff auf eine Datenbank bekommt.

## CONNECT-Berechtigung

Die tiefste Ebene der Berechtigungen ist die CONNECT-Berechtigung.Sie gibt einem Benutzer die grundlegenden Möglichkeiten, in einer Tabelle Abfragen auszuführen und Daten zu modifizieren. Benutzer mit der CONNECT-Berechtigung können die folgenden Funktionen ausführen:

- Die Anweisungen SELECT, INSERT, UPDATE und DELETE ausführen( vorausgesetzt, sie haben die erforderlichenBerechtigungen auf Tabellenebene).
- Eine in der Datenbank gespeicherte Prozedur ausführen( vorausgesetzt, sie haben die erforderlichen Berechtigungen auf Tabellenebene).
- Views erzeugen( vorausgesetzt, Sie haben die Erlaubnis zur Abfrage der Tabellen, auf denen die Views basieren).
- Temporäre Tabellen und Indizes für diese Tabellen erzeugen.

Das CONNECT-Recht ist erforderlich, damit Benutzer überhaupt auf eine Datenbank zugreifen können. Wenn es sich nicht um eine Datenbank mit besonders sensiblen oder privaten Daten handelt, dann werden Sie unmittelbar nach Erzeugung der Datenbank dem Pseudo-Benutzer **public** mit der Anweisung GRANT das CONNECT-Recht erteilen (GRANT CONNECT TO PUBLIC).

Wenn Sie **public** das CONNECT-Recht nicht erteilen, dann sind die einzigen Benutzer, die auf die Datenbank zugreifen können, diejenigen, die explizit das CONNECT-Recht erhalten haben. Sollen nur bestimmte Benutzer Zugriff bekommen, so liegt es in Ihrer Hand, diesen den Zugriff zu ermöglichen und allen anderen zu verweigern.

### Einzelne Benutzer und Public

Berechtigungen erteilen Sie entweder namentlich einzelnen Benutzern oder über den Namen **public** allen Benutzern. Jede Zuweisung an **public** hat die Bedeutung eines Standardrechts.

Vor Ausführung einer Anweisung stellt der Datenbankserver sicher, ob der Benutzer die erforderlichen Berechtigungen hat (Diese Information befindet sich in den Systemtabellen; siehe hierzu "Berechtigungen in den Systemtabellen" auf Seite 11-10.).

Der Datenbankserver ermittelt zunächst die Berechtigungen, die dem jeweiligen Benutzer zugeordnet sind; er wertet diese Informationen aus und stoppt. Gibt es für den Benutzer keine Zuweisung, so sucht er nach Berechtigungen für **public**. Findet er welche, so wertet er sie aus.

Auf diese Weise können Sie durch Zuweisung von Rechten an **public** eine Minimal-Ebene an Berechtigungen für alle Benutzer angeben. In besonderen Fällen können Sie diese Berechtigungen dadurch hochstufen, daß Sie den einzelnen Benutzern höhere Berechtigungen erteilen.

## RESOURCE-Berechtigung

Die RESOURCE-Berechtigung beinhaltet die gleichen Rechte wie die CONNECT-Berechtigung. Zusätzlich können Benutzer mit der RESOURCE-Berechtigung neue, permanente Tabellen, Indizes und in der Datenbank zu speichernde Prozeduren erzeugen.

## Berechtigung des Datenbank-Verwalters

Die höchste Ebene der Berechtigungen auf eine Datenbank ist die Berechtigung des *Datenbank-Verwalters*, kurz DBA. Wenn Sie eine Datenbank erzeugen, dann sind Sie automatisch der Verwalter.

Mit DBA-Berechtigung können Sie die folgenden Funktionen ausführen:

- Die Kommandos DROP DATABASE, START DATABASE und ROLLFORWARD DATABASE ausführen.
- NEXT SIZE (aber keine anderen Attribute) in den Systemtabellen ändern, Sätze bei beliebigen Systemtabellen mit Ausnahme von **systables** neu aufnehmen, löschen oder korrigieren.

***Warnung:*** *Obwohl Benutzer mit der DBA-Berechtigung Systemtabellen ändern dürfen, wird es dringend empfohlen, keine Sätze in den Systemtabellen zu korrigieren, zu löschen oder zu ändern. Änderungen an den Systemtabellen kann die Integrität der Datenbank zerstören.*

- unabhängig von den jeweiligen Eigentumsrechten Objekte löschen oder ändern.
- Tabellen, Views und Indizes erzeugen, die anderen Benutzern gehören.
- Datenbank-Berechtigungen anderen Benutzern erteilen, einschließlich der DBA-Berechtigung.

## Rechte des Eigentümers

Die Datenbank und jede Tabelle, jede View, jeder Index, jede Prozedur und jedes Synonym hat einen Eigentümer. Der Eigentümer eines Objekts ist normalerweise die Person, die es erzeugt hat. Allerdings kann ein Benutzer mit der DBA-Berechtigung auch Objekte unter dem Namen anderer Benutzer erzeugen.

Der Benutzer eines Objekts hat alle Rechte an dem Objekt, er kann es löschen oder ändern, ohne weitere Berechtigungen zu benötigen.

# Berechtigungen auf Tabellenebene

Sie können den einzelnen Tabellen sieben unterschiedliche Berechtigungen erteilen, und so den Nicht-Eigentümern die Rechte des Eigentümers zuweisen. Vier hiervon – die Berechtigungen für SELECT, INSERT, DELETE und UPDATE – kontrollieren den Zugriff auf die Daten einer Tabelle. Die Berechtigung INDEX kontrolliert die Erzeugung eines Index. Die Berechtigung ALTER kontrolliert das Recht zur Änderung einer Tabellendefinition. Die Berechtigung REFERENCES kontrolliert das Recht, einer Tabelle einen Referenz-Constraint zuweisen zu dürfen.

Bei einer ANSI-kompatiblen Datenbank hat nur der Eigentümer einer Tabelle Berechtigungen. Bei anderen Datenbanken erteilt der Datenbankserver – als eine der bei der Erzeugung beteiligten Komponenten – dem Pseudo-Benutzer **public** bei einer Tabelle alle Berechtigungen, mit Ausnahme von ALTER und REFERENCES. Damit kann jeder beliebige Benutzer mit der CONNECT-Berechtigung auf eine neu erzeugte Tabelle zugreifen. Wenn dies nicht erwünscht ist, müssen Sie **public** nach Erzeugung der Tabelle alle Berechtigungen für diese Tabelle entziehen.

## Die einzelnen Berechtigungen

Vier Berechtigungen legen fest, wie Benutzer auf eine Tabelle zugreifen dürfen. Als Eigentümer oder Eigentümerin einer Tabelle können Sie diese Berechtigungen unabhängig voneinander erteilen oder entziehen:

- Die SELECT-Berechtigung erlaubt die Abfrage von Tabellen, auch von temporären Tabellen.
- Die INSERT-Berechtigung erlaubt, neue Sätze aufzunehmen
- Die UPDATE-Berechtigung erlaubt, Sätze zu überarbeiten.
- Die DELETE-Berechtigung erlaubt, Sätze zu löschen.

Um eine Tabelle abzufragen, benötigt der Benutzer die SELECT-Berechtigung. Die SELECT-Berechtigung ist jedoch keine Voraussetzung für andere Berechtigungen. Ein Benutzer kann die INSERT- oder UPDATE-Berechtigung haben, ohne die SELECT-Berechtigung haben zu müssen.

Beispielsweise könnte Ihre Anwendung über eine Hilfs-Tabelle verfügen. Jedesmal beim Start eines bestimmten Programms würde ein Satz in die Verwendungs-Tabelle eingefügt werden, womit die Verwendung des Programms dokumentiert würde. Vor Programmende könnte der Satz korrigiert werden und damit anzeigen, wie lange das Programm gelaufen ist, und wieviele Aktivitäten vom Benutzer durchgeführt wurden.

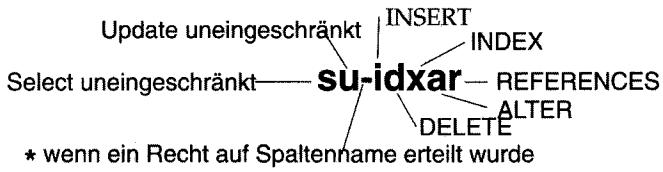
Wenn Sie wünschen, daß beliebige Benutzer des Programms bei dieser Verwendungs-Tabelle Sätze aufnehmen und korrigieren können sollen, so müssten Sie **public** die INSERT- und UPDATE-Rechte erteilen. .

### Berechtigungen in den Systemtabellen

Berechtigungen werden in den Systemtabellen aufgezeichnet. Jeder Benutzer mit der CONNECT-Berechtigung kann die Systemtabellen abfragen.

Berechtigungen auf Datenbank-Ebene werden in der Systemtabelle **sysusers** aufgezeichnet. Die Spalte user-id ist hier der Primärschlüssel und die einzige weitere Spalte enthält zur Identfizierung des Grades einer Berechtigung ein einzelnes Zeichen: `C`, `R` oder `D`. Wird bei der Erteilung das Schlüsselwort PUBLIC angegeben, so ist damit der Benutzer **public** (kleingeschrieben) gemeint.

Berechtigungen auf Tabellenebene werden in der Tabelle **systabauth** aufgezeichnet. Hier gibt es einen zusammengesetzten Primärschlüssel über die Tabellennummer, den Erteiler und den Empfänger (grantor und grantee) einer Berechtigung. In der Spalte **tabauth** sind die Berechtigungen in einem Code aus sechs Buchstaben wie folgt aufgelistet:




Ein Bindestrich stetht für ein nicht-erteiltes Recht; die Zuteilung aller Berechtigungen wird also wie folgt aussehen: `su-idxar`, wohingegen `-u------` anzeigt, daß nur das UPDATE-Recht erteilt wurde. Die Code-Buchstaben sind normalerweise klein geschrieben. Sie sind jedoch dann groß geschrieben, wenn bei der GRANT-Anweisung die Schlüsselwörter WITH GRANT OPTION verwendet wurden.

Ein Stern an dritter Position bedeutet, daß es für diese Tabelle und diesen Benutzer eigene Rechte auf Spaltenebene gibt. Das jeweilige Berechtigung wird in der Systemtabelle **syscolauth** aufgezeichnet. Der Primärschlüssel hierfür setzt sich zusammen aus der Tabellennummer, dem Erteiler sowie dem Empfänger (grantor und grantee) einer Berechtigung und der Spaltennummer. Das einzige Attribut besteht aus einer Liste von drei Buchstaben, die den Typ der Berechtigung anzeigt: `s`, `u` oder `r`.

### INDEX-, ALTER- und REFERENCES-Berechtigungen

Die INDEX-Berechtigung erlaubt es dem Eigentümer, einen Index für eine Tabelle zu erzeugen und zu ändern. Wie die Berechtigungen SELECT, INSERT, UPDATE und DELETE wird auch die Berechtigung INDEX beim Erzeugen einer Tabelle automatisch **public** zugewiesen.

Sie können die INDEX-Berechtigung jedem beliebigen Benutzer zuweisen. Der Benutzer muß aber auch über die Berechtigung RESOURCE verfügen, um arbeiten zu können. Obwohl die INDEX-Berechtigung automatisch zugewiesen wird (nicht bei ANSI-kompatiblen Datenbanken), können Benutzer, die nur über die CONNECT-Berechtigung auf Datenbankebene verfügen, ihre INDEX-Berechtigung nicht umsetzen. Dies ist sinnvoll, weil ein Index sehr viel Speicherplatz beanspruchen kann.

Die ALTER-Berechtigung erlaubt seinem Eigentümer, die Ausführung der ALTER TABLE-Anweisung auf eine Tabelle. Damit kann er Spalten hinzufügen oder löschen, die Startnummer einer SERIAL-Spalte setzen usw. Sie sollten die ALTER-Berechtigung nur Benutzern zuteilen, die die Datenbank gut kennen.

Die REFERENCES-Berechtigung erlaubt Ihnen, einer Tabelle einen Referenz-Constraint zuzuordnen. Wie bei ALTER sollten Sie die REFERENCES-Berechtigung nur Benutzern zuordnen, die die Datenbank sehr gut kennen.

## Berechtigungen auf Spaltenebene

Sie können die SELECT, UPDATE und REFERENCES-Berechtigungen differenzieren: Sie können einem Benutzer erlauben, nur bestimmte Spalten korrigieren zu dürfen oder Sie können einem Benutzer erlauben, Referenz-Constraints nur bestimmten Spalten zuweisen zu dürfen.

Verwendet man **INFORMIX-OnLine**, so wird hierdurch ein weiter oben aufgeführtes Problem gelöst: daß nur bestimmte Benutzer Gehalt, Leistungsbeurteilung und andere sensible Daten eines Angestellten sehen können sollten. Um das Beispiel auf den Punkt zu bringen: stellen Sie sich eine Tabelle mit Daten von Angestellten wie in Bild 11-1 vor.

```
CREATE TABLE hr_data
    (
    emp_key INTEGER,
    emp_name CHAR(40),
    hire_date DATE,
    dept_num SMALLINT,
    user-id CHAR(18),
    salary DECIMAL(8,2)
    performance_level CHAR(1),
    performance_notes TEXT
    )
```

*Bild 11-1* ***Eine Tabelle mit vertraulichen Daten über Angestellte***

Da die Tabelle vertrauliche Daten enthält, werden Sie unmittelbar nach dem Erzeugen die folgende Anweisung ausführen:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

Für ausgewählte Mitarbeiter der Personalabteilung und für alle Führungskräfte werden Sie eine Anweisung wie die folgende ausführen:

```
GRANT SELECT ON hr_data TO harold_r
```

Auf diese Weise erlauben Sie bestimmten Personen, sich alle Spalten anzuschauen. Für hochrangige Manager, die Leistungsbeurteilungen aussprechen dürfen, könnten Sie eine Anweisung wie die folgende ausführen:

```
GRANT UPDATE (performance_level, performance_notes)
    ON hr_data TO wallace_s, margot_t
```

Diese Anweisung erlaubt den Managern, Beurteilungen ihrer Angestellten einzutragen. Nur für Manager der Personalabteilung bzw. für Personen, denen die Änderungen der Gehaltsebene anvertraut sind, werden Sie eine Anweisung wie die folgende ausführen:

```
GRANT UPDATE (salary) ON hr_data to willard_b
```

Für untere Angestellte einer Personalabteilung könnten Sie eine Anweisung wie die folgende ausführen:

```
GRANT UPDATE (emp_key, emp_name, hire_date, dept_num)
    ON hr_data TO marvin_t
```

Diese Anweisung gibt bestimmten Benutzern das Recht zur Bearbeitung nicht-sensibler Spalten, aber sie verweigert ihnen die Berechtigung zur Änderung der Leistungsbeurteilungen und der Gehälter. Für Personen der Organisationsabteilung, die die Benutzerkennungen zuweisen, ist eine Anweisung wie die folgende sinnvoll:

```
GRANT UPDATE (user_id) ON hr_data TO eudora_b
```

Im Interesse aller Benutzer, die Zugang zur Datenbank haben, aber nicht berechtigt sind, Gehälter oder Leistungsbeurteilungen einzusehen, führen Sie eine Anweisung wie die folgende aus. Die Anweisung erlaubt die Einsicht in nicht-sensible Daten:

```
GRANT SELECT (emp_key, emp_name, hire_date, dept_num, user_id)
    ON hr_data TO george_b, john_s
```

Diese Benutzer können Abfragen wie die folgende ausführen:

```
SELECT COUNT(*) FROM hr_data WHERE dept_num IN (32,33,34)
```

Jedoch liefert der Versuch einer Abfrage wie die folgende eine Fehlermeldung, aber keine Daten:

```
SELECT performance_level FROM hr_data
    WHERE emp_name LIKE "*Smythe"
```

## Berechtigungen auf Prozedurebene

Sie können die Berechtigungen auf eine Prozedur so gestalten, daß auch ein Nicht-Eigentümer das Recht zur Ausführung der Prozedur hat. Wenn Sie eine Prozedur in einer nicht ANSI-kompatiblen Datenbank erzeugen, dann erhält **public** standardmäßig die Berechtigung. Es ist also nicht erforderlich, bestimmten Benutzern das Ausführungsrecht zu erteilen, es sei denn, sie haben sie ihnen vorher entzogen. Wenn Sie eine Prozedur bei einer ANSI-kompatiblen Datenbank erzeugen, dann hat standardmäßig kein weiterer Benutzer das Ausführrecht; Sie müssen es einzelnen Benutzern explizit erteilen. Im folgenden Beispiel werden dem Benutzer **orion** die Berechtigung zugewiesen, die gespeicherte Prozedur mit dem Namen **read-address** auszuführen:

```
GRANT EXECUTE ON read_address TO orion;
```

Die Berechtigungen auf eine Prozedur werden in der Systemtabelle **sysprocauth** gespeichert. Bei der Tabelle **sysprocauth** gibt es einen Primärschlüssel über die Prozedurnummer, Erteiler und Empfänger (grantor und grantee) einer Berechtigung. In der Spalte **procauth** wird das Recht zur Ausführung mit einem kleingeschriebenen Buchstaben "e" angezeigt. Wenn das Recht zur Ausführung mit der WITH GRANT-Option erteilt wurde, so wird das Recht zur Ausführung durch einen großgeschriebenen Buchstaben "E" angezeigt.

## Berechtigungen automatisch erteilen

Manchmal erzwingt das Design einer frisch eingerichteten Datenbank eine Vielzahl von GRANT-Anweisungen auszuführen. Sobald es Veränderungen bei den Aufgaben der Angestellten gibt, sind auch Veränderungen bei den Berechtigungen erforderlich. Wenn beispielsweise das Arbeitsverhältnis eines Angestellten der Personalabteilung beendet wird, sollten Sie ihm so schnell wie möglich die UPDATE-Berechtigung entziehen. Anderenfalls riskieren Sie, daß der unglückliche Angestellte eine Anweisung wie die folgende ausführt:

```
UPDATE hr_data
    SET (emp_name, hire_date, dept_num) = (NULL, NULL, 0)
```

Bei sensiblen Daten kann es nötig sein, daß die Berechtigungen täglich oder sogar stündlich geändert werden. Wenn diese Notwendigkeit vorhanden ist, sollten Sie sich durch automatisch ablaufende Prozeduren die Arbeit erleichtern.

Als erstes sollten Sie die Berechtigungen in Klassen einteilen, und zwar auf Basis der Benutzertätigkeiten und nicht auf Basis der Tabellenstruktur. Beispielsweise benötigt ein hochrangiger Manager die folgenden Berechtigungen:

- Das Recht zur Auswahl (select) und begrenzten Korrektur (update) bei der hypothetischen Tabelle **hr_data**
- Die Berechtigung (connect) bei dieser und anderen Datenbanken
- Einige abgestufte Berechtigungen auf einige Tabellen dieser Datenbanken

Wenn ein Manager in eine Führungsposition aufsteigt oder in eine Außenstelle versetzt wurde, müssen Sie all diese Berechtigungen entziehen und neue einrichten.

Definieren Sie die erforderlichen Klassen an Berechtigungen und ermitteln Sie die Datenbanken, Tabellen und Spalten, auf die der Zugriff erforderlich ist. Dann teilen Sie jeder dieser Klassen zwei automatisch ablaufende Prozeduren zu: eine zur Erteilung und eine zur Entziehung der Berechtigungen in der Klasse.

## Automatisierung mit INFORMIX-4GL

Welchen Mechanismus Sie verwenden, hängt von Ihrem Betriebssystem und anderen Software-Tools ab. Wenn Sie Programmierer sind, dann ist das für Sie flexibelste Werkzeug sicherlich **INFORMIX-4GL**. Wie folgendes Programmfragment zeigt, ist es bei **4GL** nicht schwer, ein einfaches Programm für einen Benutzerdialog zu schreiben:

```
DEFINE mgr_id char(20)
PROMPT 'What is the user-id of the new manager? ' FOR mgr_id
CALL table_grant ('SELECT', 'hr_data', mgr_id)
```

Obwohl **INFORMIX-4GL** erlaubt, GRANT- und REVOKE-Anweisungen frei mit anderen Programmanweisungen zu mischen, ist es Ihnen leider nicht erlaubt, Parameter über Programmvariablen zu versorgen. Um eine GRANT-Anweisung ausführen zu können, bei der die Benutzerkennung vom Benutzer eingegeben wurde, muß das Programm eine Zeichenkette mit der Anweisung zusammenstellen, diese mit der PREPARE-Anweisung aufbereiten und mit der EXECUTE-Anweisung ausführen. (Diese Anweisungen werden in Kapitel 5 im einzelnen behandelt; dort wird auch das folgende Beispiel eingehend analysiert.)

Bild 11-2 zeigt die Definition einer **4GL**-Funktion namens **table_grant()**, die durch die CALL-Anweisung im vorausgegangenen Beispiel aufgerufen wird.

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
    DEFINE  priv_to_grant char(100),{may include column-list}
            table_name CHAR(20),
            user_id CHAR(20),
            grant_stmt CHAR(200)
    LET grant_stmt =' GRANT ', priv_to_grant,
                    ' ON ', table_name,
                    ' TO ', user_id
    WHENEVER ERROR CONTINUE
    PREPARE the_grant FROM grant_stmt
    IF status = 0 THEN
        EXECUTE the_grant
    END IF
    IF status <> 0 THEN
        DISPLAY 'Sorry, got error #', status, 'attempting:'
        DISPLAY '      ', grant_stmt
    END IF
    WHENEVER ERROR STOP
END FUNCTION
```

*Bild 11-2* ***Eine 4GL-Funktion, die eine GRANT-Anweisung zusammenstellt, aufbereitet und ausführt***

### Automatisierung mit einer Anweisungs-Prozedur

Ihr Betriebssystem unterstützt sicherlich die automatische Ausführung von Anweisungs-Prozeduren. Bei den meisten Betriebssystemen wird bei den interaktiven SQL-Werkzeugen, **DB-Access** und **INFORMIX-SQL**, die Ausführung von Kommandos und SQL-Anweisungen auf Betriebssystemebene unterstützt. Sie können diese beiden Möglichkeiten zur automatisierten Erteilung von Berechtigungen miteinander kombinieren.

Die Einzelheiten hängen von Ihrem Betriebssystem und der jeweils verwendeten Version von **DB-Access** bzw. **INFORMIX-SQL** ab. Im wesentlichen müßten Sie eine Kommandoprozedur erzeugen, die die folgenden Funktionen ausführt:

- Übernahme eines Parameters für die Benutzerkennung des Benutzers, dessen Berechtigungen geändert werden sollen
- Aufbereitung einer Datei mit GRANT- oder REVOKE-Anweisungen unter Verwendung der Benutzerkennung
- Aufruf von **DB-Access** bzw. **INFORMIX-SQL** mit Parametern, über die die Datenbank ausgewählt und die aufbereitete Datei mit GRANT- oder REVOKE-Anweisungen ausgeführt werden kann

Auf diese Weise kann man die Veränderung der Berechtigungsklasse auf ein oder zwei Kommandos beschränken.

## Verwendung von gespeicherten Prozeduren

Eine in der Datenbank gespeicherte Prozedur ist ein Programm, das mit Informix Stored Procedure Language (SPL) geschrieben wurde. Ist eine solche Prozedur einmal erzeugt, dann wird sie im ausführbaren Format in der Datenbank gespeichert. Da sie wie eine Tabelle ein in der Datenbank gespeichertes Objekt ist, kann jeder mit den geeigneten Berechtigungen diese Prozedur ausführen.

Sie können eine in der Datenbank gespeicherte Prozedur dazu verwenden, den Zugriff auf einzelne Tabellen und Spalten einer Datenbank zu kontrollieren. Über eine gespeicherte Prozedur können Sie verschiedene Grade der Zugriffskontrolle realisieren. Eine machtvolle Eigenschaft von SPL besteht in der Möglichkeit, eine in der Datenbank gespeicherte Prozedur mit der DBA-Berechtigung zu belegen. Über eine gespeicherte Prozedur mit der DBA-Berechtigung können auch Benutzer, die die Prozedur ausführen, die DBA-Berechtigung erhalten, die sonst geringe oder keine Berechtigungen auf eine Tabelle

haben. Über eine gespeicherte Prozedur können Benutzer mit ihrer temporären DBA-Berechtigung spezielle Aufgaben ausführen. Die DBA-Berechtigung erlaubt Ihnen die Ausführung der folgenden Aufgaben:

- Sie können bestimmen, welche Daten der Benutzer lesen darf.
- Sie können alle an einer Datenbank durchführbaren Änderungen einschränken und sicherstellen, daß Tabellen nicht insgesamt gelöscht oder versehentlich geändert werden.
- Sie können Änderungen überwachen, wie z. B. Löschen oder Einfügen .
- Sie können im Rahmen einer gespeicherten Prozedur die Erzeugung aller Objekte (Datendefinition) einschränken, so daß Sie die vollständige Kontrolle über das Erstellen von Tabellen, Indizes und Views haben.

## Das Lesen von Daten einschränken

Die in Bild 11-3 gezeigte Prozedur verbirgt zwar die SQL-Syntax vor den Benutzern, erfordert aber, daß die Benutzer über die SELECT-Berechtigung für die Tabelle **customer** verfügen. Wenn Sie einschränken wollen, was der Benutzer mit der SELECT-Anweisung auswählen kann, dann müssen Sie Ihre Prozedur unter Berücksichtigung der folgenden Umstände schreiben:

- Sie sind der Datenbank-Administrator (DBA) der Datenbank.
- Die Benutzer haben für die Datenbank die CONNECT-Berechtigung. Die SELECT-Berechtigung auf die Tabelle benötigen sie nicht.
- Ihre Prozedur (bzw. Abfolge von Prozeduren) wurde unter Verwendung des Schlüsselwortes DBA erzeugt.
- Ihre Prozedur (bzw. Abfolge von Prozeduren) liest für die Benutzer aus der Tabelle.

Wenn Sie möchten, daß ein Benutzer nur Namen, Adresse und Telefonnummer eines Kunden lesen kann, dann müssen Sie die Prozedur abändern (siehe Bild 11-3 wie in Bild 11-3 gezeigt) .

```
CREATE DBA PROCEDURE read_customer(cnum INT)
RETURNING CHAR(15), CHAR(15), CHAR(18);

DEFINE p_lname,p_fname CHAR(15);
DEFINE p_phone CHAR(18);

SELECT fname, lname, phone
    INTO p_fname, p_lname, p_phone
    FROM customer
    WHERE customer_num = cnum;

RETURN p_fname, p_lname, p_phone;

END PROCEDURE;
```

*Bild 11-3* ***Gespeicherte Prozedur zur Einschränkung der Lesevorgänge bei Kundendaten***

# Das Ändern von Daten einschränken

Durch gespeicherte Prozeduren können Sie die Änderungen an einer Tabelle einschränken. Einfach dadurch, daß Sie Änderungen durch eine Prozedur kanalisieren. Die Änderungen führt die Prozedur durch, nicht aber die Benutzer direkt. Wenn Sie die Benutzer dahingehend einschränken wollen, daß sie nur je einen Satz löschen können, um dadurch die versehentliche Löschung aller Sätze einer Tabelle zu verhindern, dann ordnen Sie der Datenbank die Berechtigungen wie folgt zu:

- Sie sind der DBA der Datenbank.
- Alle Benutzer verfügen für die Datenbank über die CONNECT-Berechtigung. Sie können – müssen aber nicht – über die RESOURCE-Berechtigung verfügen. Die DELETE-Berechtigung für die zu schützende Tabelle müssen sie (in diesem Beispiel) nicht haben.
- Ihre gespeicherte Prozedur wurde unter Verwendung des Schlüsselwortes DBA erzeugt.
- Ihre gespeicherte Prozedur führt das Löschen durch.

Schreiben Sie eine Prozedur, ähnlich der in Bild 11-4 gezeigten, die Sätze der Tabelle **customer** löscht. Hierbei kommt eine WHERE-Klausel zur Anwendung, bei der eine vom Benutzer eingegebene **customer_num** (Kundennummer) ausgewertet wird.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DELETE FROM customer
    WHERE customer_num = cnum;

END PROCEDURE;
```

*Bild 11-4* ***Gespeicherte Prozedur zum Löschen von Sätzen***

## Datenänderungen überwachen

Mit gespeicherten Prozeduren können Sie die Änderungen an einer Datenbank aufzeichnen. Sie können Änderungen durch einen bestimmten Benutzer oder jede einzelne Änderung aufzeichnen lassen.

Sie können Änderungen eines einzelnen Benutzers überwachen, indem Sie alle Änderungen über Prozeduren kanalisieren, die die Änderungen von einzelnen Benutzern aufzeichnen. Wenn Sie alle Änderungen an der Datenbank aufzeichnen wollen, die der Benutzer **acctclrk** durchführt, dann ordnen Sie der Datenbank Berechtigungen wie folgt zu:

- Sie sind DBA der Datenbank.
- Alle anderen Benutzer haben für die Datenbank die CONNECT-Berechtigung. Sie können – müssen aber nicht – das RESOURCE-Zugriffrecht haben. Die DELETE-Berechtigung benötigen sie für die zu schützende Tabelle (in diesem Beispiel) nicht.
- Ihre gespeicherte Prozedur wurde unter Verwendung des Schlüsselwortes DBA erzeugt.
- Ihre gespeicherte Prozedur führt das Löschen durch und zeichnet die von einem bestimmten Benutzer durchgeführten Änderungen auf.

Schreiben Sie eine Prozedur ähnlich der in Bild 11-5 gezeigten; diese ändert eine Tabelle unter Verwendung einer vom Benutzer übergebenen Kundennummer. Handelt es sich um den Benutzer **acctclrk**, so wird der Löschvorgang in der Datei "updates" aufgezeichnet.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);

DELETE FROM customer
    WHERE customer_num = cnum;

IF username = "acctclrk" THEN
    SYSTEM "echo Delete from customer by acctclrk >> /mis/records/updates" ;
ENF IF
END PROCEDURE;
```

*Bild 11-5* ***Gespeicherte Prozedur, die Sätze löscht und Änderungen durch einen bestimmten Benutzer aufzeichnet***

Indem Sie die IF-Anweisung entfernen und die SYSTEM-Anweisung allgemeiner halten, können Sie alle durch die Prozedur durchgeführten Löschvorgänge aufzeichnen. Wenn Sie die in Bild 11-5 gezeigte Prozedur dahingehend ändern, dann wird sie wie in Bild 11-6 gezeigt aussehen.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);
LET username = USER ;
DELETE FROM tbname WHERE customer_num = cnum;

SYSTEM
 "echo Deletion made from customer table, by "||username ||">>/hr/records/deletes";

END PROCEDURE;
```

*Bild 11-6* ***Gespeicherte Prozedur, die Sätze löscht und Benutzer aufzeichnet***

# Das Erzeugen von Objekten einschränken

Wenn Sie kontrollieren wollen, welche Objekte wie erzeugt werden, verwenden Sie eine gespeicherte Prozedur mit den folgenden Vorgaben:

- Sie sind DBA der Datenbank.
- Alle anderen Benutzer haben die CONNECT-Berechtigung auf die Datenbank. Sie benötigen die RESOURCE-Berechtigung nicht.
- Ihre Prozedur (bzw. Abfolge von Prozeduren) wurde unter Verwendung des Schlüsselwortes DBA erzeugt.
- Ihre Prozedur (bzw. Abfolge von Prozeduren) erzeugt nach Ihren Definitionen Tabellen, Indizes und Views. Sie sollten die Änderungen zunächst mit einer Trainings-Datenbank vornehmen.

Ihre Prozedur könnte die Erzeugung einer oder mehrerer Tabellen und zugehöriger Indizes vornehmen, so, wie in Bild 11-7 gezeigt.

```
CREATE DBA PROCEDURE all_objects()

CREATE TABLE learn1 (intone SERIAL, inttwo INT NOT NULL, charcol CHAR(10) )
CREATE INDEX learn_ix ON learn1 (inttwo).
CREATE TABLE toys (name CHAR(15) NOT NULL UNIQUE,
            description CHAR(30), on_hand INT);
END PROCEDURE;
```

*Bild 11-7* ***Unter dem DBA-Modus laufende Prozedur, die einer Datenbank Tabellen und Indizes hinzufügt***

Die Prozedur **all_objects** kontrolliert das Hinzufügen von Spalten. Um die Prozedur ausführen zu können, müssen Sie allen Benutzern der Datenbank die RESOURCE-Berechtigung entziehen. Kein Benutzer außerhalb Ihrer Prozedur kann dann eine Tabelle, einen Index oder eine View erzeugen. Wenn Benutzer die Prozedur ausführen, dann haben sie vorübergehend DBA-Berechtigung. Dadurch kann beispielsweise die CREATE TABLE-Anweisung erfolgreich ausgeführt werden. Hinzu kommt, daß ein von einem Benutzer erzeugtes Objekt auch dem entsprechenden Benutzer gehört. Bei der Prozedur **all_objects** gehören die beiden Tabellen und Indizes demjenigen, der die Prozedur ausgeführt hat.

# Verwendung von Views

Eine *View* ist eine von Ihnen definierte Sichtweise auf eine oder mehrere Tabellen. Für die Benutzer ist zunächst kein Unterschied zwischen einer View und einer Tabelle zu bemerken. Man könnte eine View auch als synthetische Tabelle bezeichnen. Jedoch ist er keine Tabelle, eher die Synthese von Daten tatsächlich existierender realer Tabellen und anderen Views.

Die Grundlage einer View ist eine SELECT-Anweisung. Wenn Sie eine View erzeugen, dann definieren Sie damit eine SELECT-Anweisung, die den Inhalt der View zum Zeitpunkt des Zugriffs erzeugt. Ein Benutzer befragt eine View wiederum mit einer SELECT-Anweisung. Der Datenbankserver verschmilzt die SELECT-Anweisung des Benutzers mit der, die bei der Definition der View verwendet wird und führt dann die kombinierten Anweisungen aus.

Das Ergebnis sieht wie eine Tabelle aus. Es ist in dem Sinne eine Tabelle, daß eine View wiederum auf anderen Views basieren kann, bzw. auf Joins von Tabellen und anderen Views.

Da Sie den Inhalt einer View über eine selbst geschriebene SELECT-Anweisung bestimmen, können Sie über eine View eine der folgenden Aufgaben bewältigen:

- Den Zugang zu bestimmten Tabellenspalten einschränken

  Sie geben in der Auswahlliste der View nur die Namen der zugelassenen Spalten an.

- Den Zugang zu bestimmten Tabellensätzen einschränken

  Sie geben eine WHERE-Klausel an, die nur die zulässigen Sätze zurückliefert.

- Aufzunehmende oder zu korrigierende Werte auf bestimmte Bereiche überprüfen

  Sie können die Prüfungen durch die Klausel WITH CHECK OPTION verstärken (siehe Seite 11-30).

- Zugriff auf abgeleitete Daten ermöglichen, ohne daß Daten redundant in der Datenbank gespeichert werden müßten

  Sie geben die Ausdrücke, die die Daten ableiten, in der Auswahlliste bei der View-Definition an. Bei jeder Abfrage der View werden die Daten erneut abgeleitet.

- Einzelheiten einer komplizierten SELECT-Anweisung verbergen

  Sie können die Komplexität eines Joins über mehrere Tabellen in einer View verbergen, so daß weder die Benutzer noch die Anwendungsprogrammierer sie nachvollziehen müssen.

## Views erzeugen

Im folgenden Beispiel wird eine View erzeugt, der auf einer Tabelle der Beispiel-Datenbank basiert:

```
CREATE VIEW name_only AS
SELECT customer_num, fname, lname FROM customer
```

Die View gibt nur drei Spalten der Tabelle aus. Weil keine WHERE-Klausel enthalten ist, gibt es keine Einschränkungen bezüglich der auszugebenden Sätze.

Die im folgenden Beispiel erzeugte View basiert auf einer Tabelle, die nur bei Nutzung von NLS vorhanden ist. Der Name der View, der Tabelle und der Spalten enthalten im Spanischen vorkommende Buchstaben

```
CREATE VIEW çà_va AS
SELECT numéro, nom FROM abonnés;
```

Das folgende Beispiel basiert auf einem Join zwischen zwei Tabellen:

```
CREATE VIEW full_addr AS
SELECT address1, address2, city, state.sname, zipcode
    FROM customer, state
    WHERE customer.state = state.code
```

Die Tabelle mit den Namen der Bundesstaaten ermöglicht, daß Namen nur einmal in der Datenbank gespeichert werden. Über der View **full_addr** können Benutzer Adressen so erhalten, als wären die vollständigen Namen der Staaten in den Sätzen gespeichert. Die folgenden beiden Abfragen sind gleich:

```
SELECT * FROM full_addr WHERE customer_num = 105

SELECT address1, address2, city, state.sname, zipcode
    FROM customer, state
    WHERE customer.state = state.code
        AND customer_num = 105
```

Sie müssen sorfältig vorgehen, wenn Sie Views definieren, die auf Joins basieren. Views dieser Art sind nicht *änderbar*; das bedeutet, Sie können die in der View gezeigten Daten nicht ändern, also die Anweisungen UPDATE, DELETE oder INSERT nicht verwenden. (Änderungen über Views werden ab Seite 11-28 beschrieben.)

Im folgenden Beispiel werden die Sätze eingeschränkt, die ein Benutzer über eine View sehen kann:

```
CREATE VIEW no_cal_cust AS
    SELECT * FROM customer WHERE NOT state = "CA"
```

Diese View gibt alle Spalten der Tabelle **customer** aus, jedoch nur bestimmte Sätze. Im nächsten Beispiel wird über eine View der Zugriff der Benutzer auf diejenigen Sätze eingeschränkt, die für sie relevant sind:

```
CREATE VIEW my_calls AS
    SELECT * FROM cust_calls WHERE user_id = USER
```

Alle Spalten der Tabelle **cust_calls** sind verfügbar, jedoch nur für diejenigen Sätze, für die gilt, daß sie die Benutzerkennung des ausführenden Benutzers enthalten.

## Doppelt vorkommende Datensätze bei Views

Es ist möglich, daß eine View doppelt vorkommende Datensätze liefert; selbst dann, wenn die zugrundeliegende Tabelle nur eindeutige Sätze hat. Wenn die der View zugrundeliegende SELECT-Anweisung doppelt vorkommende Datensätze liefert, dann sieht es so aus, als würde die View selbst doppelt vorkommende Datensätze enthalten. Dies kann bei den Benutzern für erhebliche Verwirrung sorgen.

Man kann dieses Problem auf zwei Arten lösen. Ein Weg besteht darin, das Schlüsselwort DISTINCT in der Auswahlliste der View anzugeben. Dies macht es jedoch unmöglich, Änderungen an den Daten über eine View durchzuführen. Die Alternative besteht darin, nur eine Spalte oder eine Gruppe von Spalten auszuwählen, die mit Sicherheit eindeutig ist. (Sie können nur dann sicher sein, daß eindeutige Sätze zurückgeliefert werden, wenn Sie die Spalten eines Primär- oder Kandidatenschlüssels auswählen. Primär- und Kandidatenschlüssel werden in Kapitel 8 behandelt.)

## Einschränkungen bei Views

Da eine View nicht wirklich eine Tabelle ist, darf er nicht indiziert werden und er kann auch nicht als Objekt bei Anweisungen wie ALTER TABLE und RENAME TABLE angegeben werden. Die Spalten einer View können nicht mit RENAME COLUMN umbenannt werden. Wenn Sie irgendetwas an der Definition einer View ändern wollen, so müssen Sie die View löschen und neu erzeugen.

Weil sie mit der Abfrage der Benutzer gemischt wird, darf die SELECT-Anweisung, die die View begründet, keine der folgenden Klauseln enthalten:

| | |
|---|---|
| INTO TEMP | Die Benutzerabfrage könnte INTO TEMP enthalten; wenn die View ebenfalls diese Klausel enthalten würde, dann wäre nicht klar, wohin die Daten geschickt werden müßten. |
| UNION | Die Benutzerabfrage könnte UNION enthalten; in der Tat könnte die Abfrage über die View in der UNION-Klausel einer Benutzerabfrage stehen. Verschachtelte UNION-Klauseln werden jedoch nicht unterstützt. |
| ORDER BY | Die Benutzerabfrage könnte ORDER BY enthalten. Wenn die View dies ebenfalls enthält, könnte dies bei der Auswahl der Spalten oder der Sortierrichtung zu Konflikten führen. |

### Wenn die Basis verändert wird

Die Tabellen und Views, auf denen eine View basiert, können auf vielfältige Weise verändert werden. In einer View werden die meisten dieser Änderungen automatisch übernommen.

Wenn eine Tabelle oder eine View gelöscht wird, dann werden auch die hiervon abhängigen Views der gleichen Datenbank automatisch gelöscht.

Der einzige Weg, eine Viewdefinition zu verändern, besteht darin, sie zu löschen und neu zu erzeugen. Wenn Sie die Definition einer View ändern, von dem andere Views abhängen, so müssen Sie deshalb die anderen Views ebenfalls neu erzeugen (weil sie alle gelöscht wurden).

Wenn eine Tabelle umbenannt wurde, werden alle abhängigen Views der gleichen Datenbank auf den neuen Namen hin abgeändert. Wenn eine Spalte umbenannt wurde, werden alle von der entsprechenden Tabelle abhängigen Views der gleichen Datenbank dergestalt abgeändert, daß sie die richtige Spalte auswählen. Um dies an einem Beispiel zu sehen, können Sie die folgende View für die Tabelle **customer** erzeugen:

```
CREATE VIEW name_only AS
    SELECT customer_num, fname, lname FROM customer
```

Sorgen Sie nun dafür, daß die Tabelle **customer** auf die folgende Weise geändert wird:

```
RENAME COLUMN customer.lname TO surname
```

Um die Nachnamen der Kunden direkt auswählen zu können, müssen Sie nun in der Auswahlliste den neuen Spaltennamen angeben. Sieht man die Spalte jedoch durch einen View, so bleibt ihr Name unverändert. Die folgenden beiden Abfragen sind gleich:

```
SELECT fname, surname FROM customer

SELECT fname, lname FROM name_only
```

Wenn Sie bei einer Tabelle eine Spalte löschen, so werden die Views nicht geändert. Wenn sie gerade benutzt werden, wird der Fehler Nummer -217 ausgeben (Spalte ist in keiner angegebenen Tabelle enthalten). Der Grund hierfür besteht darin, daß man die Reihenfolge der Spalten in einer Tabelle ändern kann, indem man zunächst in der Tabelle eine Spalte löscht und dann eine

neue Spalte mit dem neuen Namen hinzufügt. Wenn dies so durchgeführt wird, funktionieren die Views, die auf dieser Tabelle basieren, auch weiterhin. Sie behalten die ursprüngliche Spaltenreihenfolge bei.

**INFORMIX-OnLine Dynamic Server** erlaubt, daß Sie eine View auf Tabellen oder Views begründen können, die sich in externen Datenbanken befinden. Die Änderungen an Tabellen oder Views in anderen Datenbanken werden in Views nicht nachvollzogen. Änderungen können solange verborgen bleiben, bis jemand auf die View eine SELECT-Anweisung durchführt und eine Fehlermeldung darüber zurückbekommt, daß die externe Tabelle geändert wurde.

## Änderungen über eine View

Sie können den Inhalt von Views genauso ändern wie den Inhalt von Tabellen. Die Änderungen können in einigen Views durchgeführt werden und in anderen nicht, abhängig von der jeweiligen SELECT-Anweisung. Die Einschränkungen hängen davon ab, ob man DELETE-, UPDATE- oder INSERT-Anweisungen verwendet.

Änderungen über eine View sind nicht möglich, wenn die SELECT-Anweisung eine der folgenden Eigenschaften aufweist:

- Ein Join über zwei oder mehr Tabellen

  Wenn der Datenbankserver versuchen würde, die geänderten Daten korrekt über die verbundenen Tabellen zu verteilen, kann es zu Problemen kommen.

- Eine Mengenfunktion oder eine GROUP BY-Klausel

  Die Datensätze der View repräsentieren eine Vielzahl miteinander kombinierter Datensätze; der Datenbankserver kann geänderte Sätze nicht auf sie verteilen.

- Das Schlüsselwort DISTINCT oder sein Synonym UNIQUE

  Die Sätze der View repräsentieren die Auswahl einer Vielzahl von möglicherweise doppelt vorkommenden Sätzen. Der Datenbankserver kann nicht feststellen, welcher der Original-Datensätze geändert werden sollen.

Wenn eine View all diese Dinge vermeidet, so entspricht ein Satz einer View genau einem Satz einer Tabelle. Eine solche View ist *veränderbar* (natürlich können einzelne Benutzer eine View nur dann ändern, wenn sie die entsprechenden Berechtigungen haben. Berechtigungen auf Views werden ab Seite 11-32 behandelt.)

## Löschen über eine View

Bei einer View, über die man Daten ändern kann, kann man auch die DELETE-Anweisung so verwenden, als würde man in einer Tabelle arbeiten. Der Datenbankserver löscht die ausgewählten Sätze der zugrunde liegenden Tabelle.

## Korrekturen über einer View

Bei einer änderbaren View können Sie die UPDATE-Anweisung genauso verwenden, als würden Sie in einer Tabelle arbeiten. Eine änderbare View kann jedoch auch abgeleitete Spalten enthalten, d. h. Spalten, die über Ausdrücke in der Auswahlliste der CREATE VIEW-Anweisung erzeugt wurden. Abgeleitete Spalten (manchmal auch als *virtuelle* Spalten bezeichnet) kann man nicht korrigieren.

Wenn eine Spalte aus einer einfachen arthmetischen Kombination einer Spalte mit einer Konstanten abgeleitet wird (z. B. `order_date+30`), so kann der Datenbankserver im Prinzip ermitteln, wie der Ausdruck aufzulösen ist (in diesem Fall durch Subtraktion von 30 vom zu korrigierenden Wert) und könnte die Änderungen durchführen. Es sind jedoch auch kompliziertere Ausdrücke denkbar, die nicht alle einfach aufgelöst werden könnten. Deshalb unterstützt der Datenbankserver die Korrektur von abgeleiteten Spalten überhaupt nicht.

Bild 11-8 zeigt einen änderbaren View, der eine abgeleitete Spalte enthält; eine UPDATE-Anweisung kann deshalb nicht ausgeführt werden.

```
CREATE VIEW call_response(user_id,received,resolved,duration)AS
    SELECT user_id,call_dtime,res_dtime,res_dtime-call_dtime
        FROM cust_calls
        WHERE user_id = USER

UPDATE call_response SET resolved = TODAY
    WHERE resolved IS NULL
```

*Bild 11-8* ***Änderbare View und UPDATE-Anweisung***

Die Spalte **duration** der View kann nicht korrigiert werden, weil die Spalte einen Ausdruck repräsentiert (der Datenbankserver kann nicht entscheiden, wie ein korrigierter Wert auf zwei im Ausdruck genannte Spalten verteilt werden sollte). Solange jedoch keine abgeleiteten Spalten in der SET-Klausel angegeben werden, kann die Korrektur so durchgeführt werden, als wäre die View eine Tabelle.

Eine View kann doppelt vorkommende Sätze zurückliefern, obwohl die Sätze der zugrunde liegenden Tabelle eindeutig sind. Man kann die doppelt vorkommenden Datensätze nicht voneinander unterscheiden. Wenn Sie einen einzelnen Satz einer Reihe von doppelt vorkommenden Datensätzen korrigieren, (indem Sie z. B. für die Korrektur einen Cursor mit WHERE CURRENT verwenden), können Sie nicht sicher sein, welcher Satz in der zugrunde liegenden Tabelle korrigiert wird.

## Neuaufnahme über eine View

Man kann über eine View dann Datensätze neu aufnehmen, wenn sie änderbar ist *und* keine abgeleiteten Spalten enthält. Der Grund für die zweite Einschränkung besteht darin, daß bei einem neu aufzunehmenden Datensatz Werte für alle Spalten vorliegen müssen. Der Datenbankserver kann jedoch nicht feststellen, wie ein aufzunehmender Wert über einen Ausdruck verteilt werden sollte. Der Versuch der Neuaufnahme übe rdie View **call_response** – gezeigt in Bild 11-8

würde scheitern.

Wenn eine änderbare View keine abgeleiteten Spalten enthält, kann man über ihn Datensätze so aufnehmen, als wäre sie eine Tabelle. Der Datenbankserver fügt jedoch bei allen Spalten, die von der View nicht versorgt werden, NULL-Werte ein. Erlaubt eine derartige Spalte jedoch keine NULL-Werte, so erfolgt ein Fehler und die Neuaufnahme scheitert.

## Die WITH CHECK OPTION-Klausel verwenden

Sie können Datensätze über eine View aufnehmen, die nicht der in der View gestellten Bedingung genügen; dies ist also ein Satz, der nach der Eingabe über die View nicht mehr zu sehen ist. Man kann außerdem einen Satz über eine View so korrigieren, daß er die in der View gestellten Bedingungen nicht weiter erfüllt.

Wenn dies verhindert werden soll, können Sie beim Erzeugen der View die WITH CHECK OPTION-Klausel hinzufügen. Diese Klausel sorgt dafür, daß der Datenbankserver jeden einzufügenden oder zu korrigierenden Satz daraufhin abprüft, ob er die Bedingungen erfüllt, die in der WHERE-Klausel der View-Definition gestellt wurden. Der Datenbankserver weist eine Operation mit einer Fehlermeldung zurück, wenn die geforderte Bedingung nicht erfüllt wurde.

In Bild 11-8 wird eine View mit dem Namen **call_response** wie folgt definiert:

```
CREATE VIEW call_response(user_id,received,resolved,duration)AS
    SELECT user_id,call_dtime,res_dtime,res_dtime-call_dtime
        FROM cust_calls
        WHERE user_id = USER
```

Es ist möglich, die Spalte **user_id** in der View zu verändern, wie das folgende Beispiel zeigt:

```
UPDATE call_response SET user_id = "lenora"
    WHERE received BETWEEN TODAY AND TODAY-7
```

Die View liefert nur Sätze, für die gilt, daß **user_id** identisch ist mit USER. Wenn diese Korrektur von einem Benutzer mit dem Namen **tony** durchgeführt wird, bleibt der korrigierte Satz für die View verborgen. Man kann jedoch eine View definieren wie im folgenden Beispiel gezeigt:

```
CREATE VIEW call_response(user_id,received,resolved,duration)AS
    SELECT user_id,call_dtime,res_dtime,res_dtime-call_dtime
        FROM cust_calls
        WHERE user_id = USER
WITH CHECK OPTION
```

Dann wird der Korrekturversuch von **tony** mit einem Fehler abgewiesen.

Mit der Klausel WITH CHECK OPTION können Sie jede beliebige Datenüberprüfung festlegen, die man über einen Booleschen Ausdruck formulieren kann. Man kann beispielsweise eine View für eine Tabelle so erzeugen, daß alle logisch erforderlichen Datenüberprüfungen über Bedingungen in der WHERE-Klausel ausgedrückt werden. Weiterhin kann man dafür sorgen, daß alle Änderungen an der Tabelle über die View durchgeführt werden.

```
CREATE VIEW order_insert AS
    SELECT * FROM orders O
        WHERE order_date = TODAY -- no back-dated entries
            AND EXISTS -- ensure valid foreign key
                (SELECT * FROM customer C
                    WHERE O.customer_num = C.customer_num)
            AND ship_weight < 1000 -- reasonableness checks
            AND ship_charge < 1000
WITH CHECK OPTION
```

Will man lediglich die Daten der Tabelle **orders** anzeigen lassen, dann ist dies wegen des Schlüsselworts EXISTS und anderer Bedingungen, die zur Ausgabe der vorhandenen Sätze alle erfüllt sein müssen, ein höchst komplizierter und damit uneffizienter View. Werden über diesn View allerdings Neuaufnahmen für die Tabelle **orders** durchgeführt, (und Sie verwenden nicht bereits Integritätsprüfungen zur Datenüberprüfung), so werden verschiedene Sicherheitsprüfungen ausgeführt. So ist es in diesem Beispiel unmöglich, daß zurückdatierte Aufträge, falsche Kundennummern oder unnormale Versandgewichte und Versandkosten aufgenommen werden.

# Berechtigungen und Views

Wenn Sie eine View *erzeugen*, so überprüft der Datenbankserver Ihre Berechtigungen auf die zugrunde liegenden Tabellen und Views. Wenn Sie eine View *verwenden*, dann werden nur die Berechtigungen auf die View selbst überprüft.

## Berechtigungen beim Erzeugen einer View

Wenn Sie eine View erzeugen, dann prüft der Datenbankserver, ob Sie alle erforderlichen Berechtigungen zur Ausführung der SELECT-Anweisung haben. Ist dies nicht der Fall, wird die View nicht erzeugt.

Diese Überprüfung stellt sicher, daß sich Benutzer keinen unzulässigen Zugriff auf eine Tabelle verschaffen, indem sie eine View für diese Tabelle erzeugen und dann in der View unerlaubte Abfragen vornehmen.

Nachdem Sie eine View erzeugt haben, erteilt der Datenbankserver Ihnen, dem Erzeuger und Eigentümer der View, hierfür die SELECT-Berechtigung. Anders als bei einer neu erzeugten Tabelle gibt es in diesem Fall keine automatische Zuweisung an **public**.

Der Datenbankserver überprüft die View-Definition daraufhin, ob die View veränderbar ist. Ist dies der Fall, dann weist der Datenbankserver Ihnen die Berechtigungen für INSERT, DELETE und UPDATE für die View zu. Voraussetzung hierfür ist allerdings, daß Sie auch über diese Berechtigungen für die zugrundeliegenden Tabelle oder die View verfügen. Haben Sie bei der zugrundeliegenden Tabelle nur die INSERT-Berechtigung, so werden Sie auch bei der View nur die INSERT-Berechtigung bekommen.

Da man einenView weder in seiner Struktur verändern noch indizieren kann, werden die ALTER- und INDEX-Berechtigungen einer View niemals zugewiesen.

# Berechtigungen zum Verwenden einer View

Wenn Sie eine View benutzen, so überprüft der Datenbankserver lediglich die Berechtigungen, die Ihnen bei der View zugewiesen wurden. Er überprüft *nicht* Ihre Berechtigungen auf die darunterliegendenTabellen.

Wenn Sie eine View erzeugen, dann erhalten Sie diejenigen Berechtigungen, wie im vorausgegangenen Abschnitt beschrieben. Wenn Sie nicht der Ersteller der View sind, so besitzen Sie diejenigen Berechtigungen, die Ihnen vom Ersteller bzw. einer Person zugeteilt wurden, die die WITH GRANT OPTION Berechtigungen hat.

Hieraus ergibt sich, daß Sie eine Tabelle erzeugen können und ihr die Berechtigung von **public** entziehen können. Anschließend können Sie der Tabelle über eine View begrenzte Berechtigungen zuweisen. Dies kann entsprechend dem vorausgegangenen Beispiel bei der Tabelle **hr_data** demonstriert werden. Die Definition dieser Tabelle wird in Bild 11-9 wiederholt.

```
CREATE TABLE hr_data
    (
    emp_key INTEGER,
    emp_name CHAR(40),
    hire_date DATE,
    dept_num SMALLINT,
    user_id CHAR(18),
    salary DECIMAL(8,2)
    performance_level CHAR(1)
    performance_notes TEXT
    )
```

*Bild 11-9* ***Eine Tabelle mit vertraulichen Daten der Angestellten (Duplikat von Bild 11-1)***

Beim Beispiel von Bild 11-9 dreht es sich hauptsächlich darum, daß der Tabelle direkt Berechtigungen zugewiesen wurden. Die folgenden Beispiele schlagen einen anderen Weg ein. Stellen Sie sich vor, daß nach Erzeugen der Tabelle die folgende Anweisung ausgeführt würde:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

(Bei einer ANSI-kompatiblen Datenbank ist dies nicht notwendig.) Nun erzeugen Sie für die einzelnen Benutzerklassen eine Reihe von Views. Für diejenigen, die nur einen Lese-Zugriff auf die nicht-sensiblen Spalten bekommen sollen, erzeugen Sie den folgenden View:

```
CREATE VIEW hr_public AS
    SELECT emp_key, emp_name, hire_date, dept_num, user_id
        FROM hr_data
```

Die Benutzer, denen die SELECT-Berechtigung für diese View zugeordnet wurde, können die nicht-sensiblen Daten zwar sehen, aber nicht ändern. Für Mitarbeiter der Personalabteilung, die neue Sätze aufnehmen müssen, erzeugen Sie eine andere View:

```
CREATE VIEW hr_enter AS
    SELECT emp_key, emp_name, hire_date, dept_num
        FROM hr_data
```

Diesen Benutzern erteilen Sie für die View sowohl das SELECT-, als auch die INSERT-Berechtigung. Weil Sie, der Erzeuger von Tabelle und View, die INSERT-Berechtigung auf die Tabelle und die View besitzen, können Sie die INSERT-Berechtigung auf die View auch anderen erteilen, die keine Berechtigungen auf die Tabelle haben.

Im Interesse der Mitarbeiter der Organisationsabteilung, die Personalnummern eingeben oder korrigieren können müssen, erzeugen Sie noch eine weitere View:

```
CREATE VIEW hr_MIS AS
    SELECT emp_key, emp_name, user_id
        FROM hr_data
```

Diese View unterscheidet sich von den vorangegangenen dadurch, daß sie die Abteilungsnummer und das Einstellungsdatum nicht enthält.

Schließlich benötigen die Manager den Zugriff auf alle Spalten, und sie müssen die Möglichkeit erhalten, die Leistungsbeurteilungen ihrer eigenen Mitarbeiter bearbeiten zu können. Diese Forderungen können auf die folgende Weise erfüllt werden:

Die Tabelle **hr_data** enthält für jeden Angestellten eine Abteilungsnummer und eine Benutzerkennung. Es ist davon auszugehen, daß die Manager auch Mitglieder der Abteilung sind, die sie managen. Die folgende View begrenzt den Zugriff der Manager auf diejenigen Datensätze, die nur die eigenen Mitarbeiter beinhalten:

```
CREATE VIEW hr_mgr_data AS
    SELECT * FROM hr_data
        WHERE dept_num =
            (SELECT dept_num FROM hr_data
                WHERE user_id = USER)
        AND NOT user_id = USER
```

Als letztes ist dafür zu sorgen, daß die Manager nicht ihren eigenen Datensatz in der Tabelle bearbeiten. Dies kann man dergestalt lösen, daß man dem Manager zwar die UPDATE-Berechtigung für diese View erteilt, jedoch nur auf ausgewählte Spalten, wie die folgende Anweisung zeigt:

```
GRANT SELECT, UPDATE (performance_level, performance_notes)
    ON hr_mgr_data TO peter_m
```

# Zusammenfassung

Bei einer Datenbank mit öffentlich zugänglichen Daten bzw. bei einer Datenbank, die nur von Ihnen oder anderen vertrauenswürdigen Personen bearbeitet wird, ist die Sicherheit kein vordringliches Problem. Dann sind nur wenige der in diesem Kapitel vorgestellten Maßnahmen erforderlich. Aber im gleichen Maße, in dem immer mehr Personen die Verwendung und Änderung der Daten erlaubt wird, und in gleichem Maße, in dem die Daten immer vertraulicher werden, müssen Sie immer mehr Zeit aufwenden und immer raffiniertere Kontrollen für die Zugangswege zu den Daten errichten.

Die hier behandelten Techniken kann man den zwei folgenden Gruppen zuordnen:

- Daten vertraulich halten.

  Wenn die Datenbank in Betriebssystem-Dateien gehalten wird, kann man die Möglichkeiten des Betriebssystems dazu verwenden, den Zugriff auf die Datenbank zu unterbinden. Auf jeden Fall kann man die Vergabe bzw.

der Entzug der CONNECT-Berechtigung dazu verwenden, Personen von der Datenbank fern zu halten.

Unabhängig von der jeweiligen Benutzerklasse oder den jeweiligen Berechtigungs-Ebenen benötigen alle Benutzer die CONNECT-Berechtigung. Über Berechtigungen auf Tabellenebene können Sie den Zugriff auf vertrauliche Tabellenspalten verweigern. Oder Sie können eine in derDatenbank gespeicherte Prozedur dazu verwenden, begrenzten Zugriff auf vertrauliche Tabellen oder Spalten zu gewähren. Zusätzlich können Sie jeglichen Zugriff auf Tabellen verwehren und ihn lediglich über Views erlauben, bei denen vertrauliche Sätze nicht enthalten sind.

- Änderungen an den Daten und der Datenbank-Struktur kontrollieren.

  Indem Sie die RESOURCE-, ALTER-, REFERENCES- und DBA-Berechtigungen verweigern, sorgen Sie für die Integrität des Datenmodells. Dadurch, daß Sie die Zuweisung der DELETE- und UPDATE-Berechtigungen kontrollieren können und dadurch, daß Sie die UPDATE-Berechtigung für so wenig Spalten wie möglich gewähren, stellen Sie sicher, daß nur berechtigte Personen die Daten ändern. Indem Sie die INSERT-Berechtigung nur für Views vergeben, die über Ausdrücke logische Prüfungen an den Daten vornehmen, stellen Sie sicher, daß nur konsistente Daten aufgenommen werden. Alternativ hierzu können Sie die Neuaufnahme und Änderung der Daten bzw. Änderung der Datenbank insgesamt dadurch kontrollieren, daß Sie den Zugriff über gespeicherte Prozeduren einschränken.

Kapitel

# Datenbankserver und Netzwerke

12

# Kapitelüberblick

Dieses Kapitel vermittelt Ihnen einen Überblick über den Einsatz von Datenbanken in Rechnernetzen. Sie lernen die grundlegende Terminologie und verschiedene Netz-Konfigurationen kennen. Außerdem erfahren Sie, wie eine lokale Verbindung oder eine Netz-Verbindung so aufgebaut wird, daß das Anwendungsprogramm seine Daten auf dem Datenbankserver finden kann.

Es werden mehrere Netz-Konfigurationen vorgestellt, die Sie mit Informix-Datenbanken verwenden können. Ihre jeweilige Leistungsfähigkeit wird beschrieben, sowie der erforderliche Aufwand bei Ihrer Benutzung. Im einzelnen geht es um folgende Konfigurationen:

- Alles liegt auf einem Rechner
- Eine einfache Netz-Konfiguration
- Multiple-Verbindungen in einem Netz
- Datenverwaltung von Datenbankservern, die nicht von Informix stammen (TP/XA)

Es reicht nicht, einfach ein Netz aufzubauen. Sie müssen sich noch einmal vergegenwärtigen, wie Ihre Anwendungsprogramme und Ihr Datenbankserver Daten austauschen. Dieses Kapitel geht unter anderem auf die folgenden Fragen ein:

- Verteilte Daten
- Verbindung zu den Daten aufbauen
- Daten schützen
- Synonymketten
- Transparenz im Netz

Die letzten Abschnitte befassen sich mit der Frage, wie Sie Ihre Daten im Netz schützen.

## Was ist ein Netzwerk

Ein Netzwerk ist eine Gruppe von Rechern und anderen Geräten, die mit einem Kommunikationssystem verbunden sind, um Daten und andere Resourcen zu teilen.

Um ein Netzwerk zum Laufen zu bringen, müssen Sie eine Reihe von Hard- und Softwaredetails in den Griff bekommen. Es gibt sehr viele dieser Punkte und sie ändern sich schnell. Aus diesem Grund können Sie in diesem Buch nicht aufgeführt werden. Dieses Buch wird stattdessen die Fragen konzeptuell behandeln, die bei der Arbeit mit einem Netzwerk in der Regel entstehen. Weitere Informationen erhalten Sie in dem Handbuch zu Ihrem Informix Client-/Server-Produkt und/oder in einem Lehrbuch über Computernetze.

# Konfigurationen des Datenbankverwaltungssystems (DBMS)

Ein relationales Datenbankverwaltungssystem RDBMS (*Relational Database Management Systems*) beinhaltet alle Komponenten, die man benötigt, um eine relationale Datenbank aufzubauen. Die RDBMS von Informix bestehen aus mehreren Teilen: der Benutzerschnittstelle (Anwendungsprogramm), dem Datenbankserver und den Daten selbst. Am einfachsten ist, sich all diese Elemente auf einem Rechner vorzustellen. Aber es sind auch andere Anordnungen denkbar. Bei einigen liegen Benutzerschnittstelle und Datenbankserver auf verschiedenen Rechnern, und die Daten liegen verteilt auf weiteren Rechnern.

Das Anwendungsprogramm eines RDBMS muß nicht geändert werden, wenn es im Netz laufen soll. Die Kommunikationswerkzeuge übernehmen die Aufgabe, die Datenbankserver zu suchen und die Verbindung herzustellen. Für das Anwendungsprogramm besteht kein Unterschied zwischen einer lokalen Datenbank und einer solchen im Netz.

# Eine Einplatz-Konfiguration

Bild 12-1 zeigt ein einfaches DBMS auf einem Einplatzsystem. Diese Organisationsform findet man in der Regel auf einem PC unter MS-DOS. Einplatz-Konfigurationen unter UNIX sind eher ungewöhnlich, sind aber trotzdem möglich. Ein Beispiel für so ein UNIX-System wäre eine Workstation in einer Entwicklungsumgebung.

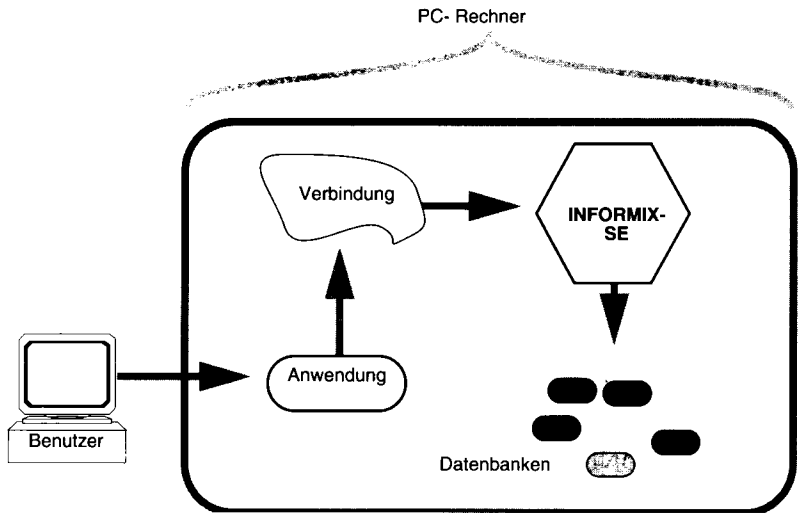



*Bild 12-1* ***Ein Datenbankverwaltungssystem auf einem PC***

Bild 12-1 zeigt folgende Komponenten:

- Ein Anwendungsprogramm. Dieses Programm könnte beispielsweise mit **INFORMIX-4GL** oder der Sprache C mit eingebettetem SQL erstellt worden sein. Denkbar wäre auch, daß es sich um compilierte Bildschirmmasken und Listengeneratoren handelt.
- Eine Verbindung. In einem solchen System wie dem hier veranschaulichten ist die Kommunikationskomponente so einfach, daß sie häufig nicht gezeigt wird.
- Ein Datenbankserver. Der Datenbankserver empfängt vom Anwendungsprogramm Abfragen, durchsucht die Datenbank und liefert die gesuchte Information an die Anwendung zurück. Der Datenbankserver verwaltet die Datenbank.

  Da unsere Abbildung einen PC zeigt, muß es sich bei dem Datenbankserver um **INFORMIX-SE** handeln. Der Datenbankserver ist hier ein lokaler

Server, da er auf dem gleichen Rechner liegt wie das Anwendungsprogramm.

- Die Datenbank. Sie ist in der Regel auf einer Festplatte gespeichert.

  Falls es sich um ein UNIX-System mit **INFORMIX-OnLine Dynamic Server** handeln würde, könnte die Datenbank auf dem gleichen oder auf einem anderen Medium liegen, beispielsweise auf einem WORM-Laufwerk („Write Once, Read Many times", also ein System, das man einmal beschreiben und immer wieder lesen kann). Diese Art von Laufwerk wird von **INFORMIX-OnLine/Optical** unterstützt.

### Vor- und Nachteile des Einplatz-Systems

Das Einplatz-System kann leicht erstellt und gewartet werden; außerdem bietet es den schnellsten Zugriff auf die Daten. Allerdings stehen diese Daten den Benutzern anderer Rechner nicht zu Verfügung. Außerdem könnte die Größe der Datenbank bzw. die benötigte Verarbeitungskapazität die Leistungsfähigkeit des Systems übersteigen.

## Eine lokale Mehrplatz-Konfiguration

Bild 12-2 zeigt ein Mehrplatz-System mit einem lokalen Datenbankserver. So eine Konfiguration findet man häufig auf Rechnern mit dem Betriebssystem UNIX.

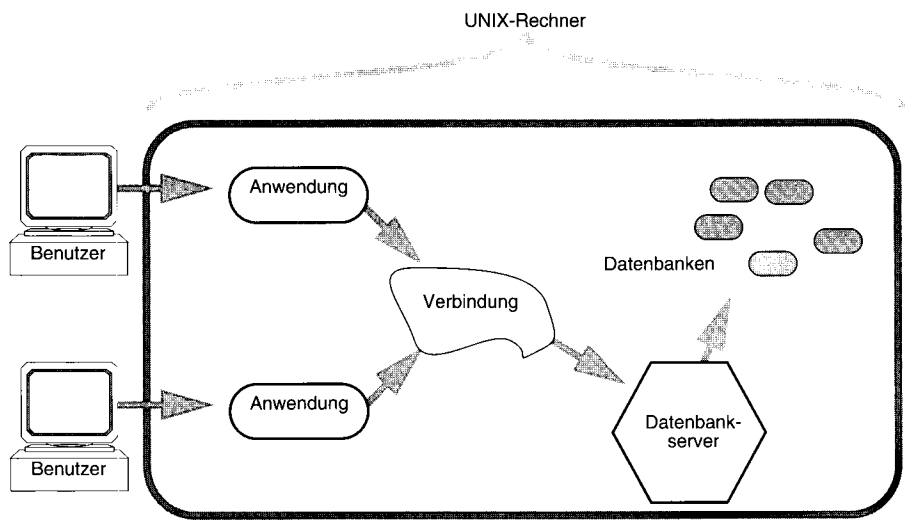



*Bild 12-2* ***Ein Datenbankverwaltungssystem auf einem UNIX-Rechner***

Die Komponenten dieses Systems ähneln denen des Systems aus Bild 12-1:

- Anwendungsprogramme: Zwei oder mehr Anwendungen verwenden den gleichen Datenbankserver, um Informationen aus der Datenbank zu erhalten. Sie haben hier entweder zwei Benutzer an verschiedenen Terminals (wie in dieser Abbildung), oder es sind gleichzeitig mehrere Fenster auf dem Bildschirm eines Rechners geöffnet.
- Verbindung: Bei einem lokalen UNIX System kommen zwei Arten von Verbindungen vor:
  - o Inter-process Kommunikation (IPC)
  - o Netzwerk-Verbindung

  IPC ist eine UNIX Funktionalität, die eine sehr schnelle Datenübertragung zwischen Anwendungsprogramm und Datenbankserver ermöglicht. Sie ist nur dann vorhanden, wenn Anwendungsprogramm und Datenbankserver auf dem selben Rechner liegen. Datenbanken unter **INFORMIX-SE** verwenden eine IPC-Verbindung, die als *Unnamed Pipes* bezeichnet wird. **INFORMIX-OnLine Dynamic Server** Datenbanken verwenden eine IPC-Verbindung, die als *Shared Memory* bezeichnet wird.
- Ein Datenbankserver, entweder **INFORMIX-OnLine Dynamic Server** oder **INFORMIX-SE** Datenbanken.

## Vor- und Nachteile der lokalen Mehrbenutzer-Konfiguration

Eine Konfiguration für mehrere Anwender stellt einen besseren Zugang zu den Daten dar als eine Einplatz-Konfiguration. Eine lokale Datenbank ist leicht zu erstellen und zu warten. Wie bei einem Einplatz-System sind die Daten allerdings nicht für Benutzer von anderen Rechnern verfügbar. Außerdem könnte die Größe der Datenbank bzw. die benötigte Verarbeitungskapazität die Leistungsfähigkeit des Systems übersteigen.

*IPC Shared Memory* ermöglicht eine sehr schnelle Kommunikation zwischen Client-Anwendungsprogramm und Datenbankserver. Allerdings ist dieses System anfällig bei Programmierfehlern, falls das Anwendungsprogramm den Speicher direkt adressiert oder falls die Daten-Arrays überindiziert werden. Diese Probleme treten nicht auf, wenn Sie mit der IPC-Variante *Unnamed Pipes* oder mit einer Netzwerk-Verbindung arbeiten. (Weitere Information hierzu erhalten Sie im Abschnitt "Einplatz-System im Netzwerk" auf Seite 12-9.)

# Eine Remote-Konfiguration

Bild 12-3 zeigt eine Remote-Verbindung, bei der das Anwendungsprogramm auf dem einen Rechner und der Datenbankserver sowie die Datenbanken auf einem anderen Rechner im Netz liegen. Im Gegensatz dazu sind die Datenbankserver auf Bild 12-1 und Bild 12-2 lokale Server.

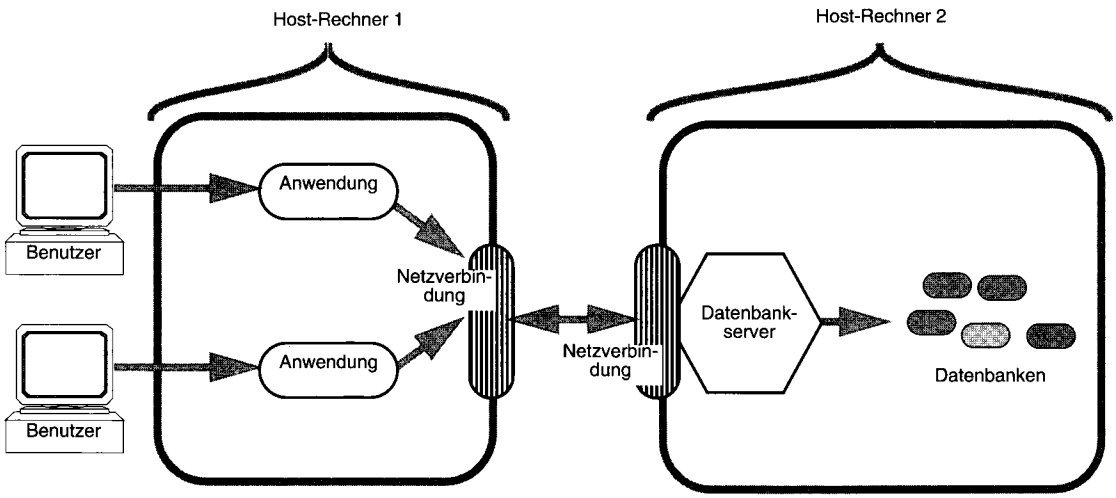



*Bild 12-3* ***Eine einfache Netz-Verbindung***

In Bild 12-3 könnte es sich bei der Anwendung um ein Programm handeln, das mit **ESQL/C** oder mit **ESQL/COBOL** geschrieben wurde. Als Datenbankserver kommen **INFORMIX-OnLine Dynamic Server** (UNIX) oder **INFORMIX-OnLine for NetWare** (PC) in Frage.

Einzelne Rechner im Netz werden als Host-Rechner bezeichnet. Der Datenbankserver in Bild 12-3 ist ein Remote-Datenbankserver (remote = fern), da er sich auf einem anderen Host-Rechner bzw. auf einer anderen Station befindet als das Anwendungsprogramm. Ein Datenbankserver kann in Bezug auf verschiedene Anwendungsprogramme sowohl lokal als auch remote sein, wie später in diesem Kapitel in Bild 12-6 gezeigt wird.

## Vor- und Nachteile der Remote-Konfiguration

Die Konfiguration aus Bild 12-3 ist ein Beispiel der sog. *verteilten Verarbeitung*. Bei verteilter Verarbeitung arbeiten mehrere Rechner an einer einzigen Aufgabe. In unserem Beispiel verarbeitet Host-Rechner1 das Anwendungsprogramm, d. h. er steuert die Bildschirmausgabe, erzeugt Listen und

Ausdrucke. Gleichzeitig übernimmt Host-Rechner2 die Aufgaben des Datenbankservers. In einem Netz können Client-Anwendungsprogramme also auf Daten von verschiedenen Rechnern zuzugreifen.

Anfragen an den Datenbankserver über die Netzwerk-Verbindung werden nicht so schnell beantwortet wie Anfagen von Datenbankservern über lokale IPC-Kommunikation. Dies liegt daran, daß zusätzliche Rechenzeit benötigt wird, um die Daten für den Versand im Netz aufzubereiten und auch die Übertragung selbst Zeit kostet. Außerdem ist ein Netz etwas schwieriger zu konfigurieren und zu pflegen als ein lokales System mit IPC-Kommunikation.

## Einplatz-System im Netzwerk

Bild 12-4 zeigt eine Anordnung, die sich so verhält, als ob sich mehrere Stationen *auf einer Maschine* befinden würden. Bei dieser Konfiguration, die auch als *Local Loopback* bezeichnet wird, befinden sich alle Komponenten auf demselben Rechner. Die Verbindungen zwischen ihnen sind jedoch so aufgebaut, als ob sie über ein Netz verbunden wären.

Da die Verbindung mit Netz-Software arbeitet, scheint der Datenbankserver für das Anwendungsprogramm auf einem fernen (remoten) Rechner zu liegen. Die gepunktete Linie markiert die Trennung zwischen den beiden „Stationen". Der Datenbankverwalter konfiguriert das System für eine lokale oder eine Local Loopback-Verbindung (oder für beide).

Die Software für das Anwendungsprogramm und für den Server können im gleichen oder in zwei verschiedenen Verzeichnissen liegen.

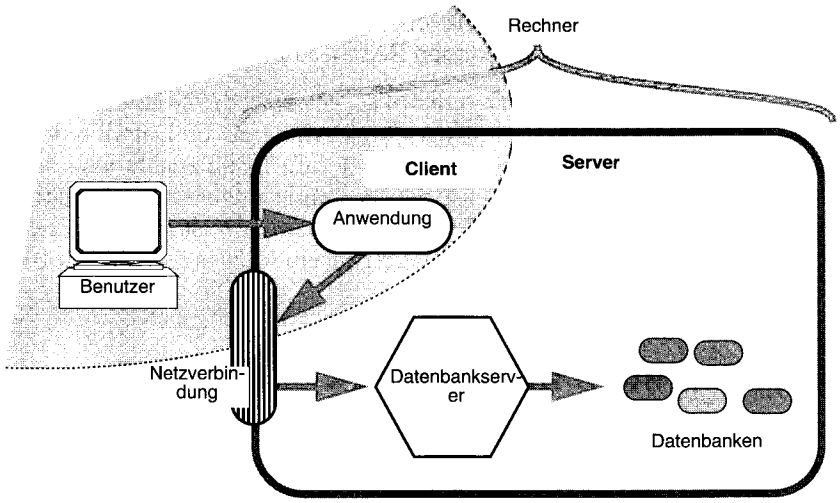



*Bild 12-4 Beispiel für einen Local Loopback*

### Vor- und Nachteile des Local Loopback

Mit Hilfe des Local Loopback können Sie Netzwerkoperationen testen, ohne eine tatsächliche Verbindung zu einem entfernten Rechner zu haben. Allerdings ist diese Verbindung nicht so schnell wie eine lokale IPC-Verbindung. Anders als die IPC-Verbindung Shared Memory ist Local Loopback nicht anfällig bei Programmierfehlern, falls das Anwendungsprogramm den Speicher direkt adressiert oder falls die Daten-Arrays überindiziert werden (Siehe "Vor- und Nachteile der lokalen Mehrbenutzer-Konfiguration" auf Seite 12-7).

## Verteilte Datenbanken

Zwar erlaubt Ihnen ein Netzwerk, Daten und Anwendungsprogramme getrennt zu halten, aber das Anwendungsprogramm kann immer nur auf die jeweils aktuelle Datenbank zugreifen. Mit den meisten Datenbankservern können Sie immer nur auf Tabellen der aktuellen Datenbank arbeiten.

In einer verteilten Datenbank liegt die Information über verschiedene Datenbanken verteilt. Dies ist aber so organisiert, daß der Endbenutzer den Eindruck hat, nur mit einer einzigen Datenbank zu arbeiten. Die Daten können dabei von verschiedenen relationalen Datenbankverwaltungssystemen (RDBMS) verwaltet werden, sie können auf verschiedenen Rechnern mit verschiedenen Betriebssystemen und verschiedenen Netzwerken liegen.

Unter **INFORMIX-OnLine Dynamic Server** können Sie Daten in verschiedenen Datenbanken auf dem gesamten Netz abfragen. Wenn **INFORMIX-OnLine Dynamic Server** mit **INFORMIX-TP/XA** ausgestattet wird, können Sie globale Transaktionen programmieren, die auf zahlreiche Rechnersysteme zugreifen und sogar auf Datenbanken verschiedener Hersteller, solange diese XA-kompatibel sind. **INFORMIX-SE** unterstützt keine verteilten Datenbanken.

### Vor- und Nachteile verteilter Datenbanken

Verteilte Datenbanken sind deshalb sinnvoll, weil Organisationen, die die Datenbanken benutzen, von ihrer eigenen Struktur her ebenfalls häufig verteilt sind - sei es geographischer oder struktureller Art. Verteilte Datenbanken haben folgende Vorteile:

- Lokale Daten können dort lokal abgelegt werden, wo Sie am meisten benötigt und am leichtesten gewartet werden.
- Daten von entfernten Rechnern stehen allen zur Verfügung.
- Aus Gründen der Datensicherheit können Duplikate der Daten erstellt werden.

Verteilte Datenbanken haben folgende Nachteile:

- Die Verwaltung von verteilten Datenbanken ist aufwendiger als die von Einplatz-Systemen.
- Der Zugriff über das Netz ist immer langsamer als der lokale Zugriff.

## Verteilte Datenbanken mit Datenbanksystemen verschiedener Hersteller

Wenn Sie Ihre Daten in Datenbanksystemen verschiedener Hersteller ablegen und gleichzeitig den Zugang über ein gemeinsames Anwendungsprogramm herstellen wollen, benötigen Sie **INFORMIX-TP/XA**. Dieses Produkt enthält eine Bibliothek von Funktionen, die es dem Datenbankserver **INFORMIX-OnLine Dynamic Server** erlauben, in einer verteilten Transaktionsumgebung , die dem von X/Open gesetzten Schnittstellen-Standard entspricht, als sogenannte *Resource Manager* zu fungieren.

Die Terminologie von X/Open unterscheidet sich von der, die Informix in seinen Produkten und den Handbüchern verwendet. Ein sogenannter *Transaction Manager* fungiert als Vermittler und leitet Anfragen von der Benutzerschnittstelle an den Resource-Manager weiter. Transaktions-Manager stammen von Drittherstellern wie TUXEDO. In diesem Zusammenhang entspricht der Resource-Manager dem Datenbankserver. Bild 12-5 zeigt eine Konfiguration, in der Transaktionsverarbeitung verwendet wird.

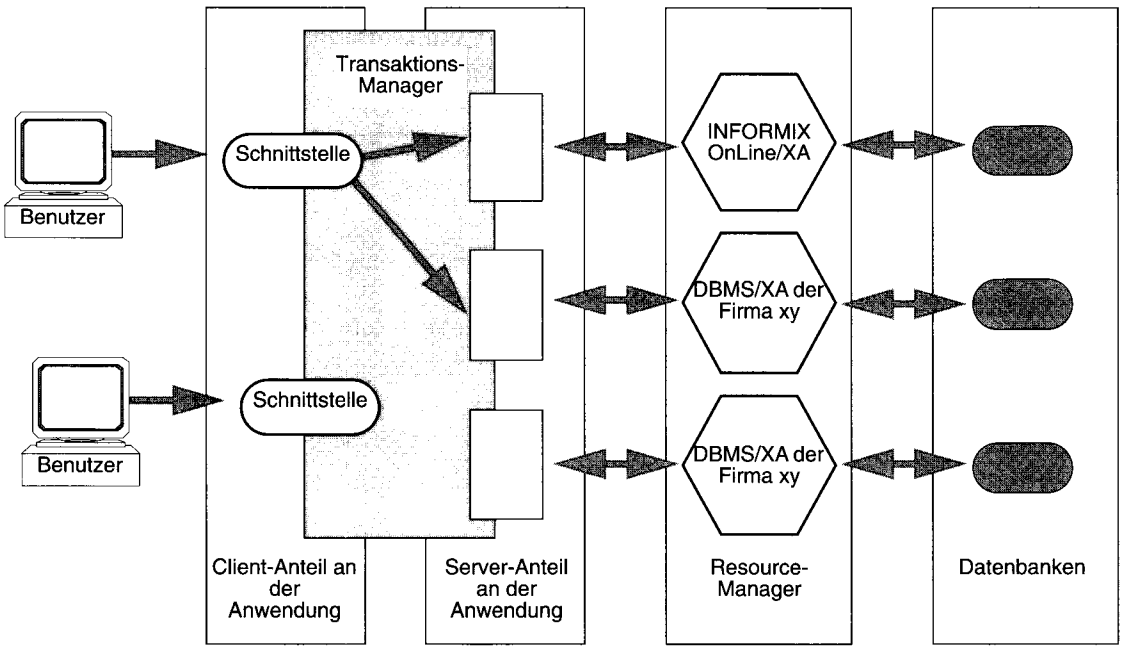



*Bild 12-5* ***Diese Konfiguration arbeitet mit einem Transaktionsmanager***

# Verbindungsaufbau in einem UNIX-Netz

Wenn Anwendungen und Daten auf verschiedenen Rechnern liegen, entstehen sofort zwei Fragen: Welche Verbindung kann man aufbauen? Wie bringt man seinem Anwendungsprogramm bei, die Daten zu finden, die auf einem anderen Rechner liegen?

Tatsächlich macht es keinen Unterschied, ob Sie ein Anwendungsprogramm an einen lokalen Datenbankserver oder über das Netz an einen fernen (remote) Server anschließen. Sie müssen dabei nur die folgenden Teile bedenken:

- Umgebungsvariablen
- Verbindungsinformation
- Verbindungsanweisungen

Dieses Kapitel faßt zusammen, wie Verbindungen zwischen Anwendungen und Datenbankservern bei Version 6.0 aufgebaut werden. Ausführliche Anweisungen zum Aufbau von lokalen Verbindungen und der Netzwerk-Ver-

bindungen für die Version 6.0 und für Vorgängerversionen erhalten Sie in den Handbüchern *INFORMIX-OnLine Dynamic Server Administrator's Guide* und *INFORMIX-SE Administrator's Guide*.

## Beispiel einer Client-/Server-Verbindung

Bild 12-6 zeigt sowohl lokale Verbindungen als auch Netzwerk-Verbindungen von Produkten der Version 6.0. Alle hier gezeigten Verbindungen sind möglich, allerdings nicht gleichzeitig. Client X kann die Verbindungen eins, zwei und drei aufbauen, während Client Y die Verbindungen vier und fünf eingehen kann.

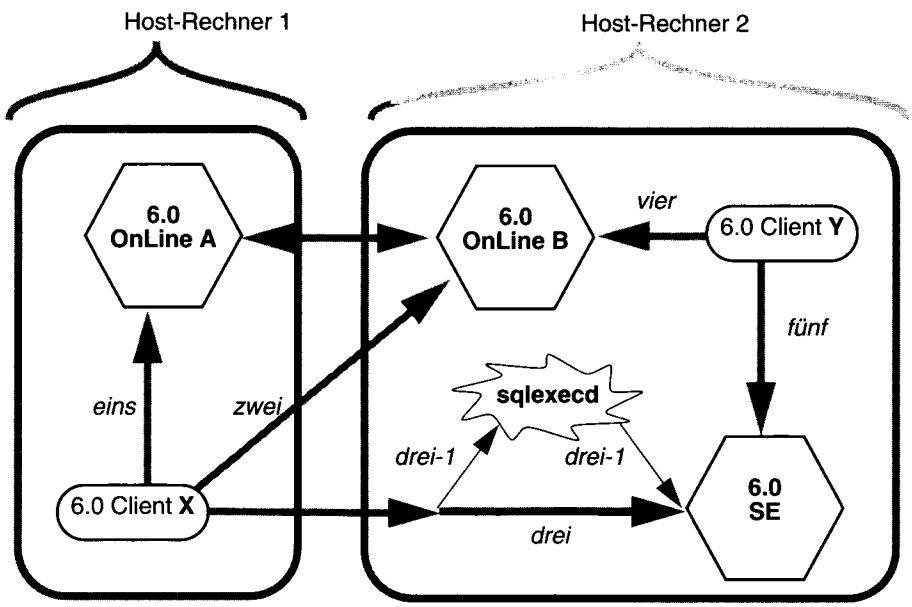



*Bild 12-6* ***Produkte der Version 6.0 lokal und im Netzverbund***

Sie können einen Client direkt an einen Datenbankserver mit **INFORMIX-OnLine Dynamic Server** anschließen, wenn Sie entweder Shared Memory (IPC) oder eine Netzverbindung verwenden. Wie Sie sehen, können Datenbankserver **INFORMIX-OnLine Dynamic Server** direkt miteinander kommunizieren (Server A und Server B).

Clients können lokal mit einem **INFORMIX-SE** Datenbankserver Verbindung aufnehmen, wenn sie *Unnamed Pipes* verwenden. Wenn sie die Verbindung zu **INFORMIX-SE** stattdessen über ein Netz aufbauen, werden sie zuerst mit dem Dämon, **sqlexecd**, verbunden (Ein *Dämonprozeß* ist ein Prozeß, der im

Hintergrund abläuft und auf Anfragen von anderen Programmen wartet). Der Dämonprozeß sqlexecd wird vom Systemverwalter gestartet. Wenn ein Client eine Verbindung zu einem **INFORMIX-SE** Datenbankserver aufbauen will, bemerkt dies der Dämonprozeß und erstellt eine temporäre Verbindung (in der Abbildung als hellgraue Pfeile dargestellt). Diese temporäre Verbindung (z. B. drei-1) ermöglicht es dem Client und dem Datenbankserver, eine direkte Verbindung (drei) aufzubauen. Anschließend zieht sich der Dämonprozeß zurück.

# Umgebungsvariablen

Der Datenbankverwalter muß sicherstellen, daß für jeden Benutzer die korrekten Umgebungsvariablen gesetzt werden. Hier sehen Sie die wichtigsten Umgebungsvariablen unter **INFORMIX-OnLine Dynamic Server** und **INFORMIX-SE**:

- PATH
- INFORMIXDIR
- INFORMIXSERVER
- TERM
- DBPATH

Die Umgebungsvariable INFORMIXDIR muß den kompletten Pfad des Verzeichnisses beinhalten, in dem die Informix-Programmdateien liegen. Unter PATH muß der volle Pfadname des Verzeichnisses angegeben werden, in dem die ausführbaren Programme derDatenbankserver **INFORMIX-OnLine** und **INFORMIX-SE** liegen. Diese beiden Umgebungsvariablen müssen belegt werden. Nachdem der Datenbankverwalter sie einmal belegt hat, werden Sie sie in der Regel nicht verändern.

Unter INFORMIXSERVER geben Sie den Namen des standardmäßig verwendeten Datenbankservers an. Diese Umgebungsvariable muß belegt werden. Sie können den Inhalt dieser Variable ändern, wenn Sie das Anwendungsprogramm wechseln.

Die Umgebungsvariable TERM (oder TERMCAP und/oder INFORMIXTERM) ermöglicht es dem Client, Ihr Terminal zu erkennen und mit ihm zu kommunizieren. Diese Variablen sind system- ( terminal-)abhängig. Um sie zu setzen, benötigen Sie wahrscheinlich die Hilfe Ihres Systemverwalters.

Die Variable DBPATH ist optional. Falls ein Anwendungsprogramm nicht den vollen Pfad eines **INFORMIX-SE** Datenbankservers angibt, werden die Verzeichnisse vom Server durchsucht, die in DBPATH angegeben sind.Mit

der Variablen DBPATH geben Sie sowohl bei **INFORMIX-OnLine Dynamic Server** als auch bei **INFORMIX-SE** auch die Verzeichnisse an, in denen Masken, Listen und Anweisungsdateien liegen.

Die Umgebungsvariablen werden ausführlich im Kapitel 4 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen* beschrieben.

# Verbindungsinformation

Wenn der Datenbankverwalter einen Server unter **INFORMIX-OnLine Dynamic Server** oder unter **INFORMIX-SE** einrichtet, muß er mit dem UNIX-Systemverwalter zusammenarbeiten. Die Zusammenarbeit muß die Informationen klären, die für den Verbindungsaufbau zwischen Anwendungsprogramm und Datenbankserver angegeben werden müssen. Das Anwendungsprogramm gibt den Server an und die Informix Verbindungswerkzeuge bauen die Verbindung auf. Die Verbindung ist transparent. Der Anwender braucht nicht zu wissen, auf welchem Rechner der Datenbankserver liegt.

Die Verbindungsinformation liegt in der Datei **$INFORMIXDIR/etc/sqlhosts** sowie in den zwei UNIX-Systemdateien, **/etc/host**s und **etc/services**. Diese drei Dateien geben zusammen an, wo der Datenbankserver liegt sowie die Art der Verbindung (des Verbindungsprotokolles) (siehe Bild 12-7). Auf jedem Rechner muß jeder Datenbankserver, der vielleicht einmal von einem Anwendungsprogramm benötigt werden wird, einen Eintrag in der Datei **$INFORMIXDIR/etc/sqlhosts** sowie entsprechende Einträge in den Dateien **/etc/host**s und **etc/services** haben. Diese Dateien werden ausführlich in den Handbüchern *INFORMIX-OnLine Dynamic Server Administrator's Guide* und *INFORMIX-SE Administrator's Guide* beschrieben.

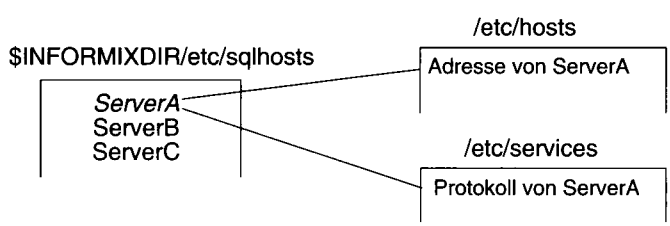



*Bild 12-7* ***Verbindungsinformation für Netzwerke***

## SQL-Verbindungsanweisungen

Bevor ein Anwendungsprogramm ablaufen kann, muß es eine Datenbank mit der Anweisung CONNECT TO oder DATABASE öffnen. Die Anweisung CONNECT TO wird in der Regel vorgezogen, weil sie sowohl dem ANSI- als auch dem X/Open-Standard entspricht. Dieser ist um eine einheitliche Terminologie in Netzwerkumgebungen bemüht.

Hier ein Beispiel einer CONNECT TO Anweisung:

```
CONNECT TO datenbankname@servername
```

Wenn der Name des Datenbankservers nicht in der Anweisung enthalten ist, wird das Anwendungsprogramm mit dem standardmäßig eingestellten Datenbankserver Verbindung aufnehmen, der in der Umgebungsvariablen INFORMIXSERVER festgelegt ist.

Wenn der Name der Datenbank fehlt, wird zwar eine Verbindung zum genannten DB-Server aufgebaut, aber keine Datenbank geöffnet. Bevor Sie eine Datenbank verwenden können, müssen Sie eine der Anweisungen DATABASE, CREATE DATABASE oder START DATABASE eingeben.

Beachten Sie, daß die Anweisung CONNECT TO keinerlei Information über die Lage des Datenbankservers enthält. Diese Information befindet sich in der Datei **sqlhosts** (Siehe Abschnitt "Verbindungsinformation" auf Seite 12-15).

# Tabellen ansprechen

Die Datenbank, die durch eine der Anweisungen CONNECT, DATABASE oder CREATE DATABASE geöffnet wird, wird als *aktuelle* Datenbank bezeichnet. Wenn Sie mit **INFORMIX-OnLine Dynamic Server** arbeiten, können Sie Tabellen abfragen, die sich nicht in der aktuellen Datenbank befinden. Wenn Sie dies tun, müssen Sie den Datenbanknamen als Teil des Tabellennames angeben, wie die folgende SELECT-Anweisung zeigt:

```
SELECT name, number FROM salesdb:contacts
```

Hier wird die Tabelle **contacts** in der Datenbank **salesdb** angesprochen. Diese Schreibweise können Sie auch in einem Join verwenden. Wenn Sie die Datenbanknamen explizit angeben können, kann dies mit erheblicher Schreibarbeit verbunden sein. Hier empfiehlt sich der Einsatz von Alias-Namen, wie das folgende Beispiel zeigt:

```
SELECT C.custname, S.phone
    FROM salesdb:contacts C, stores:customer S
    WHERE C.custname = S.company
```

Falls Sie eine Tabelle ansprechen wollen, die von einem anderen **INFORMIX-OnLine Dynamic Server** Datenbankserver verwaltet wird, müssen Sie dessen Namen mit angeben. Im folgenden Beispiel bezieht sich die SELECT-Anweisung auf die Tabelle **customer** in der Datenbank **masterdb**, die auf dem Datenbankserver **central** liegt:

```
SELECT O.order_num, C.fname, C.lname
    FROM masterdb@central:customer C, sales@boston:orders O
    WHERE C.customer_num = O.Customer_num
    INTO TEMP mycopy
```

In diesem Beispiel werden zwei Tabellen mit einem Join verbunden. Die verbundenen Datensätze werden in einer temporären Tabelle in der aktuellen Datenbank abgelegt. Die Tabellen befinden sich auf zwei verschiedenen Datenbankservern. Einer davon heißt **central**, der andere **boston**.

## Synonyme für Tabellennamen verwenden

Ein *Synonym* ist ein Name, den Sie für einen anderen Namen einsetzen können. Die Hauptverwendung der Anweisung CREATE SYNONYM besteht darin, den Zugang zu Tabellen zu erleichtern, die nicht in der aktuellen Datenbank liegen.

Im folgenden Beispiel werden Synonyme für Tabellennamen verwendet:

```
CREATE SYNONYM mcust FOR masterdb@central:customer;

CREATE SYNONYM bords FOR sales@boston:orders;

SELECT bords.order_num, mcust.fname, mcust.lname
    FROM mcust, bords
    WHERE mcust.customer_num = bords.Customer_num
    INTO TEMP mycopy
```

Die Anweisung CREATE SYNONYM speichert die Synonyme in der Systemtabelle **syssyntable** in der aktuellen Datenbank. Das Synonym kann in jeder Abfrage innerhalb dieser Datenbank verwendet werden.

Mit einem Synonym können Abfragen schneller geschrieben werden. Aber Synonyme können noch mehr: Mit ihrer Hilfe können Sie eine Tabelle in eine andere Datenbank verschieben, ja sogar auf einen anderen Rechner. Dabei ändert sich an Ihren Abfragen nichts.

Sie haben beispielsweise eine Reihe von Abfragen, die sich auf die Tabellen **customer** und **orders** beziehen. Die Abfragen sind in Programme, Masken und Listen eingebettet. Die Tabellen gehören zur Datenbank **stores6**, welche auf dem Datenbankserver **avignon** liegt.

Jetzt wird beschlossen, daß die gleichen Masken, Programme und Listen den den Benutzern eines anderen Rechners im Netz zur Verfügung gestellt werden sollen (Datenbankserver **nantes**). Diese Benutzer haben eine Datenbank mit der Tabelle **orders**, die die Bestellungen aus Ihrer Niederlassung beinhaltet. Aber zusätzlich benötigen Sie noch Zugang zur Tabelle **customer** auf dem Server **avignon**.

Für diese Benutzer ist die Tabelle **customer** extern, d. h. sie liegt auf einem fernen Datenbankserver. Bedeutet dies, daß Sie eigene Versionen von Programmen und Listen erstellen müssen, Versionen, in denen die Tabelle **customer** mit dem Namen des Datenbankservers qualifiziert wird? Einfacher ist es, ein Synonym in der Datenbank der Benutzer zu erstellen:

```
DATABASE stores6@nantes;
CREATE SYNONYM customer FOR stores6@avignon:customer
```

Wenn die gespeicherten Abfragen in Ihrer Datenbank ausgeführt werden, bezieht sich **customer** auf die aktuelle Tabelle. Wenn sie von einer anderen Datenbank aus aufgerufen werden, sorgt der Synonymname dafür, daß die externe Tabelle auf dem anderen Datenbankserver gefunden wird.

## Synonymketten

Wir setzen das obige Beispiel fort und nehmen an, daß ein neuer Rechner namens **db_crunch** an das Netz angeschlossen werden soll. Die Tabelle **customer** sowie einige andere Tabellen werden auf ihn verlagert, um **avignon** zu entlasten. Sie können die Tabellen auf dem neuen Datenbankserver leicht neu aufbauen. Aber wie leiten Sie alle Anfragen auf diese Tabellen zum neuen Server um? Eine Möglichkeit besteht darin, mit einem Synonym zu arbeiten:

```
DATABASE stores6@avignon EXCLUSIVE;
RENAME TABLE customer TO old_cust;
CREATE SYNONYM customer FOR stores6@db_crunch:customer;
CLOSE DATABASE
```

Wenn Sie eine Abfrage innerhalb von **stores6@avignon** ausführen, wird eine Anfage auf die Tabelle **customer** dank des Synonyms auf den neuen Rechner umgeleitet. Dies gilt auch für Abfragen, die vom Datenbankserver **nantes** aus gestartet werden. Das Synonym in der Datenbank **stores6@nantes** wird nach wie vor Anfragen zur Tabelle **customer** in der Datenbank **stores6@avignon** umleiten; dort sendet daß neue Synonym die Anfrage weiter zur Datenbank **stores6@db_crunch**.

Synonymketten sind dann sinnvoll, wenn Sie wie in diesem Beispiel alle Zugriffe auf eine Tabelle in einem Schritt umleiten wollen. Trotzdem sollten Sie so bald wie möglich die Datenbanken aller Benutzer aktualisieren, so daß Ihre Synonyme *direkt* auf die gewünschte Tabelle zeigen. Falls nämlich ein Rechner in der Kette ausfällt, funktioniert der Zugriff auf die Tabelle nicht mehr.

Sie können mit einem Anwendungsprogramm auf einer lokalen Datenbank arbeiten und später mit dem gleichen Programm auf einer Datenbank auf einem entfernten Rechner. In beiden Fällen läuft das Programm gleich (obwohl es möglicherweise langsamer wird, wenn es über das Netz auf die ferne Datenbank zugreift). Solange die Struktur der Datenbanken gleich ist, erkennt das Anwendungsprogramm keinen Unterschied zwischen einer lokalen und einer fernen Datenbank.

# Datensicherheit im Netz

Dieser Abschnitt gibt Ihnen einen Überblick über die Möglichkeiten, im Netzbetrieb Datensicherheit zu gewährleisten. Kapitel 4 dieses Handbuches beschäftigt sich ebenfalls mit diesem Thema.

## Datensicherheit unter INFORMIX-SE

**INFORMIX-SE** Datenbanken verwenden die UNIX-Dateistrukturen. Daher können Sie mit den normalen Betriebssystembefehlen Sicherungen Ihrer Datenbank machen. Im Netz ist es möglich, die Sicherungen an einem entfernt liegenden Gerät zu erstellen.

## Datensicherheit unter INFORMIX OnLine Dynamic Server

Datenreplikation bedeutet, daß bestimmte Informationen auf mehreren, verschiedenen Servern vorhanden ist. **INFORMIX-OnLine Dynamic Server** ermöglicht dies, indem es zwei verschiedene Rechner im Netz verwendet. Jeder Rechner hat einen **OnLine** Datenbankserver und identische Datenbanken. Ein Datenbankserver wird als primär bezeichnet, der andere als sekundär. Die Daten werden immer zuerst auf den primären Datenbankserver geschrieben, und dann auf den sekundären Datenbankserver übertragen.

Anwendungsprogramme können Daten von beiden Servern *lesen*. So hat der sekundäre Server eine doppelte Funktion: Er sorgt für Datensicherheit und er verbessert die Performance für Benutzer, die nur Daten lesen, aber nicht schreiben müssen.

Falls der primäre Datenbankserver (Server A) aus irgendeinem Grund ausfällt, wird der sekundäre Server (Server B) unabhängig weiterarbeiten. Benutzer, die normalerweise auf Server A arbeiten würden, werden auf Server B umgeleitet. Während Server A repariert wird, können alle Benutzer praktisch ohne Verzögerung weiterarbeiten.

Datenreplikation bietet ein hohes Maß an Datensicherheit, ist aber teuer. Es muß Speicherplatz für zwei komplette Kopien der Daten bezahlt werden. Der Transfer der Daten kostet Rechenzeit. Der Datenbankverwalter muß sich um beide Server kümmern.

### Archivierung

**INFORMIX-OnLine Dynamic Server** stellt Ihnen spezielle Werkzeuge zur Archivierung zur Verfügung. Dies kann lokal geschehen oder über das Netz. Die archivierten Bänder sollten aus Sicherheitsgründen in einem anderen Raum aufbewahrt werden als der Rechner steht.

## Datenintegrität bei verteilten Daten

Unter **INFORMIX-OnLine Dynamic Server** ist es möglich, Daten in verschiedenen Datenbanken auf verschiedenen Servern zu aktualisieren. Beispielsweise ist es möglich, daß *eine* Bestellung dazu führt, daß Datenbanken auf verschiedenen Servern aktualisiert werden müssen. Damit die Integrität der Daten gewährleistet ist, muß entweder die Information in allen oder in keiner Datenbank aktualisiert werden.

### Zwei-Phasen-Commit

Als *Zwei-Phasen-Commit* wird ein Protokoll bezeichnet, daß die Aktionen kontrolliert, die innerhalb einer Transaktion auf mehreren Datenbankservern ausgeführt werden. Anders als die Datenreplikation und die Werkzeuge zur Archivierung erzeugt das Zwei-Phasen-Commit **keine** zwei Kopien der Daten. Es gewährleistet die Gültigkeit **jeweils einer** Transaktion, die auf verschiedenen Servern abläuft (mit oder ohne Netz). Da es hier jeweils nur um eine Transaktion geht, wird der Zwei-Phasen-Commit häufig nicht im Zusammenhang mit Datensicherheit genannt.

Eine Transaktion über mehrere Datenbankserver wird als globale Transaktion bezeichnet. Das Zwei-Phasen-Commit ist die natürliche Erweiterung der Transaktionen, wie sie im Abschnitt "Die Aktualisierung wird unterbrochen" auf Seite 4-24 beschrieben ist. Das Handbuch *INFORMIX-OnLine Dynamic Server Administrator's Guide* erläutert das Zwei-Phasen-Commit im Detail.

Das Zwei-Phasen-Commit setzt ein, sobald ein Benutzerprozeß abgeschlossen ist und eine Bestätigung der globalen Transaktion verlangt:

**Phase 1**: Der aktuelle Datenbankserver fragt bei allen beteiligten Servern nach, ob sie ihre lokalen Transaktionen bestätigen können. Jeder Server antwortet mit „Ja" oder „Nein".

**Phase 2**: Falls alle Antworten positiv ausfallen, weist der aktuelle Datenbankserver alle anderen Server an, ihre Transaktionen zu bestätigen. Damit ist die globale Transaktion beendet. Falls auch nur ein Server eine negative Antwort gibt oder nicht antwortet, müssen alle Server ihre lokalen Transaktionen zurücksetzen.

# Zusammenfassung

Ein Netzwerk ermöglicht es, daß ein Anwendungsprogramm auf einem Rechner liegt, während der Datenbankserver in einem anderen Rechner installiert ist. So werden verteilte Verarbeitung und ein verteilter Datenbankzugang möglich.Es gibt zahlreiche Kombinationen aus Netzwerksoftware, Betriebssystemen und Datenbankservern.

Kapitel

# Datenbankserver-Abfragen optimieren

13

# Kapitelüberblick

Wie lange darf eine Abfrage dauern? Wie oft sollte der Rechner auf die Platte zugreifen, während eine Abfrage ausgeführt wird? Bei vielen Abfragen ist dies unerheblich, solange der Computer die Daten schneller findet als dies ein Mensch tun kann. Aber manche Abfragen müssen in einer begrenzten Zeit durchgeführt werden oder können nur eine begrenzte Rechnerkapazität verwenden.

Dieses Kapitel zeigt Techniken auf, wie man Abfragen effizienter gestaltet. *Es wird vorausgesetzt, daß Sie mit einer existierenden Datenbank arbeiten* und die Struktur der Tabellen nicht verändern können. (In den Kapiteln 8 bis 11 dieses Handbuchs werden Techniken erläutert, wie man eine neue Datenbank im Hinblick auf Zuverlässigkeit und Performance entwirft.)

Dieses Kapitel enthält die folgenden Themen:

- Eine allgemeine Erörterung über Techniken, Software zu optimieren. Hierbei werden all die Punkte hervorgehoben, die Sie sich anschauen sollten, bevor Sie eine SQL-Anweisung verändern.
- Eine Beschreibung des *Optimierers*. Der Optimierer ist ein Teil des Datenbankservers; er entscheidet, wie eine Abfrage durchgeführt wird. Wenn Sie wissen, wie der Optimierer einen Abfrageplan gestaltet, können Sie Abfragen effektiver gestalten.
- Eine Erläuterung der Vorgänge, die während einer Abfrage Zeit benötigen, so daß Sie besser zwischen langsamen und schnellen Operationen wählen können.
- Eine Auswahl von Techniken, die dem Optimierer dabei helfen, den schnellsten Weg für die Durchführung einer Abfrage zu wählen.

Dieses Kapitel konzentriert sich auf die Performance bei SQL- Anweisungen. Performance-Probleme können aber auch in anderen Programmteilen entstehen, in denen SQL-Anweisungen eingebettet sind. Zwei Bücher, die allgemeine Performance-Themen behandeln, sind *The Elements of Programming Style* von Kernighan und Ritchie (McGraw-Hill 1978) und *Writing Efficient*

*Programs* von Jon Louis Bentley (Prentice-Hall 1982). Viele Ratschläge von Bentley können auf SQL-Anweisungen und Datenbank-Design angewendet werden.

# Optimierungs-Techniken

Die Optimierung der SQL-Anweisungen sollten Sie als Teil eines größeren Systems betrachten. Das System besteht aus folgenden Komponenten:

- Einem oder mehreren *Programmen*

  Gespeicherte Abfragen, compilierten Bildschirmmasken, Listen und Programme, die in einer oder mehreren Sprachen geschrieben wurden und in die SQL-Anweisungen eingebettet sind.

- Einer oder mehreren *gespeicherten Prozeduren*

  Die compilierten Prozeduren, die aus SQL- und SPL- (Structured Procedure Language) Anweisungen bestehen; diese Anweisungen sind in ausführbarer Form in der Datenbank gespeichert.

- Einem oder mehreren *Rechner*

  Rechner, die die Programme und die Datenbanken speichern.

- Einem oder mehreren *Datenbankverwaltern*

  Die Personen, die für die Programmpflege verantwortlich sind.

- Einen oder mehreren *Benutzern*

  Die Personen, die mit dem Rechner arbeiten.

- Einen oder mehreren *Organisationsbereichen*

  Die Gruppen, denen die Rechner gehören und die entscheiden, welche Arbeit getan werden muß.

Möglicherweise arbeiten Sie in einer großen Firma, in der diese Komponenten voneinander getrennt sind. Oder Sie sind Besitzer, Datenbankverwalter und alleiniger Benutzer eines Rechners. In jedem Fall ist es wichtig, zwei Aspekte über das System als Ganzes zu erkennen. Erstens, das Ziel des Systems ist es, seinen *Benutzern* nützlich zu sein. Zweitens, SQL-Anweisungen sind nur ein kleiner Teil des Systems. Oft können die effektivsten Verbesserungen an anderen Teilen des Systems vorgenommen werden.

Die folgenden Abschnitte skizzieren ein allgemeines Verfahren, mit dem man jedes Performance-Problem des Rechners analysieren kann. Gehen Sie nach diesem Verfahren vor, damit Sie mögliche Lösungen nicht übersehen. Zu den Lösungen gehören auch nicht-technische Möglichkeiten, die manchmal die beste Antwort auf Performance-Probleme liefern.

## Das Problem prüfen

Bevor Sie mit dem Optimieren der SQL-Anweisungen beginnen, sollten Sie sicher sein, daß das Problem am Programm liegt. Wie viel effektiver werden die Benutzer arbeiten, wenn Sie das Programm schneller machen? Wenn die Antwort "nicht viel" lautet, sollten Sie die Lösung an einer anderen Stelle suchen.

## Das gesamte System betrachten

Betrachten Sie das ganze System mit den Programmen, Computern und Benutzern innerhalb eines Organisationsbereichs. Sie finden vielleicht durch Veränderung der Zeitplanung oder der Speicherverwaltung eine bessere Lösung. Falls Sie **INFORMIX-OnLine Dynamic Server** arbeiten, wollen Sie Ihr System vermutlich einer Feinabstimmung unterziehen. Eine problematische Operation würde vielleicht schneller laufen, wenn sie zu einer anderen Zeit, auf einem anderen Rechner oder zusammen mit anderen Aufträgen ausgeführt würde. Sie sollten diese Möglichkeiten zunächst einmal in Betracht ziehen, bevor Sie sich für die beste Vorgehensweise entscheiden.

Weitere Informationen zur Feinabstimmung eines **INFORMIX-OnLine** Systemes erhalten Sie im Handbuch **INFORMIX-OnLine Dynamic Server** *Administrators's Guide.*

## Die Anwendung verstehen

Versuchen Sie, die fragliche Anwendung als ein Ganzes zu verstehen. Eine Datenbankanwendung besteht gewöhnlich aus vielen Teilen, wie gespeicherten Prozeduren, Bildschirmmasken, Listenprogrammen und Programmen. Stellen Sie sich folgende Fragen:

- *Was* soll getan werden?

  Sie entdecken vielleicht redundante Schritte, die entfernt werden können. Selbst wenn jeder Schritt oder jede Handlung wichtig ist, wird es für Sie hilfreich sein, all diese Schritte und Handlungen und deren Reihenfolge zu kennen.

- *Warum* wird es getan?

  Besonders in alten Anwendungen werden Sie Programmteile finden, die keinen Zweck erfüllen; z. B. könnte vor einiger Zeit eine Operation zur Fehlersuche eingebaut worden sein, die nicht wieder entfernt wurde.

- *Für wen* wird es getan?

  Überprüfen Sie, ob wirklich alle Ausgaben der Anwendung benötigt werden.

- *Wo* ist der langsame Teil?

  Kristallisieren Sie so genau wie möglich diejenigen Teile heraus, die zu langsam sind.

# Die Anwendung messen

Bei Performance-Problemen müssen Sie zunächst einmal die Performance messen. Sie müssen einen Weg finden, um wiederholbare Messungen an den langsamen Teilen der Anwendung vorzunehmen. Aus folgenden Gründen ist dies äußerst wichtig:

- Ohne Zahlen können Sie den Benutzern oder den Verantwortlichen das Problem nicht präzise schildern.

  Um das Problem schildern zu können, müssen Sie mit Messungen Ergebnisse erzielen wie z. B. "Die Liste läuft 2 Stunden 38 Minuten" oder "Ein durchschnittlicher Update dauert 13 Sekunden, während der Hauptbelastungszeit dauert er dagegen bis zu 49 Sekunden".

- Ohne Zahlen können Sie keine sinnvollen Ziele setzen.

  Nur wenn Sie Messungen durchführen, können Sie eine Übereinstimmung mit in Zahlen ausgedrückten Performance-Zielen erhalten.

- Ohne Zahlen können Sie Ihren Fortschritt nicht messen.

  Sie brauchen Messungen, um kleine Verbesserungen herauszufinden und um zwischen alternativen Lösungen zu wählen.

## Manuelle Zeitmessung

Sie können mit einer einfachen Stoppuhr wiederholbare Messungen durchführen. Manuelle Zeitmessung ist dann sinnvoll, wenn Sie nur ein paar Ereignisse messen, die mindestens mehrere Sekunden dauern. (Wenn es sich um sehr kurze Zeitintervalle handelt, sollten Sie mit wiederholten Messungen arbeiten, damit der Messfehler durch die Mittelwertsbildungen möglichst klein gehalten wird.)

## Zeitangaben von Betriebssystemkommandos

Ihr Betriebssystem verfügt sehr wahrscheinlich über ein Kommando, das die Zeit anzeigt. Sie können die Operation, für die Sie den Zeitbedarf ermitteln wollen, zwischen zwei Zeitkommandos setzen.

### Zeitangaben von der Programmiersprache

Die meisten Programmiersprachen verfügen über Bibliotheksfunktionen für die Tageszeit. Wenn Sie Zugriff auf den Quellcode haben, können Sie Anweisungen einfügen, um die Zeit für bestimmte Handlungen zu messen. Wenn eine Anwendung z. B. mit **INFORMIX-4GL** geschrieben wurde, können Sie die Funktion CURRENT verwenden, um die aktuelle Zeit als DATETIME-Wert zu erhalten. Ein **4GL**-Programm kann mit einem Code, der dem folgenden Fragment ähnlich ist, eine automatische Zeitmessung durchführen:

```
DEFINE start_time DATETIME HOUR TO FRACTION(2),
       elapsed INTERVAL MINUTE(4) TO FRACTION(2)
LET start_time = EXTEND(CURRENT,HOUR TO FRACTION(2))

       { -- hier wird die zu messende Anweisung eingefügt -- }

LET elapsed = EXTEND(CURRENT,HOUR TO FRACTION(2))-start_time
DISPLAY ''Elapsed time was '',elapsed
```

In einem Multiprogramming-System oder einer Netzwerkumgebung, in der verfügbare Resourcen von mehreren Prozessen geteilt werden, entspricht die verstrichene Zeit nicht immer der Ausführungszeit. Die meisten C-Bibliotheken enthalten eine Funktion, die die CPU-Zeit eines Programms zurückliefert. Die C-Funktionen können von **4GL**-Programmen und **ACE**-Listen aufgerufen werden.

## Störfunktionen ermitteln

Bei den meisten Programmen benötigt nur ein sehr kleiner Teil des Codes (normalerweise 20% oder weniger) den Hauptteil der Programmausführungszeit (normalerweise 80% oder mehr). Die sogenannte *80/20 Regel*, wie sie genannt wird, stimmt in den meisten Fällen. Häufig sind die Proportionen sogar noch größer. Nachdem Sie einen Zeitmeß-Mechanismus eingerichtet haben, verwenden Sie diesen, um die *kritischen Punkte* in der Anwendung zu bestimmen. Die kritischen Punkte sind die Anweisungen, die am meisten Zeit benötigen. Zusätzlich verfügen viele Computersysteme über Profiling-Tools, die bei der Suche nach kritischen Punkten helfen können.

## Für alles offen bleiben

Bei eingebettetem SQL, bei dem eine Anweisung tausende von Plattenzugriffen auslösen kann, ist es sehr wahrscheinlich, daß die störenden 20% des Programmcodes SQL enthalten. Jedoch, *dies ist keineswegs sicher.* Gehen Sie

unvoreingenommen an die Untersuchung des Programms. Es kann sonst leicht passieren, daß Sie den Programmteil übersehen, der am meisten Zeit verbraucht.

Wenn die langsamen Operationen nicht auf SQL zurückzuführen sind, schlagen Sie in den Büchern nach, die auf Seite 13-3 angeführt sind. Wenn allerdings die SQL-Anweisungen die Performance-Probleme verursachen, müssen Sie den Abfrage-Optimierers verstehen.

# Der Optimierer

Der *Optimierer* als ein Bestandteil eines Datenbankservers entscheidet, wie eine Abfrage durchgeführt wird. Seine wichtigste Aufgabe besteht darin, die Reihenfolge festzulegen, in der jeder Satz der Tabelle durchsucht wird. Damit der Optimierer diese Entscheidung treffen kann, muß er sich für die effektivste Art entscheiden, um auf jede Tabelle zuzugreifen. Mögliche Zugriffsarten auf Tabellen sind sequentielles Durchsuchen der Datensätze, Durchsuchen anhand eines existierenden Index, Durchsuchen mit Hilfe eines temporären Index (falls mit verknüften Tabellen gearbeitet wird), der für diesen Zweck erstellt wird oder Durchsuchen anhand einer Sortierung. Weiterhin muß der Optimierer abschätzen, wie viele Datensätze jede Tabelle zum Endergebnis beisteuert.

## Arbeitsweise des Optimierers

Der Optimierer formuliert alle möglichen Abfragepläne. Für jeden Plan schätzt er die Anzahl der zu verarbeitenden Datensätze, der zu lesenden Pages und der benötigten Netzzugriffe. Ausgewählt wird der Plan mit dem niedrigsten Aufwand. Im folgenden werden die Schritte beschrieben, die der Optimierer bei der Bewertung der verschiedenen Abfragepläne unternimmt. Ziel ist es, daß Sie eine grundlegende Vorstellung von der Funktionsweise des Optimierers bekommen, damit Sie Ihre Abfragen dementsprechend gestalten können.

## Wie der Optimierer Tabellen verknüpft

Der Optimierer sucht nach allen verknüpften Tabellen, unabhängig davon, ob die Verknüpfungen Filter (also WHERE-Klauseln) oder Indizes beinhalten, und entfernt redundante Verknüpfungen. Stellen Sie sich folgendes Beispiel vor:

```
SELECT ... FROM customer c, items i, orders o
    WHERE c.customer_num = o.customer_num
    AND o.order_num = i.order_num
```

Der Optimierer bildet alle möglichen Tabellenpaare und entfernt dann die redundanten Paare. Die am wenigsten aufwendigen Verknüpfungen bleiben übrig, wie das folgende Beispiel zeigt:

```
(c,i), (c,o), (i,c), (i,o), (o,c), (o,i)
```

## Input zur Verfügung stellen

Der Optimierer kann nur dann erfolgreich arbeiten, wenn seine Schätzungen genau sind (Die Schätzungen müssen nicht nach absoluten Maßstäben genau sein, sondern relativ, damit bessere Pläne niedrigere Schätzwerte erhalten als schlechtere Pläne.). Der Optimierer verfügt jedoch nur über begrenzte Informationen. Um seine Arbeit auf einen kleinen Bruchteil der Ausführungszeit zu beschränken, muß er mit den Informationen der Systemtabellen und einigen anderen Informationen auskommen. Der Optimierer hat z. B. keine Zeit, die Operation SELECT COUNT(*) durchzuführen, um die genaue Anzahl der Datensätze einer Tabelle zu ermitteln.

Die Informationen, die für den Optimierer verfügbar sind, sind in den Systemtabellen enthalten (Informationen zu den Systemtabellen erhalten Sie in Kapitel 2 des Handbuchs *SQL-Sprachbeschreibung, Nachschlagen.*). Jeder Datenbankserver verfügt über folgende Informationen:

- Die Anzahl der Datensätze in einer Tabelle (auf dem Stand der zuletzt ausgeführten UPDATE STATISTICS-Anweisung)
- Ob eine Spalte eindeutig sein soll
- Die Indizes, die für eine Tabelle existieren; welche Spalten die Indizes umfassen, ob sie auf- oder absteigend sind und ob sie Cluster-Indizes sind
- Information über die Datenverteilung in gefilterten Spalten

Systemtabellen von **INFORMIX-OnLine Dynamic Server** verfügen über die folgenden zusätzlichen Informationen:

- Die Anzahl der Platten-Pages, die von Datensätzen belegt sind
- Die Tiefe des Index der B+ Baumstruktur (ein Meßwert über den Zeitbedarf zur Erzeugung eines aufsteigenden Index)
- Die Anzahl der Platten-Pages, die von Index-Einträgen belegt sind
- Die Anzahl der eindeutigen Einträge in einen Index (geteilt durch die Anzahl der Datensätze; dies gibt eine Vorstellung darüber, wie viele Datensätze zu einem gegebenen Schlüssel passen könnten)
- Die zweitgrößten und zweitkleinsten Schlüsselwerte einer indizierten Spalte

Es werden nur die zweitgrößten und zweitkleinsten Schlüsselwerte vermerkt, da die größten und kleinsten Werte spezielle Werte sein könnten, die die Überschreitung eines Wertebereichs anzeigen. Der Datenbankserver setzt voraus, daß die Schlüsselwerte zwischen dem zweitgrößten und zweitkleinsten Wert verteilt sind. Nur die ersten vier Zeichen dieser Schlüssel werden gespeichert. Falls über die Index-Spalten Information zur Datenverteilung vorhanden ist, werden eher jedoch diese Verteilungen als die Schlüsselwerte verwendet, um die Anzahl der Sätze zu schätzen, die einen Filter passieren.

## Filter beurteilen

Der Optimierer untersucht zuerst die Ausdrücke der WHERE-Klausel nach Filtern. Der Optimierer schätzt die *Selektivität* eines jeden Filters. Die Selektivität wird mit einer Zahl zwischen 0 und 1 angegeben. Dies zeigt den Bruchteil der Datensätze an, von denen der Optimierer annimmt, daß sie den Filter passieren. Einem sehr selektiven Filter, den nur sehr wenig Datensätze passieren, wird eine Selektivität nahe 0 zugewiesen. Einem Filter, der die meisten Datensätze passieren läßt, wird eine Selektivität nahe 1 zugewiesen. Einzelheiten hierzu finden Sie im Abschnitt "Zuordnung der Filter-Selektivität" auf Seite 13-33.

Der Optimierer vermerkt auch andere Informationen über die Abfrage, wie z. B. die, welche Filter-Spalten indiziert sind und ob *ein* Index genügt. Wenn nur eine indizierte Spalte ausgewählt wurde, dann ist es schneller, nur die Index-Pages und keine Datensätze zu lesen.

Weiterhin vermerkt der Optimierer, ob ein Index verwendet werden kann, um einen Filter auszuwerten. Eine indizierte Spalte ist in diesem Sinne eine Spalte, zu der ein Einzel-Index gehört, oder die als erste in einem zusammengesetzten Index genannt wird. Es gibt mehrere Fälle, die man berücksichtigen muß:

- Wenn eine indizierte Spalte mit einem Literal, einer Hostvariablen oder einer nicht korrelierten Unterabfrage verglichen wird, dann kann der Datenbankserver nach übereinstimmenden Werten im Index suchen, anstatt die Datensätze zu lesen.
- Wenn eine indizierte Spalte mit einer Spalte einer anderen Tabelle verglichen wird (Join-Ausdruck), dann kann der Datenbankserver den Index verwenden, um übereinstimmende Werte zu finden. Dies setzt voraus, daß der Abfrageplan zuerst Datensätze aus einer anderen Tabelle liest. Der folgende Join-Ausdruck ist ein Beispiel:

```
WHERE customer.customer_num = orders.customer_num
```

  Wenn die Datensätze der Tabelle **customer** zuerst gelesen werden, dann können die Werte der Spalte **customer_num** anhand des Index auf **orders.customer_num** gesucht werden.
- Ob ein Index bei der Bearbeitung einer ORDER BY-Klausel verwendet werden kann. Wenn alle Spalten einer Klausel in einem Index in derselben Reihenfolge enthalten sind, dann kann der Datenbankserver den Index verwenden, um die Datensätze in dessen sortierter Reihenfolge zu lesen. Dadurch wird eine Sortierung vermieden.
- Ob ein Index bei der Bearbeitung einer GROUP BY-Klausel verwendet werden kann. Wenn alle Spalten der Klausel in einem Index in derselben Reihenfolge enthalten sind, dann kann der Datenbankserver anhand des Index Gruppen herauslesen, die über den gleichen Schlüssel verfügen. Dabei ist keine Sortierung erforderlich.

## Zugriffspfade auf Tabellen auswählen

Als nächstes wählt der Optimierer die Zugriffsart auf Tabellen aus, die er für die effektivste hält. Die Zugriffsart wird für jede Tabelle gewählt, die in der Abfrage genannt ist. Der Optimierer hat vier Auswahlmöglichkeiten:

- Die Datensätze einer Tabelle sequentiell lesen
- Einen der Indizes einer Tabelle lesen und die Datensätze lesen, auf die der Index zeigt,
- Einen temporären Index erzeugen und verwenden

- Einen Sort-Merge durchführen

Die Wahl zwischen einer der beiden ersten Möglichkeiten hängt hauptsächlich von der Angabe eines Filter-Ausdrucks ab. Wenn ein Filter für eine indizierte Spalte angegeben ist, dann wählt der Datenbankserver die Datensätze anhand des Index aus und liefert nur diese Datensätze.

Wenn kein Filter angegeben ist, dann muß der Datenbankserver auf jeden Fall alle Datensätze lesen. Normalerweise geht es schneller, einfach nur die Datensätze zu lesen, als die Pages eines Index und anschließend die Datensätze zu lesen. Wenn die Datensätze jedoch sortiert vorliegen sollen, dann kann es aber auch schneller sein, den Index zu lesen und die Datensätze dann anhand des Index in sortierter Reihenfolge zu lesen.

Der Optimierer könnte die dritte Möglichkeit – das Erzeugen eines temporären Index – in zwei Fällen auswählen. Wenn keine der beiden Tabellen einen Index in der Join-Spalte hat und die Tabellen groß genug sind, dannkann sich der Optimierer für diese Möglichkeit entscheiden. Der Grund dafür liegt darin, daß es schneller geht, einen Index für eine Tabelle aufzubauen als eine Tabelle sequentiell zu lesen und dabei nach jedem Datensatz der anderen Tabelle zu suchen. Ein temporärer Index kann auch dazu verwendet werden, die Datensätze zu sortieren oder in Gruppen zusammenzufassen (Eine Alternative hierfür ist, die ausgegebenen Datensätze in eine temporäre Tabelle zu schreiben und diese Tabelle zu sortieren.).

Bei der vierten Möglichkeit des Optimierers – Durchführung eines Sort-Merge – ist kein temporärer Index erforderlich. Wenn bei einem Join mindestens ein Filter ein Gleichheits-Operator ist, und wenn ein temporärer Index aufwendiger wäre, dann wird diese Möglichkeit automatisch durchgeführt.

## Einen Abfrageplan auswählen

Der Optimierer erstellt anhand der verfügbaren Informationen alle möglichen Abfragepläne zur Verknüpfung zweier Tabellen. Wenn mehr als zwei Tabellen verknüpft werden, dann verwendet der Optimierer den besten Plan zur Verknüpfung von zwei Tabellen, um aus diesen alle Pläne für die Verknüpfung von zwei Tabellen zu einer dritten, von drei Tabellen zu einer vierten usw. zu gestalten.

Der Optimierer führt an den vollständigen Join-Plänen abschließende Arbeiten aus. Wenn z. B. eine ORDER BY- oder GROUP BY-Klausel vorhanden ist und der Plan die Datensätze in unsortierter Reihenfolge ausgibt, dann fügt der Optimierer dem Plan eine Schätzung über die Zeitdauer einer Sortierung hinzu. Das Sortieren wird im Abschnitt "Zeitbedarf beim Sortieren" auf Seite 13-17 erörtert.

Schließlich wählt der Optimierer den Plan aus, der den geringsten Arbeitsaufwand verspricht. Er leitet ihn an den Hauptteil des Datenbankservers weiter, damit der Plan ausgeführt wird.

## Den Plan lesen

Die Wahl, die der Optimierer trifft, muß kein Geheimnis bleiben. Sie können genau ermitteln, welchen Plan er wählt. Bevor Sie eine Abfrage durchführen, führen Sie die Anweisung SET EXPLAIN ON aus. Beginnend mit der nächsten Abfrage schreibt der Optimierer eine Erläuterung zu seinem Abfrageplan in eine bestimmte Datei (der Name der Datei und deren Position hängen von dem verwendeten Betriebssystem ab). Eine typische Erläuterung des Optimierers wird in Bild 13-1 gezeigt.

Nachdem die Abfrage wiederholt wurde, zeigt der Optimierer seine Schätzung des Arbeitsaufwands (102 in Bild 13-1) an. Ein einzelner Plattenzugriff ist eine Einheit. Da dieser Abfrageplan am niedrigsten geschätzt wurde, wurde er den anderen vorgezogen.

```
QUERY:
------
SELECT C.customer_num, O.order_num, SUM (I.total_price)
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
        AND O.order_num = I.order_num
    GROUP BY C.customer_num, O.order_num;

Estimated Cost: 102
Estimated # of Rows Returned: 1
Temporary Files Required For: GROUP BY

1) pubs.o: SEQUENTIAL SCAN

2) pubs.c: INDEX PATH

 (1) Index Keys: customer_num (Key-Only)
 Lower Index Filter: pubs.c.customer_num = pubs.o.customer_num

3) pubs.i: INDEX PATH

 (1) Index Keys: order_num
 Lower Index Filter: pubs.i.order_num = pubs.o.order_num
```

*Bild 13-1* ***Typische Ausgabe, die vom Optimierer mit SET EXPLAIN ON erstellt wird***

Der Optimierer zeigt auch die geschätzte Anzahl der Datensätze, die die Abfrage zurückliefert. In Bild 13-1 schätzt der Optimierer fälschlicherweise 1. Die Schätzung ist deshalb falsch, weil der Optimierer schlecht ermitteln kann, wieviele Gruppen die GROUP BY-Klausel erstellt. Sie wissen, daß es für jeden Datensatz der Tabelle **orders** eine Gruppe gibt; der Optimierer kann es aber nicht wissen.

Im Hauptteil der Erläuterung listet der Optimierer auf, in welcher Reihenfolge auf die Tabellen zugegriffen wird. Weiterhin ist die Methode oder der Zugriffspfad aufgelistet, mit dem jede Tabelle gelesen wird. Die folgende Aufschlüsselung erklärt den Plan:

1. Die Tabelle **orders** wird zuerst gelesen. Da es für die Tabelle **orders** keinen Filter gibt, muß jeder ihrer Datensätze gelesen werden. Der „preiswerte-

ste" Zugang besteht also darin, die Tabelle in physikalischer Reihenfolge lesen zu lassen.

2. Für jeden Datensatz der Tabelle **orders** wird in der Tabelle **customer** nach übereinstimmenden Datensätzen gesucht. Die Suche wird anhand des Index auf die Spalte **customer_num** durchgeführt.

   Der Hinweis *Key-Only* bedeutet, daß nur der Index gelesen wird, da in der Ausgabe nur die Spalte **customer_num** verwendet wird; aus der Tabelle wird kein Datensatz gelesen.

3. Für jeden Datensatz der Tabelle **orders**, für den es eine übereinstimmende Kundennummer (**customer_num**) gibt, wird in der Tabelle **items** gesucht; hierbei wird der Index der Spalte **order_num** verwendet.

Es ist nicht immer offensichtlich, weshalb der Optimierer eine bestimmte Wahl getroffen hat. Wenn Sie die Pläne, die von mehreren Varianten einer Abfrage erstellt werden, vergleichen, dann können Sie die Logik des Optimierers teilweise nachvollziehen.

# Zeitbedarf einer Abfrage

Der Datenbankserver braucht bei der Ausführung einer Abfrage am meisten Zeit für die Durchführung der folgenden zwei Operationen: Daten von der Platte lesen und Spaltenwerte vergleichen. Von diesen beiden Operationen geht das Lesen der Daten bei weitem langsamer. Der folgende Abschnitt beschreibt, wofür der Datenbankserver die Zeit braucht. Der nächste Abschnitt erläutert, wie man Abfragen schneller macht.

## Aktivitäten im Speicher

Der Datenbankserver kann Daten nur im Speicher bearbeiten. Er muß einen Datensatz erst in den Speicher laden, bevor er den Satz mit einem Filter-Ausdruck überprüfen kann. Entsprechend muß der Server erst die Datensätze beider Tabellen in den Speicher laden, bevor er eine Join-Bedingung überprüfen kann. Der Datenbankserver erzeugt einen Ausgabesatz im Speicher, indem er die ausgewählten Spalten aus anderen Datensätzen zusammensetzt.

Die meisten dieser Aktivitäten gehen sehr schnell. Der Datenbankserver kann in einer Sekunde Hunderte oder sogar Tausende von Vergleichen durchführen; dies hängt von der Rechnerkapazität ab. Daraus folgt, daß die Zeit, die für Speicheroperationen gebraucht wird, in der Regel nur ein kleiner Teil der gesamten Ausführungszeit darstellt.

Zwei Speicheraktivitäten können jedoch beträchtliche Zeit in Anspruch nehmen. Die eine ist das Sortieren, das im Abschnitt "Zeitbedarf beim Sortieren" auf Seite 13-17 beschrieben ist. Die andere Aktivität führt Vergleiche durch, die LIKE und MATCHES verwenden; hierbei braucht besonders die Überprüfung nach "keinem oder mehreren" Zeichen am Anfang oder in der Mitte eines Wertes viel Zeit.

# Plattenzugriff verwalten

Einen Datensatz von der Platte zu lesen nimmt viel mehr Zeit in Anspruch, als einen Datensatz im Speicher zu prüfen. Das Hauptziel des Optimierers liegt darin, die Menge der Daten zu reduzieren, die von der Platte gelesen werden müssen. Der Optimierer kann aber nur die Pläne ausschließen, die offensichtlich am unproduktivsten sind.

## Platten-Pages

Der Datenbankserver arbeitet mit Einheiten des Plattenspeichers, die als *Pages* bezeichnet werden. Eine Page ist ein Block mit einer festgelegten Größe. Für alle Datenbanken, die von einem Datenbankserver verwaltet werden, wird die gleiche Größe verwendet. Die Indizes werden ebenfalls in Pageeinheiten gespeichert.

Die Größe einer Page hängt vom Datenbankserver ab. Bei **INFORMIX-OnLine Dynamic Server** wird die Page-Größe bei der Initialisierung von **OnLine** festgelegt. Normalerweise beträgt die Größe 2 KB (2.048 Byte), aber Sie können auch denjenigen, der **OnLine** installiert hat, fragen, welcher Wert gewählt wurde. Andere Informix Datenbankserver verwenden die Datenspeicherung des Betriebssystems; daraus ergibt sich, daß die Page-Größe gleich der Block-Größe ist, die vom Betriebssystem verwendet wird. Die übliche Größe ist 1 KB (1.024 Byte).

Es ist möglich, Tabellen so groß zu machen , daß ein Datensatz eine Page füllt (einige Datenbankserver erlauben es sogar, daß ein Satz größer als eine Page ist). Eine typische Platten-Page faßt 5 bis 50 Sätze. Die Größe eines Indexeintrages besteht aus der Breite des indizierten Feldes in Bytes plus einem Zeiger, der 4 Byte lang ist; daher enthält eine Index-Page normalerweise zwischen 50 und 500 Einträgen.

## Zeitbedarf beim Sortieren

Die Speicheraktivität ist proportional zu $c*w*n*\log_2(n)$, wobei gilt:

Beim Sortieren sind Speicher- und Plattenaktivitäten erforderlich.

| | |
|---|---|
| $c$ | Anzahl der Spalten, die sortiert werden; es repräsentiert den Zeitbedarf für die Entnahme von Spaltenwerten aus Sätzen und für deren Verkettung zu einem Sortierschlüssel. |
| $w$ | proportional zu der Größe des zusammengesetzten Sortierschlüssels, ausgedrückt in Byte; $w$ steht für den Zeitbedarf, um einen Sortierschlüssel zu kopieren oder zu vergleichen. Ein numerischer Wert für $w$ hängt stark von der verwendeten Hardware ab. |
| $n*\log_2(n)$ | Anzahl der Vergleiche, die bei der Sortierung einer Tabelle durchgeführt werden, die $n$ Sätze enthält. |

Die Plattenaktivität ist proportional zu $2*n*m$,wobei $n$ auch hier die Anzahl der Sätze ist. Der Faktor $m$ steht für die Anzahl der *Verbindungsstufen,* die für eine Sortierung erforderlich sind; dieser Faktor hängt von der Anzahl der Sortierschlüssel ab, die im Speicher gehalten werden können.

Wenn alle Schlüssel im Speicher gehalten werden können, dann ist $m$=1 und die Plattenaktivität ist proportional zu $2n$. Anders ausgedrückt, die Sätze werden gelesen, im Speicher sortiert und geschrieben. (Wenn ein Index erstellt wird, werden nur die Index-Pages geschrieben.)

Bei mittleren bis großen Tabellen werden die Sätze zu Einheiten sortiert, die in den Speicher passen. Anschließend werden die Einheiten *miteinander verbunden.* Wenn $m$=2 ist, werden die Sätze gelesen, sortiert und in Einheiten geschrieben. Dann werden die Einheiten erneut gelesen, *miteinander verbunden* und wieder geschrieben, woraus sich eine Plattenaktivität ergibt, die proportional zu $4n$ ist. Bei sehr langen Tabellen müssen die Einheiten in weitere Einheiten geteilt werden. Die Plattenaktivität ist dann proportional zu $6n$. Kurz gesagt, bei zunehmender Tabellengröße wächst die Plattenaktivität beim Sortieren schnell und ungleichmäßig von $2n$ über $4n$ auf $6n$. Die Tabellengröße, aus der sich diese Schritte ergeben, hängt von vielen Faktoren ab; hierbei können Sie nur einen (die Schlüsselgröße) kontrollieren.

Da der Faktor $n$ in beiden Ausdrücken dominiert, kann man den Zeitbedarf für das Sortieren am besten dadurch verringern, daß man weniger Sätze sortiert. Wenn dies nicht möglich ist, sollte man versuchen, nach weniger oder kleineren Spalten zu sortieren. Dies verringert nicht nur die Faktoren $c$ und $w$, sondern es verzögert auch die Ausführung der nächsten Verbindungsstufe.

Beim Sortieren werden Informationen temporär auf Platte geschrieben. Sie können Schreibzugriffe entweder in eine von **INFORMIX-OnLine Dynamic Server** verwalteten Dbspace oder in eine Datei des Betriebssystems lenken.

Für die Performance ist es meist sehr viel besser, mit einem Dbspace statt mit einer Datei zu arbeiten. Mit der Umgebungsvariable PSORT_DBTEMP geben Sie ein oder mehrere Dateiverzeichnisse an, wenn Sie die Informationen in Dateien speichern wollen. Ist PSORT_DBTEMP nicht belegt, wird die Information in den Dbspace geschrieben, der in der Umgebungsvariablen DBSPACETEMP gespeichert ist. Ist keine der genannten Umgebungsvariablen belegt, wird die Information in den Dbspace geschrieben, der über den Parameter DBSPACETEMP ONCONFIG angegeben ist. Falls weder die Umgebungsvariablen noch der Parameter gesetzt sind, wird die Information temporär in Dateien des Verzeichnisses /tmp abgelegt. Detailierte Information zu diesem Thema finden Sie in Kapitel 4 des Handbuchs *Informix-SQL, Nachschlagen.* Information zur Datei ONCONFIG finden Sie im Handbuch *INFORMIX-OnLine Dynamic Server Administrator's Guide.*

### Page-Puffer

Der Datenbankserver verfügt über eine Reihe von Speicherplätzen, in denen er Kopien der vor kurzem eingelesenen Pages aufbewahrt. Er tut dies in der Hoffnung, daß diese bald wieder benötigt werden. Falls dies zutrifft, dann muß der Datenbankserver diese nicht erneut von der Platte lesen.

Ähnlich wie die Größe einer Platten-Page hängt die Anzahl der Page-Puffer vom Datenbankserver und dem Betriebssystem ab.

## Zeitbedarf zum Lesen eines Satzes

Wenn ein Datenbankserver einen Datensatz überprüfen muß, der noch nicht im Speicher ist, dann muß er ihn von der Platte lesen. Er liest aber nicht genau einen Satz, sondern er liest die ganze Page, in der der Datensatz enthalten ist. (Wenn ein Datensatz länger als eine Page ist, dann liest er soviele Pages, wie nötig sind.) Die Zeit, um eine Page zu lesen, ist die zugrunde gelegte Arbeitseinheit, die der Optimierer für seine Berechnung verwendet. Die tatsächliche Zeit, die benötigt wird, um eine Page zu lesen, variiert und ist schwer vorherzusagen. Sie setzt sich aus den folgenden Faktoren zusammen:

| | |
|---|---|
| Pufferung | Die benötigte Page könnte sich bereits in einem Page-Puffer befinden. In diesem Fall ist der Zugriffsaufwand fast Null. |
| konkurrierende Zugriffe | Wenn mehr als eine Anwendung auf die Festplatte zugreift, dann kann sich die Antwort des Datenbankserver verzögern. |
| Zugriffszeit | Beim *Zugriff* ist die Platte am langsamsten; der Schreib-/Lesekopf muß in die Spur positioniert werden, die die Daten enthält. Die Zugriffszeit hängt ab von der Geschwindigkeit der Platte und der Position des Schreib-/Lesekopfes zu Beginn der Operation. Die Zugriffszeit bewegt sich zwischen Null und einem Bruchteil einer Sekunde. |
| Latenzzeit | Die Übertragung kann erst dann beginnen, wenn der-Schreib-/Lesekopf an den Anfang der Page positioniert ist. Diese *Latenzzeit* bzw. Verzögerungszeit durch Rotation hängt ab von der Geschwindigkeit der Platte und der Position der Platte zu Beginn der Operation. Die Latenzzeit kann von Null bis zu ein paar Tausendstelsekunden dauern. |

Der Zeitbedarf, um eine Page zu lesen, variiert von Mikrosekunden (bei einer Page, die im Puffer ist) über ein paar Tausendstelsekunden (wenn es keine konkurrierende Zugriffszeit gibt und der Schreib-/Lesekopf bereits richtig positioniert ist) bis hin zu Zehntelsekunden.

## Zeitbedarf bei sequentiellem Zugriff

Wenn der Datenbankserver die Datensätze in physikalischer Reihenfolge liest, sind die Zugriffszeiten auf die Platten am niedrigsten. Wenn der erste Datensatz einer Page angefordert wird, dann wird die ganze Page gelesen. Nachfolgende Datensätze werden vom Puffer erst dann angefordert, wenn die ganze Page abgearbeitet wurde. Folglich wird eine Page nur einmal gelesen.

Ist der Datenbankserver das einzige Programm, das auf die Platte zugreift, dann sind die Zugriffszeiten entsprechend kurz. Die Pages aufeinanderfolgender Datensätze befinden sich normalerweise an aufeinanderfolgenden Positionen der Platte, so daß sich der Schreib-/Lesekopf von einer Page zur nächsten nur sehr wenig bewegen muß. Selbst wenn dies nicht der Fall ist, sind normalerweise Gruppen aufeinanderfolgender Pages zu Einheiten zusammengefaßt, so daß nur wenige zeitaufwendige Zugriffe erforderlich sind. Beispielsweise fordert **INFORMIX-OnLine Dynamic Server** Plattenbereich für eine Tabelle in *Extents* an. Extents dürfen auf der Platte getrennt voneinander sein, in einem Extent liegen die Pages aber näher beieinander.

Die Latenzzeit kann sogar dann am niedrigsten sein, wenn die Pages sequentiell gelesen werden. Dies hängt von der Hardware ab und von der Methode des Betriebssystems, falls Betriebssystem-Dateien verwendet werden. Die Platten sind normalerweise so eingerichtet, daß beim sequentiellen Lesen der Pages die Latenzzeit heruntergesetzt ist. Andererseits ist es möglich, eine Platte so einzurichten, daß zwischen jeder folgenden Page eine ganze Plattenumdrehung stattfinden muß. Dies verlangsamt den sequentiellen Zugriff erheblich.

## Zeitbedarf bei nicht sequentiellem Zugriff

Wenn die Datensätze einer Tabelle nicht in physikalischer Reihenfolge abgerufen werden, dann sind die Zugriffszeiten auf die Platten höher. In der Praxis sind Tabellen normalerweise viel größer als die Page-Puffer des Datenbankservers, so daß nur ein kleiner Teil der Pages einer Tabelle im Speicher gehalten werden kann. Wenn eine Tabelle nicht in sequentieller Reihenfolge gelesen wird, werden nur wenige Datensätze in den Page-Puffern gefunden. Normalerweise wird für jeden gesuchten Datensatz eine Platten-Page gelesen.

Da die Pages nicht- sequentiell von der Platte gelesen werden, treten normalerweise sowohl beim Zugriff als auch bei der Rotation Verzögerungen auf. Kurz gesagt, wenn eine Tabelle nicht-sequentiell gelesen wird, dann ist die Zugriffszeit auf die Platte viel höher als wenn die Tabelle sequentiell gelesen wird.

## Zeitbedarf eines Zugriffs über die Rowid

Die einfachste Form eines nicht-sequentiellen Zugriffs besteht darin, einen Datensatz anhand seines *Rowid*-Wertes auszuwählen (Die Verwendung der Rowid in einer SELECT-Anweisung ist in Kapitel 3, Seite 17 erläutert.). Ein Rowid-Wert gibt die physikalische Position eines Datensatzes und seiner Page an. Der Datenbankserver liest einfach nur die Page und hat dafür nur den bereits bekannten Zeitbedarf.

## Zeitbedarf eines indizierten Zugriffs

Es gibt einen weiteren Zeitbedarf, der mit dem Finden eines Datensatzes über einen Index zu tun hat: Der Index selbst ist auf der Platte gespeichert und seine Pages müssen in den Speicher geholt werden.

Der Datenbankserver verwendet Indizes auf zwei Arten. Eine Art ist, einen Datensatz anhand eines gegebenen Schlüsselwertes zu suchen. Diese Art der Suche wird verwendet, wenn Sie zwei Tabellen mit einer Anweisung wie der folgenden verknüpfen:

```
SELECT company, order_num
    FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
```

Eine Tabelle, wahrscheinlich **customer**, wird sequentiell gelesen. Der Wert der Spalte **customer_num** wird verwendet, um den Index aus der Spalte **customer_num** der Tabelle **orders** zu suchen. Wenn ein übereinstimmender Wert gefunden wird, wird der Datensatz von der Tabelle **orders** gelesen.

Die Suche nach einem Index wird von der Root-Page abwärts zu den Leaf-Pages durchgeführt (Siehe "Die Struktur eines Index" auf Seite 13-22.). Die Root-Page wird fast immer im Page-Puffer gehalten, da sie sehr oft gebraucht wird. Die Chancen, eine Leaf-Page im Puffer zu finden, hängen von der Größe des Index ab; wenn die Größe der Tabelle steigt, dann werden die Chancen geringer.

Wenn eine Tabelle sehr groß ist, so daß die Anzahl der Index-Leaf-Pages viel größer als der Pufferbereich ist, dann muß bei fast jeder Suche neben der Page, die den Datensatz enthält, auch eine Leaf-Page gelesen werden. Die Anzahl der Pages, die gelesen werden, ist in etwa r*(index-height-1), wobei *r* die Anzahl der Datensätze ist, die gesucht werden. Obwohl dies sehr zeitintensiv ist, ist diese Vorgehensweise immer noch günstig. Die Alternative hierzu wäre, für jede Suche die ganze Tabelle **orders** zu lesen; in diesem Fall ist die Plattenauslastung proportional zu $r^2$.

Bei der anderen Art verwendet der Datenbankserver einen Index, indem er ihn sequentiell liest. Dies wird gemacht, um die Datensätze einer Tabelle in einer bestimmten Reihenfolge zu erhalten; diese Reihenfolge unterscheidet sich von der physikalischen Reihenfolge. Bei dieser Zugriffsart liest der Datenbankserver die Leaf-Pages der Reihe nach, und behandelt sie wie eine Liste von Datensätzen, die nach einem Schlüssel sortiert sind. Da die Index-Pages nur einmal gelesen werden, ist die gesamte Plattenauslastung proportional zu *r+i*; *i* ist die Anzahl der Leaf-Pages in einem Index.

## Zeitbedarf bei kleinen Tabellen

Eine Folgerung, die Sie aus den vorangegangenen Abschnitten ziehen können, ist, daß kleine Tabellen die Performance nicht herabsetzen. Eine "kleine" Tabelle belegt so wenig Pages, daß sie vollständig im Page-Puffer gehalten werden kann. In diesem Sinn ist jede Tabelle, die in vier oder weniger Pages paßt, "klein".

Die Tabelle **state** in der Datenbank **stores6** hat eine Größe von weniger als 1000 Byte; diese Tabelle enthält Abkürzungen der Staatennamen; sie paßt fast in zwei Pages. Sie kann praktisch ohne, daß es zu Performance-Problemen kommt, in jeder Abfrage angegeben werden. Es sind höchstens zwei Plattenzugriffe nötig, unabhängig davon, wie die Tabelle verwendet wird – sogar wenn sie mehrfach sequentiell gelesen wird.

## Zeitbedarf bei NLS

Sortieren und Indizieren von Daten, die landessprachliche Sonderzeichen enthalten, kann zu erheblichen Performanceverlusten führen. Falls Sie NLS (Native Language Support) nicht für all Ihre Daten vom Typ Character benötigen, sollten Sie lieber mit CHAR statt mit NCHAR (bzw. mit VARCHAR statt mit NVARCHAR) arbeiten. Der Aufwand für Sortieren, Indizieren und Vergleichsoperationen ist dann wesentlich geringer.

## Zeitbedarf bei einem Netzwerkzugriff

Immer, wenn Daten über ein Netz übertragen werden, treten Verzögerungen auf. Netzwerke werden in zwei Zusammenhängen verwendet:

- Eine Anwendung sendet die Abfrage über ein Netz zu einem Datenbankserver, der auf einem anderen Rechner liegt. Der Datenbankserver führt

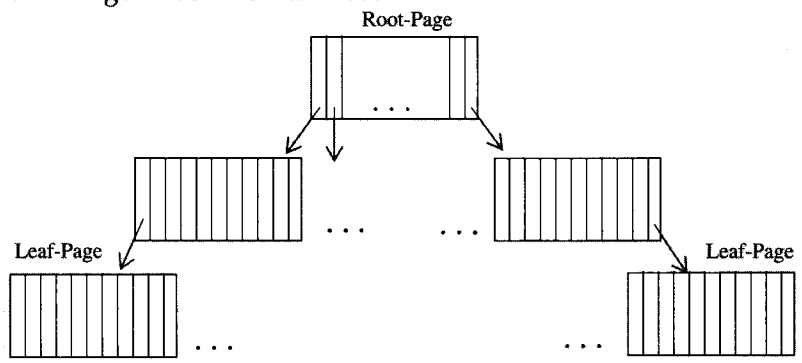
die Abfrage durch, wobei er auf seine Platte zugreift. Die gefundenen Datensätze werden über das Netzwerk an die Anwendung zurückgeliefert.

- Wenn Sie mit verteilten Daten arbeiten, liest und verändert der Datenbankserver Datensätze aus Datenbanken, die sich auf anderen Rechnern befinden als der Server selbst.

### Die Struktur eines Index

In einem Index sind die Pages hierarchisch angeordnet (fachlich ausgedrückt handelt es sich um einen *B+ Baum*), so wie es in dem Bild unten dargestellt ist. In der untersten Stufe enthält die Reihe der *Leaf-Pages* die Index-Schlüsselwerte von den Datensätzen der Tabelle; jeder Index-Wert verfügt über einen Zeiger, der auf die Position des Datensatzes zeigt, der diesen Wert enthält. Die Schlüssel der Leaf-Pages sind in üblicher Schlüssel-Reihenfolge.

In jeder höheren Stufe enthält eine Page den höchsten Schlüsselwert, der in jeder Page der nächsttieferen Stufe gefunden werden kann. Die oberste Stufe bildet eine einzelne *Root-Page*, von der aus jede weitere Page des Index gefunden werden kann.




Die Größe eines Eintrags ist gleich der Breite der indizierten Spalte in Byte plus 4. Wenn Sie dies und die Page-Größe kennen, dann können Sie die Anzahl der Einträge pro Seite abschätzen. Dies ist nur eine ungefähre Schätzung, da zum einen beim Speichern der Schlüssel die Daten etwas komprimiert werden und zum anderen nicht alle Pages voll sind.

Die Anzahl der Stufen hängt ab von der Anzahl der Datensätze, die indiziert werden und von der Anzahl der Einträge pro Seite. Nehmen wir an, eine Page enthält 100 Schlüssel. Dann hat eine Tabelle mit 100 Sätzen eine einzige Index-Page, die Root-Page. Eine Tabelle mit 101 bis 10.000 Sätzen hat einen zwei-stufigen Index, der aus einer Root-Page und aus 2 bis 100 Leaf-Pages steht. Eine Tabelle mit mehr als 1.000.000 Sätzen hat nur einen drei-stufigen Index (daher muß der vorangegangene Index einer sehr großenTabelle gehören).

- Die Daten, die über ein Netzwerk übertragen werden, bestehen aus Meldungen und aus Datensätzen, die in Blöcken zusammengefaßt sind. Ein Block entspricht der Größe eines Puffers. Beide Möglichkeiten unterscheiden sich in vielen Einzelheiten voneinander. Aber man kann sie mit Hilfe eines einfachen Modells gleich behandeln. In diesem Modell sendet ein Rechner, der *Sender*, eine Anfrage an einen anderen Rechner, den *Empfänger*. Dieser schickt als Antwort einen Datenblock von der Tabelle zurück.

Jedesmal, wenn Daten über ein Netzwerk übertragen werden, sind Verzögerungen in den folgenden Situationen unvermeidbar:

- Wenn das Netz bereits aktiv ist, muß der Sender mit der Übertragung warten, bis er an der Reihe ist. Solche Verzögerungen sind normalerweise kurz, kleiner als eine Tausendstelsekunde. Aber in stark belasteten Netzwerken können sie auf Zehntelsekunden und mehr anwachsen.
- Der Empfänger bearbeitet eventuell Anfragen von mehr als einem Sender, so daß eine Anfrage in eine Warteschlange eingereiht wird; die Wartezeit kann von Tausendstelsekunden bis zu Sekunden reichen.
- Wenn der Empfänger die Anfrage bearbeitet, kommt für den Plattenzugriff und die Speicheroperationen ein Zeitaufwand hinzu, wie er in den vorangegangenen Abschnitten beschrieben wurde.
- Auch die Übertragung der Antwort über das Netz kann zu Verzögerungen führen.

Verzögerungen bei Netzwerkzugriffen können also in sehr hohem Maße schwanken. Sobald die Netzauslastung steigt, verschlechtern sich all diese Faktoren. Die Verzögerungen, die bei der Übertragung entstehen, wachsen für beide Richtungen. Die Warteschlange des Empfängers wird länger. Und die Chance, daß eine Page im Puffer des Empfängers bleibt, verringert sich. Daher kann die Zugriffszeit auf Netzwerke ziemlich schnell von sehr niedrig auf extrem hoch ansteigen.

Der von **INFORMIX-OnLine Dynamic Server** verwendete Optimierer geht davon aus, daß der Zugriff auf einen Datensatz über ein Netzwerk länger dauert als der Zugriff auf einen Datensatz in einer lokalen Datenbank. Der Optimierer hat keine Möglichkeit, den Zeitbedarf der diversen Netzzugriffe zu berechnen.

# Die Bedeutung der Tabellenreihenfolge

Die Reihenfolge, in der die Tabellen durchsucht werden, wirkt sich stark auf die Geschwindigkeit einer Join-Operation aus. Dieses Konzept kann mit einigen Beispielen näher erläutert werden. Diese Beispiele zeigen, wie ein Datenbankserver arbeitet.

## Join ohne Filter

Diese SELECT-Anweisung führt einen dreifachen Join durch:

```
SELECT C.customer_num, O.order_num, SUM (items.total_price)
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
        AND O.order_num = I.order_num
    GROUP BY C.customer_num, O.order_num
```

Stellen Sie sich einmal vor, daß Indizes niemals erfunden wurden. Ohne Indizes hat der Datenbankserver keine andere Wahl, als diese Operation mit einer geschachtelten Schleife durchzuführen. Einer von zwei möglichen *Abfrageplänen* wird in Bild 13-2 gezeigt; dieser Abfrageplan ist in einem Pseudo-Programmcode dargestellt. Ein Abfrageplan gibt die Reihenfolge an, in der der Datenbankserver die Tabellen durchsucht, und die Methode, mit der auf die Tabellen zugegriffen wird.

```
for each row in the customer table do:
    read the row into C
    for each row in the orders table do:
        read the row into O
        if O.customer_num = C.customer_num then
            let Sum = 0
            for each row in the items table do:
                read the row into I
                if I.order_num = O.order_num then
                    let Sum = Sum + I.total_price
                end if
            end for
            prepare an output row from C,I,and Sum
        end if
    end for
end for
```

*Bild 13-2* ***Ein Abfrageplan in Pseudocode***

Diese Prozedur liest die folgenden Datensätze:

- Alle Datensätze von der Tabelle **customer** einmal
- Alle Datensätze von der Tabelle **orders** einmal für jeden Datensatz von der Tabelle **customer**
- Alle Datensätze von der Tabelle **items** einmal für jeden Datensatz von der Tabelle **orders**, der in der Ausgabe erscheint (jeder Datensatz der Tabelle **orders** sollte einmal erscheinen)

Dies ist nicht der einzig mögliche Abfrageplan; ein anderer vertauscht lediglich die Rollen der Tabellen **customer** und **orders**; für jeden Datensatz der Tabelle **orders** liest die Prozedur alle Sätze der Tabelle **customer** und sucht nach übereinstimmenden Werten in der Spalte **customer_num**. Die Prozedur liest, wenn auch in einer anderen Reihenfolge, die gleiche Anzahl Datensätze.

## Join mit Spalten-Filtern

Beim vorangegangenen Beispiel gab es bei der Ausführung der beiden Abfragepläne keinen Unterschied in der Ausführungszeit. Die Angabe eines *Spalten-Filters* verändert diesen Sachverhalt. Ein Spalten-Filter ist ein WHERE-Ausdruck, der die Anzahl derjenigen Datensätze reduziert, die eine Tabelle zu einem Join beisteuert. Hier wurde der vorangegangenen Abfrage ein Filter hinzugefügt:

```
SELECT C.customer_num, O.order_num, SUM (items.total_price)
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
        AND O.order_num = I.order_num
        AND O.paid_date IS NULL
    GROUP BY C.customer_num, O.order_num
```

Der Ausdruck `O.paid_date IS NULL` *filtert* einige Datensätze *heraus* und verringert somit die Anzahl der Datensätze, die von der Tabelle **orders** verwendet werden. Auch hier gibt es zwei mögliche Abfragepläne. Der Plan, der mit dem Lesen der Tabelle **orders** beginnt, ist in Bild 13-3 in Pseudocode dargestellt.

```
for each row in the orders table do:
    read the row into O
    if O.paid_date is null then
        for each row in the customer table do:
            read the row into C
            if O.customer_num = C.customer_num then
                let Sum = 0
                for each row in the items table do:
                    read the row into I
                    if I.order_num = O.order_num then
                        let Sum = Sum + I.total_price
                    end if
                end for
                prepare an output row from C,I,and Sum
            end if
        end for
    end if
end for
```

*Bild 13-3* ***Einer von zwei Abfrageplänen in Pseudocode***

*pdnull* steht für die Anzahl der Datensätze aus der Tabelle **orders**, die den Filter passiert haben. Dies ist die Zahl, die die folgende Abfrage zurückliefert:

```
SELECT COUNT(*) FROM orders WHERE paid_date IS NULL
```

Setzen wir voraus, daß es für jeden Auftrag genau einen Kunden gibt. Dann können wir sagen, daß der Plan in Bild 13-3 diese Datensätze liest:

- Alle Datensätze der Tabelle **orders** einmal
- Alle Datensätze der Tabelle **customer** *pdnull*-mal

- Alle Datensätze der Tabelle **items** *pdnull*-mal

```
for each row in the customer table do:
    read the row into C
    for each row in the orders table do:
        read the row into O
        if O.paid_date is null and
           O.customer_num = C.customer_num then
            let Sum = 0
            for each row in the items table do:
                read the row into I
                if I.order_num = O.order_num then
                    let Sum = Sum + I.total_price
                end if
            end for
            prepare an output row from C,I,and Sum
        end if
    end for
```

*Bild 13-4* ***Der alternative Abfrageplan in Pseudocode***

Ein alternativer Abfrageplan ist in Bild 13-4 gezeigt; dieser Plan liest zuerst aus der Tabelle **customer**. Da im ersten Schritt kein Filter verwendet wird, liest dieser Plan folgende Datensätze:

- Alle Datensätze der Tabelle **customer** einmal
- Alle Datensätze der Tabelle **orders**, je einmal für jeden Datensatz der Tabelle **customer**
- Alle Datensätze der Tabelle **items**, *pdnull*-mal

Die Abfragepläne in Bild 13-3 und Bild 13-4 erzeugen dieselbe Ausgabe, wenn auch in unterschiedlicher Reihenfolge. Sie unterscheiden sich darin, daß ein Plan eine Tabelle *pdnull*-mal liest, während der andere eine Tabelle `SELECT COUNT(*) FROM customer`-mal liest. In einer wirklichen Anwendung kann die Wahl, die der Optimierer trifft, einen Unterschied von Tausenden Plattenzugriffen ausmachen.

## Verwendung der Indizes

In den vorangegangenen Beispielen wurden keine Indizes oder Constraints verwendet. Dies war unrealistisch, da fast alle Tabellen über einen oder mehrere Indizes oder Constraints verfügen. Das Vorhandensein von Indizes oder Contraints wirkt sich in einem Abfrageplan unterschiedlich aus. Bild 13-5

zeigt für die vorherige Abfrage den Grundriß eines Abfrageplans, so wie er bei der Verwendung von Indizes aufgebaut sein könnte. (Siehe auch "Die Struktur eines Index" auf Seite 13-22.)

```
for each row in the customer table do:
    read the row into C
    look up C.customer_num in index on orders.customer_num
    for each matching row in the orders table do:
        read the row into O
        if O.paid_date is null then
            let Sum = 0
            look up O.order_num in index on items.order_num
            for each matching row in the items table do:
                read the row into I
                let Sum = Sum + I.total_price
            end for
            prepare an output row from C,I,and Sum
        end if
    end for
end for
```

*Bild 13-5* ***Ein Abfrageplan, der Indizes verwendet (in Pseudocode)***

Da in einem Index die Schlüssel sortiert sind, können, sobald der erste übereinstimmende Schlüssel gefunden wurde, alle anderen Datensätze mit gleichem Schlüssel ohne weiteres Suchen gelesen werden. Dieser Abfrageplan liest nur die folgenden Datensätze:

- Alle Datensätze der Tabelle **customer** einmal
- Alle Datensätze der Tabelle **orders** einmal
- Diejenigen Datensätze der Tabelle **items**, die mit den *pdnull*-Datensätzen der Tabelle **orders** übereinstimmen

Im Vergleich zu den Plänen ohne Indizes ist dies eine enorme Verringerung des Aufwands. (Der umgekehrte Plan, zuerst die Tabelle **orders** lesen und anhand des Index die Datensätze in der Tabelle **customer** suchen, ist genauso effektiv.)

Jedoch muß jeder Plan, der mit dem Index arbeitet, sowohl die Indexdaten als auch die Daten der Sätze von der Platte lesen. Es ist schwierig, die Anzahl der gelesenen Index-Pages vorherzusagen. Dies liegt daran, daß einige der Index-Pages, die erst verwendet wurden, noch im Speicher gehalten werden und dort gefunden werden, wenn sie benötigt werden (Siehe "Plattenzugriff verwalten" auf Seite 13-16.).

Auch die physikalische Reihenfolge einer Tabelle kann sich auf die Dauer bei einer Indexverwendung auswirken. Wenn ein Cluster-Index auf der Spalte **customer_num** liegt, dann entspricht die physikalische Reihenfolge der Tabelle **customer** der Reihenfolge der Kundennummern. Dies entspricht ungefähr der Reihenfolge, die man erhalten würde, wenn die Kundennummern vom Datenbankserver vergebene SERIAL-Werte wären.

Wenn jedoch die physikalische Reihenfolge der Kundennummern willkürlich ist, dann könnte jede Suche im Index der Tabelle **orders** zu einer anderen Page der Indexschlüssel führen. Dies hat zur Folge, daß für fast jeden Datensatz eine Index-Page gelesen werden muß.

Wenn NLS aktiviert ist, werden Indizes auf NCHAR- und auf NVARCHAR-Spalten nach landesspezifischen Vergleichswerten verglichen. Beispielsweise wird das spanische Zeichen für Doppel-L (ll) als ein einzelnes Zeichen und nicht als eine Kombination aus zwei l behandelt.

### Die Join-Technik Sort-Merge

Der Tabellen-Join Sort-Merge stellt eine Alternative zu dem Tabellen-Join mittels einer geschachtelten Schleife dar. Diese Technik wird vom Optimierer nur dann in Betracht gezogen, wenn der Join auf einem Vergleichsfilter basiert und wenn es nicht schon einen Index gibt, der den Join ausführen kann.

Der Sort-Merge-Join verändert nicht die grundlegende Handlungsweise des Optimierers. Er stellt vielmehr eine größere Anzahl an Alternativen zur Verfügung als Joins mit geschachtelten Schleifen. Zu diesen Alternativen gehören die Verwendung temporärer Indizes und eine Veränderung in der Art, wie die ORDER BY- und GROUP BY-Pfade analysiert werden. Wenn die Anweisung SET EXPLAIN ON gesetzt wurde, dann wird in der Ausgabe der Weg angezeigt, den der Optimierer gewählt hat. Dies wird im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* und im Abschnitt "Den Plan lesen" auf Seite 13-13 beschrieben.

# Abfragen beschleunigen

Im allgemeinen können Sie Abfragen durch Veränderungen dergestalt schneller machen, daß sie

- weniger Datensätze lesen
- das Sortieren vermeiden, weniger Datensätze sortieren oder nach einem einfacheren Schlüssel sortieren
- Datensätze sequentiell lesen anstatt nicht sequentiell

Der Weg, um diese Ziele zu erreichen, ist nicht immer offensichtlich. Die spezifische Methode hängt von Einzelheiten der Anwendung und des Datenbank-Designs ab. Die folgenden Absätze vermitteln einen allgemeinen Ansatz. Es werden außerdem einige Techniken dargestellt, die unter bestimmten Umständen angewendet werden können.

## Eine Test-Umgebung vorbereiten

Zuerst wählen Sie eine Abfrage aus, die zu langsam ist. Dann legen Sie eine Umgebung fest, in der Sie vorhersagbare und wiederholbare Zeitmessungen dieser Abfrage vornehmen können. Ohne diese Umgebung können Sie sich nicht sicher sein, ob die Veränderung sinnvoll war.

Wenn Sie mit einem Mehrbenutzer-System oder einem Netzwerk arbeiten, sollten Sie Ihre Untersuchung immer zur selben Zeit durchführen, um wiederholbare Ergebnisse zu erhalten. Der Grund liegt darin, daß die Systemauslastung von Stunde zu Stunde völlig anders sein kann. Sinnvollerweise sollten Sie zu einem Zeitpunkt messen, bei dem die Systemauslastung gering ist: Sie können dann sicher sein, daß Sie nur den Zeitbedarf Ihrer Abfrage und nichts anderes messen.

Wenn die Abfrage in ein kompliziertes Programm eingebettet ist, sollten Sie erwägen, die SELECT-Anweisung herauszunehmen und interaktiv auszuführen. Sie können diese Anweisung auch in ein einfacheres Programm einbetten.

Wenn die reale Abfrage viele Minuten oder sogar Stunden für die Ausführung braucht, dann könnte es vorteilhaft sein, eine verkleinerte Datenbank zu erstellen, in der Tests schneller ausgeführt werden können. Dies kann sinnvoll sein, aber Sie müssen sich zweier möglicher Probleme bewußt sein:

- Der Optimierer kann in einer kleineren Datenbank eine andere Wahl treffen als in einer großen, selbst wenn die Tabellen ungefähr gleich groß sind. Überprüfen Sie, ob der Abfrageplan der realen und der Modell-Datenbank übereinstimmt.
- Die Ausführungszeit hängt selten linear von der Tabellengröße ab. Die Zeit für die Sortierung wächst z. B. schneller als die Tabellengröße; dies gilt auch für den Zeitbedarf beim indizierten Zugriff, wenn ein Index anstatt zweistufig dreistufig ist. Was in einer verkleinerten Umgebung als eine große Verbesserung erscheint, kann bei der Anwendung in der kompletten Datenbank unbedeutend sein.

Deshalb ist jede Schlußfolgerung, zu der Sie nach den Tests mit der Modell-Datenbank kommen, solange vorläufig, bis sie sich für die größere Datenbank bestätigt.

# Das Daten-Modell untersuchen

Untersuchen Sie die Definitionen aller Tabellen, Views und Indizes, die in der Datenbank verwendet werden. Sie können diese Details interaktiv untersuchen, indem Sie bei **DB-Access** oder **INFORMIX-SQL** die Option Tabelle verwenden. Achten Sie besonders auf die Indizes, auf die Datentypen der Spalten, die in Join-Bedingungen und beim Sortieren verwendet werden und auf Views. Dieses Kapitel setzt voraus, daß Sie das Schema nicht verändern können.

Informationen zu den Datentypen und Views erhalten Sie in Kapitel 9 dieses Handbuchs.

# Den Abfrageplan untersuchen

Um herauszufinden, welcher Abfrageplan verwendet wird, benutzen Sie die Anweisung SET EXPLAIN ON. Auf die folgenden Punkte sollten Sie besonders achten:

- Indizes - untersuchen Sie, ob und wie Indizes verwendet wurden.
- Filter - prüfen Sie, ob Ihre Filter auch selektiv genug sind.
- Abfrage - prüfen Sie Ihre Abfragen nochmal um sicherzugehen, daß der Abfrageplan wirklich optimal ist.

## Autoindizes durch Indizes ersetzen

Wenn der Abfrageplan für eine große Tabelle einen *Autoindex*pfad enthält, dann sollten Sie dies als eine Empfehlung des Optimierers annehmen. Sie sollten hier die Spalte indizieren. Wenn Sie die Abfrage nur gelegentlich durchführen, dann ist es allerdings vernünftig, den Datenbankserver einen Index aufbauen und verwerfen zu lassen. Wenn die Abfrage aber täglich durchgeführt wird, können Sie durch die Erstellung eines permantenten Index Zeit sparen.

## Datenverteilungen für gefilterte Spalten erzeugen

Mit der Anweisung UPDATE STATISTICS in den Modi HIGH oder MEDIUM erhalten Sie Informationen über die Verteilung von Datenwerten in den einzelnen Spalten. Diese Information wird in der Systemtabelle **sysdistrib** abgelegt. Zum Zeitpunkt der Optimierung wird diese Information dazu verwendet, die Anzahl derjenigen Datensätze zu schätzen, die die Bedingungen einer Abfrage erfüllen. Damit wird auf den Aufwand der Abfrage geschlossen und der Optimierer wählt den besten Weg für die Ausführung aus. Da-

tenverteilungen führen zu besseren Schätzungen über die Anzahl von zurückgelieferten Datensätzen als die Spalten **colmin** und **colmax** aus der Systemtabelle **syscolumns**.

Wenn Sie die Statistiken zur Datenverteilung zum ersten Mal verwenden, sollten Sie UPDATE STATISTICS für alle Tabellen im Modus MEDIUM ablaufen lassen. Anschließend sollten Sie UPDATE STATISTICS im Modus HIGH für diejenigen Spalten laufen lassen, die indiziert werden. Bei Abfragen auf einzelnen Tabellen führt dies zu Schätzungen mit einem Fehler unter 2,5%, bezogen auf die Anzahl der Sätze in der Tabelle. (Bei Spalten, die im HIGH-Modus bearbeitet wurden, liegt die Standardschätzung bei einem Fehlerwert von 0,1%.)

Falls sich die Spaltenwerte nicht in einem erheblichen Ausmaß ändern, müssen Sie die Datenverteilung nicht erneut erzeugen. Sie können die Genauigkeit der Verteilung überprüfen, indem Sie das Ergebnis von **dbschema** (Option -hd) mit entsprechend konstruierten SELECT-Anweisungen vergleichen.

Weitere Informationen zum Thema Datenverteilung und UPDATE STATISTICS erhalten Sie im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

## Zusammengesetzte Indizes verwenden

Der Optimierer kann einen zusammengesetzten Index (einer der mehr als eine Spalte enthält) auf mehrere Arten verwenden. In seiner Hauptfunktion muß der Index sicherstellen, daß die Werte in den Spalten *abc* eindeutig sind. Zusätzlich kann ein Index über die Spalten *a*, *b* und *c* (in dieser Reihenfolge) auf folgende Arten verwendet werden:

- Filter-Ausdrücke in der Spalte *a* auswerten
- Spalte *a* mit einer anderen Tabelle verknüpfen
- ORDER BY oder GROUP BY auf die Spalten *a*, *ab* oder *abc* (aber nicht auf *b*, *c*, *ac* oder *bc*) anwenden

Beim Erzeugen eines zusammengesetzten Index sollten Sie die Spalten in folgender Reihenfolge angeben: zuerst die am häufigsten benutzte Spalte bis hin zur eher selten benutzten Spalte.

Wenn Ihre Anwendung mehrere lange Abfragen durchführt, die jeweils nach denselben Spalten sortieren, dann könnten Sie dadurch Zeit sparen, daß Sie einen zusammengesetzten Index auf diese Spalten setzen. In diesem Fall führen Sie das Sortieren nur einmal durch und speichern das Ergebnis; dieses können Sie dann in jeder weiteren Abfrage verwenden.

## Zuordnung der Filter-Selektivität

Die folgende Tabelle listet einige Selektivitäten auf, die der Optimierer den verschiedenen Filterarten zuweist. Dies ist keine vollständige Liste; weitere Ausdrucksformen werden später noch hinzukommen.

| **Ausdrucksform** | **Selektivität(F)** |
|---|---|
| *Index-Spalte* = *Literal-Wert*<br>*Index-Spalte* = *Hostvariable*<br>*Index-Spalte* IS NULL | F = 1/(Anzahl der eindeutigen Schlüssel in einem Index) |
| *Tab1.Index-Spalte* = *Tab2.Index-Spalte* | F = 1/(Anzahl der eindeutigen Schlüssel in einem größeren Index) |
| *Index-Spalte* > *Literal-Wert* | F = (*zweitgrößter* - *Literal-Wert*)/(*zweitgrößter* - *zweitkleinster*) |
| *Index-Spalte* < *Literal-Wert* | F = (*Literal-Wert* - *zweitkleinster*)/(*zweitgrößter* - *zweitkleinster*) |
| *beliebige-Spalte* IS NULL<br>*beliebige-Spalte* = *beliebiger-Ausdruck* | F = 1/10 |
| *beliebige-Spalte* > *beliebiger-Ausdruck*<br>*beliebige-Spalte* < *beliebiger-Ausdruck* | F = 1/3 |
| *beliebige-Spalte* MATCHES *beliebiger-Ausdruck*<br>*beliebige-Spalte* LIKE *beliebiger-Ausdruck* | F = 1/5 |
| EXISTS *Unterabfrage* | F = 1 wenn *Anzahl der von Unterabfragezurückgelieferten Datensätze* >0 , sonst 0 |
| NOT *Ausdruck* | F = 1 - F(*Ausdruck*) |
| *ausdr1* AND *ausdr2* | F = F(*ausdr1*) × F(*ausdr2*) |
| *ausdr1* OR *ausdr2* | F = F(*ausdr1*) + F(*ausdr2*) - F(*ausdr1*) × F(*ausdr2*) |
| *beliebige-Spalte* IN *Liste* | behandelt wie *beliebige-Spalte* = $Feld_1$ OR...OR *eine-Spalte* = $Feld_n$ |
| *beliebige-Spalte operator* ANY *Unterabfr.* | behandelt wie *beliebige-Spalte rel_operator* $Wert_1$ OR...OR *beliebige-Spalte operator* $Wert_n$ bei einer geschätzten Größe der Unterabfrage *n* |

**Schlüssel:**
*Index-Spalte*: erste oder einzige Spalte in einem Index (nur bei **INFORMIX-OnLine**)
*zweitgrößter, zweitkleinster*: zweitgrößte und zweitkleinste Schlüsselwerte einer indizierten Spalte (nur bei **INFORMIX-OnLine**)

### Die Verwendung von *oncheck* für "verdächtige" Indizes

Bei einigen Datenbankservern ist es möglich, daß ein Index durch einen internen Defekt uneffektiv wird. Wenn sich eine Abfrage, die einen Index verwendet, aus ungeklärten Gründen verlangsamt hat, dann sollten Sie das Dienstprogramm **oncheck** anwenden. Dieses überprüft die Integrität des Index und repariert diesen notfalls (Das Dienstprogramm **secheck** führt für **INFORMIX-SE** dasselbe aus.).

### Indizes nach Änderungen löschen und neu aufbauen

Nach umfangreichen Änderungen (nach dem Entfernen von vier oder mehr Datensätzen), kann die Indexstruktur uneffektiv geworden sein. Wenn ein Index weniger effektiv zu sein scheint als er sein sollte und wenn **oncheck** keine Fehler anzeigt, dann sollten Sie den Index löschen und dann wieder neu erzeugen.

## Die Filter der Spalten tunen

Die effektivsten Abfragen sind diejenigen, die nur möglichst wenig Informationen verarbeiten müssen. Sie beeinflussen die Informationsmenge einer SELECT-Anweisung mit Hilfe der WHERE-Klauseln. Die WHERE- Klauseln bezeichnet man auch als Filter. Die folgenden Abschnitte beinhalten Vorschläge, wie Sie die WHERE-Klauseln in Ihren Abfragen optimieren können:

- Vermeiden Sie korrelierte Unterabfragen.
- Vermeiden Sie komplizierte reguläre Ausdrücke.
- Vermeiden Sie in Ihren Abfragen Zeichenkettenausschnitte, die nicht am Anfang stehen.

### Korrelierte Unterabfragen vermeiden

In einer korrelierten Unterabfrage erscheint eine Spaltenüberschrift sowohl in der Auswahlliste der Hauptabfrage als auch in der WHERE-Klausel der Unterabfrage. Da das Ergebnis der Unterabfrage bei jedem Datensatz, den der Datenbankserver untersucht, unterschiedlich sein kann, wird die Unterabfrage für jeden Datensatz erneut durchgeführt. Dies ist aber nur der Fall, wenn sich die aktuellen korrelierten Werte von den vorangegangenen unterscheiden. Der Optimierer versucht, einen Index für korrelierte Werte zu verwenden, um identische Werte zusammenzubinden. Diese Prozedur kann viel Zeit verbrauchen. Einige Abfragen können mit SQL leider nicht durchgeführt

werden, ohne dabei korrelierte Unterabfragen zu verwenden. Bei den meisten ist es aber möglich, einen Join zu verwenden, der schneller ausgeführt wird als eine korrelierte Unterabfrage.

Wenn Sie in einer zeitaufwendigen SELECT-Anweisung eine Unterabfrage entdecken, dann sollten Sie überprüfen, ob diese korreliert ist. (Eine nicht korrelierte Unterabfrage wird nur einmal ausgeführt; d. h. eine, bei der in der Unterabfrage keine Werte der Hauptabfrage überprüft werden.) Falls die Abfrage korreliert ist, dann sollten Sie die Abfrage umschreiben, um dies zu vermeiden. Wenn dies nicht möglich ist, dann suchen Sie nach Möglichkeiten, um die Anzahl der zu untersuchenden Datensätze zu reduzieren. Versuchen Sie z. B., andere Filter-Ausdrücke in die WHERE-Klausel einzufügen; oder schreiben Sie eine Teilmenge der Datensätze in eine temporäre Tabelle und suchen Sie nur darin.

## Schwierige reguläre Ausdrücke vermeiden

Die Schlüsselwörter MATCHES und LIKE unterstützen *Jokerzeichen* – in der Fachsprache bekannt als *reguläre Ausdrücke.* Der Datenbankserver kann einige reguläre Ausdrücke schlechter verarbeiten als andere. Ein Jokerzeichen an der ersten Stelle zwingt den Datenbankserver, jeden Wert in der Spalte zu überprüfen. Das folgende Beispiel (suche Kunden, deren Vorname nicht mit einem *y* aufhört) zeigt dies:

```
SELECT * FROM customer WHERE fname NOT LIKE ''%y''
```

Ein Index kann bei so einem Filter nicht verwendet werden. Daher muß die Tabelle sequentiell durchsucht werden.

Wenn für eine schwierige Überprüfung reguläre Ausdrücke notwendig sind, dann sollten Sie vermeiden , diese mit einem Join zu kombinieren. Verarbeiten Sie zuerst eine Tabelle, wobei Sie für die Überprüfung reguläre Ausdrücke verwenden, um die gewünschten Datensätze auszuwählen. Sichern Sie das Ergebnis in einer temporären Tabelle und verknüpfen Sie diese dann mit den anderen Tabellen.

Die Überprüfung mit regulären Ausdrücken verhindern nur dann nicht die Verwendung eines Index (wenn einer existiert), wenn die Jokerzeichen in der Mitte oder am Ende des Operanden stehen.

### Zeichenkettenausschnitte vermeiden, die nicht am Anfang stehen

Auch ein Filter, der sich nicht auf den Anfang eines Zeichenkettenausschnitts bezieht, macht es erforderlich, daß jeder Wert der Spalte geprüft werden muß. Hierzu ein Beispiel:

```
SELECT * FROM customer
    WHERE zipcode[4,5] > ''50''
```

Der Optimierer verwendet keinen Index, um so einen Filter auszuwerten; selbst dann nicht, wenn einer vorhanden ist.

Der Optimierer verwendet einen Index, um einen Filter zu bearbeiten, der einen *einleitenden* Zeichenkettenausschnitt einer indizierten Spalte überprüft. Die Überprüfung eines Zeichenkettenausschnitts kann jedoch mit der Verwendung eines zusammengesetzten Index beeinträchtigt werden; hierbei müssen sowohl die Spalte, für die der Zeichenkettenausschnitt angegeben ist, als auch andere Spalten überprüft werden.

## Die Abfrage überdenken

Suchen Sie jetzt, da Sie verstehen, wie die Abfrage funktioniert, nach Möglichkeiten, mit weniger Aufwand dasselbe Ergebnis zu erhalten. Die folgenden Vorschläge entsprechen der vorangegangenen Liste.

- Views durch Joins ersetzen
- Sortieren vermeiden oder vereinfachen
- Sequentiellen Zugriff auf große Tabellen ausschließen
- Die Verwendung der UNION-Anweisung, um sequentielle Zugriffe zu vermeiden.

### Views durch Joins ersetzen

Sie entdecken eventuell, daß die Abfrage einer Tabelle mit einer View verknüpft ist, wobei die View selbst ein Join ist. Wenn Sie die Abfrage umschreiben, so daß Sie weniger Tabellen direkt miteinander verknüpfen, dann können Sie einen einfacheren Abfrageplan erstellen.

### Sortieren vermeiden oder vereinfachen

Das Sortieren ist nicht unbedingt von Nachteil. Der Sortier-Algorithmus des Datenbankservers ist sehr ausgefeilt und äußerst effektiv. Er ist sicherlich genauso schnell wie ein externes Sortierprogramm, das Sie für dieselben Daten

anwenden. Solange das Sortieren nur gelegentlich oder für eine relativ kleine Anzahl von Ausgabe-Datensätze durchgeführt wird, braucht es nicht vermieden zu werden.

Sie sollten jedoch wiederholtes Sortieren großer Tabellen vermeiden oder vereinfachen. Der Optimierer vermeidet einen Sortierdurchgang immer dann, wenn er die Ausgabe in der eigenen Reihenfolge automatisch erstellen kann, indem er einen Index verwendet. Hier sind einige Faktoren aufgelistet, die den Optimierer von der Verwendung eines Index abhalten:

- Eine oder mehrere Spalten, nach denen sortiert wird, sind nicht im Index enthalten.
- Die Spalten sind im Index und in der ORDER BY- bzw. GROUP BY-Klausel in unterschiedlicher Reihenfolge angegeben.
- Die Spalten, nach denen sortiert wird, sind aus verschiedenen Tabellen.

Eine weitere Möglichkeit, um das Sortieren zu vermeiden, wird im Abschnitt "Verwendung temporärer Tabellen, um Mehrfach-Sortierungen zu vermeiden" auf Seite 13-40 erläutert.

Wenn das Sortieren erforderlich ist, dann suchen Sie nach Möglichkeiten, es zu vereinfachen. Wie bereits im Abschnitt "Zeitbedarf beim Sortieren" auf Seite 13-17 erläutert, geht das Sortieren nach weniger oder kleineren Spalten schneller.

## Sequentiellen Zugriff auf große Tabellen ausschließen

Ein sequentieller Zugriff auf eine Tabelle, die nicht die erste im Plan ist, ist verhängnisvoll; dies bedeutet, daß jeder Datensatz der Tabelle einmal gelesen wird, und zwar für jeden Datensatz, der aus den vorangegangenen Tabellen ausgewählt wurde. Sie sollten beurteilen, wie oft dies sein könnte: vielleicht nur ein paarmal, aber vielleicht auch hundert- oder tausendmal.

Wenn die Tabelle klein ist, macht es nichts aus, sie immer wieder zu lesen; die Tabelle befindet sich vollständig im Speicher. Das sequentielle Durchsuchen einer Tabelle, die sich im Speicher befindet, kann schneller gehen, als wenn die gleiche Tabelle anhand des Index durchsucht wird; dies trifft besonders dann zu, wenn die Index-Pages im Speicher andere nützliche Pages aus den Puffern verdrängen.

Wenn die Tabelle jedoch größer als ein paar Pages ist, dann kann sich wiederholter sequentieller Zugriff auf die Performance sehr ungünstig auswirken. Eine Möglichkeit, um dies zu vermeiden, wäre, die Join-Spalte zu indizieren.

Das Indizieren wird im Zusammenhang mit dem Datenbank-Design im Abschnitt "Indexverwaltung" auf Seite 10-23 erläutert. Jeder Benutzer, der über die Resource-Berechtigung verfügt, kann jedoch weitere Indizes erzeugen. Mit dem Kommando CREATE INDEX erstellen Sie einen Index.

Der vom Index benötigte Speicherplatz ist proportional zu der Größe der Schlüsselwerte und der Anzahl der Datensätze (Siehe "Die Struktur eines Index" auf Seite 13-22.). Weiterhin muß der Datenbankserver den Index immer dann auf den neuesten Stand bringen, wenn Datensätze eingefügt, gelöscht oder verändert wurden; dies verlangsamt die Operationen. Wenn es erforderlich ist, können Sie mit dem Kommando DROP INDEX nach einer Reihe von Abfragen den Index löschen; dies setzt Speicherplatz frei und macht Veränderungen der Tabelle einfacher.

## Die Verwendung von Unions, um sequentiellen Zugriff zu vermeiden

Bestimmte Formen der WHERE-Klausel zwingen den Optimierer, sequentiell zuzugreifen, sogar dann, wenn alle Spalten indiziert sind, die geprüft werden. Die folgende Abfrage erzwingt einen sequentiellen Zugriff auf die Tabelle **orders**:

```
SELECT * FROM orders
    WHERE (customer_num = 104 AND order_num > 1001)
        OR order_num = 1008
```

Das Schlüsselelement dieser Anweisung ist, daß zwei (oder mehrere) einzelne Datensatzmengen abgerufen werden. Die Mengen sind durch relationale Ausdrücke bestimmt; diese Ausdrücke sind durch OR verknüpft. In dem Beispiel wird eine Menge mit dieser Überprüfung ausgewählt:

```
(customer_num = 104 AND order_num > 1001)
```

Die andere Menge wird mit dieser Prüfung ausgewählt:

```
order_num = 1008
```

Der Optimierer verwendet einen sequentiellen Zugriffspfad, obwohl die Spalten **customer_num** und **order_num** indiziert sind.

Abfragen dieser Art können Sie beschleunigen, indem Sie diese in UNION-Abfragen umwandeln. Schreiben Sie für jede Datensatzmenge eine eigene SELECT-Anweisung und verbinden Sie diese Anweisungen mit dem Schlüsselwort UNION. Das vorhergehende Beispiel wurde auf diese Weise umgeschrieben:

```
SELECT * FROM orders
    WHERE (customer_num = 104 AND order_num > 1001)

UNION

SELECT * FROM orders
    WHERE order_num = 1008
```

Der Optimierer verwendet für jede Abfrage einen Indexpfad.

Sie können die WHERE-Klausel auch folgendermaßen formulieren:

```
WHERE (customer_num = 104 OR order_num = 1008) AND
    (order_num > 1001 OR order_num = 1008)
```

## Beurteilung des Optimierungsgrades

Mit HIGH, dem voreingestellten Optimierungsgrad, erreichen Sie in der Regel ein Höchstmaß an Performance. Normalerweise ist die Zeit, die zur Optimierung benötigt wird, vernachlässigbar. Falls Ihre Experimente mit der Anwendung aber zeigen, daß Ihre Abfrage immer noch zu lange braucht, können Sie den Optimierungsgrad auf LOW setzen. Anschließend überprüfen Sie das Ergebnis von SET EXPLAIN, um herauszubekommen, ob der Optimierer die gleiche Abfragestrategie wie zuvor verwendet.

Mit der Anweisung SET OPTIMIZATION legen Sie den Optimierungsgrad fest. Diese Anweisung wird ausführlich im Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* beschrieben.

## Abfragen beschleunigen mit temporären Tabellen

Das Erstellen einer temporären, sortierten Teilmenge einer Tabelle kann eine Abfrage manchmal beschleunigen. Dies kann auch mehrfache Sortiervorgänge vermeiden helfen.

### Verwendung temporärer Tabellen, um Mehrfach-Sortierungen zu vermeiden

Nehmen wir z. B. an, Ihre Anwendung erstellt eine Reihe von Listen für Kunden mit Außenständen; hierbei wird für jeden Postleitzahl-Hauptbereich eine Liste erstellt, die nach den Kundennamen sortiert ist. Anders gesagt, es gibt eine Reihe von Abfragen dieser Form (die verwendeten Tabellen und Spaltennamen sind hypothetisch):

```
SELECT cust.name, rcvbles.balance, ...other columns...
    FROM cust, rcvbles
    WHERE cust.customer_id = rcvbles.customer_id
        AND rcvbls.balance > 0
        AND cust.postcode LIKE ''98_ _ _''
    ORDER BY cust.name
```

Diese Abfrage liest die ganze Tabelle **cust**. Bei jedem Datensatz mit der richtigen Postleitzahl sucht der Datenbankserver den Index zu **rcvbles.customer_id**; er führt für jeden übereinstimmenden Wert einen nicht sequentiellen Plattenzugriff durch. Die Datensätze werden in eine temporäre Datei geschrieben und sortiert.

Diese Prozedur ist zulässig, wenn die Abfrage nur einmal ausgeführt wird. Das Beispiel enthält aber eine Reihe von Abfragen, die alle denselben Arbeitsaufwand erfordern.

Eine Alternative hierzu ist, alle Kunden mit Außenständen in eine temporäre Tabelle zu schreiben, wobei nach den Kundennamen sortiert ist. Folgendes Beispiel zeigt dies:

```
SELECT cust.name, rcvbles.balance, ...other columns...
    FROM cust, rcvbles
    WHERE cust.customer_id = rcvbles.customer_id
        AND cvbls.balance > 0
    ORDER BY cust.name
    INTO TEMP cust_with_balance
```

Jetzt können Sie Abfragen in dieser Form direkt an die temporäre Tabelle stellen:

```
SELECT *
    FROM cust_with_balance
    WHERE postcode LIKE ''98_ _ _''
```

Jede Abfrage liest die temporäre Tabelle sequentiell, aber sie enthält weniger Datensätze als die primäre Tabelle. Es werden keine nicht sequentiellen Plattenzugriffe durchgeführt. Eine Sortierung ist nicht erforderlich, da die physikalische Reihenfolge der Tabelle der gewünschten Reihenfolge entspricht. Der gesamte Aufwand sollte um einiges niedriger sein als zuvor.

Es gibt möglicherweise einen Nachteil: jede Veränderung, die in der primären Tabelle durchgeführt wird, nachdem die temporäre Tabelle erzeugt wurde, wird in der Ausgabe nicht berücksichtigt. Für die meisten Anwendungen stellt dies kein Problem dar, für einige kann dies aber der Fall sein.

## Nicht-sequentiellen Zugriff durch Sortieren ersetzen

Ein nicht-sequentieller Plattenzugriff ist besonders langsam. Die Sprache SQL kompensiert diese Tatsache und macht es einfach, Abfragen zu schreiben, die auf eine riesige Anzahl Pages nicht-sequentiell zugreifen. Manchmal können Sie eine Abfrage dadurch verbessern, daß Sie die Sortierfähigkeit des Datenbankservers beim nicht-sequentiellen Zugriff einsetzen. Das folgende Beispiel zeigt dies; es zeigt auch, wie der Zeitaufwand eines Abfrageplans numerisch abgeschätzt wird.

In Bild 13-6 wird eine große Herstellerfirma, deren Datenbank drei Tabellen enthält, schematisch dargestellt. (Tabellendiagramme dieser Art sind in Kapitel 8 dieses Handbuchs dargestellt. Es werden nicht alle Kunden gezeigt.)

Die erste Tabelle, **part**, enthält die Teile, die in den Produkten der Firma verwendet werden.

Die zweite, **vendor**, enthält Daten über die Anbieter, die diese Teile verkaufen. Die dritte, **parven**, gibt an, welche Teile von welchem Anbieter geliefert werden, und zu welchem Preis.

part

| part_num | part_desc |
|---|---|
| pk | |
| 100,000<br>999,999 | spiral spanner<br>spotted paint |

vendor

| vendor_num | vendor_name |
|---|---|
| pk | |
| 9,100,000<br>9,999,999 | Wrenchers SA<br>Spottiswode |

parven

| part_num | vendor_num | price |
|---|---|---|
| pk, fk | pk, fk | |
| 100,000<br>999,999 | 9,100,000<br>9,999,999 | $14.98<br>$0.066 |

*Bild 13-6* *Drei Tabellen aus einer Hersteller-Datenbank*

Die folgende Abfrage dieser Tabellen wird regelmäßig ausgeführt, um eine Liste aller verfügbaren Preise zu erstellen:

```
SELECT part_desc, vendor_name, price
    FROM part, vendor, parven
    WHERE part.part_num = parven.part_num
        AND parven.vendor_num = vendor.vendor_num
    ORDER BY part.part_num
```

Obwohl dies ein relativ einfacher Join über drei Tabellen zu sein scheint, braucht die Abfrage sehr lange. Als Teil der Untersuchung bereiten Sie eine Tabelle vor, die die ungefähre Größe der Tabelle und deren Indizes in Pages anzeigt. Die Tabelle wird in Bild 13-7 gezeigt. Eine Page hat eine Größe von 4048 Byte. Es werden nur ungefähre Angaben gemacht. Die tatsächliche Anzahl der Datensätze verändert sich häufig, und in jedem Fall können nur geschätzte Berechnungen durchgeführt werden.

Bei weiteren Untersuchungen werden Sie feststellen, daß der Index auf **part_num** ein Cluster-Index ist; daher befindet sich die Tabelle **part** in physikalischer Reihenfolge der Werte der Spalte **part_num**. Genauso ist es bei der tabelle **vendor**, die physikalisch nach den Werten der Spalte **vendor_num** sortiert ist. Die Tabelle **parven** befindet sich in keiner bestimmten Reihenfolge. Die Größen dieser Tabellen zeigen an, daß die Chancen für einen erfolgreichen, nicht-sequentiellen Zugriff auf eine gepufferte Page sehr gering sind.

Der beste Abfrageplan für diese Abfrage (es ist nicht unbedingt der, den der Optimierer auswählt) ist, zuerst die Tabelle **part** sequentiell zu lesen und dann die Werte der Spalte **part_num** zu verwenden, um auf die passenden Sätze der Tabelle **parven** zuzugreifen (ungefähr 1,5 Datensätze pro **part_num**-Wert); und anschließend den Wert **parven.vendor_num** zu verwenden, um über den Index auf die Tabelle **vendor** zuzugreifen.

| Tabelle | Satz Länge | Anzahl Sätze | Sätze/Page | Data Pages |
|---|---|---|---|---|
| part | 150 | 10,000 | 25 | 400 |
| vendor | 150 | 1,000 | 25 | 40 |
| parven | 13 | 15,000 | 300 | 50 |

| Indizes | Größe des Schlüssels | Schlüssel/Page | Leaf Pages |
|---|---|---|---|
| part_num | 4 | 500 | 20 |
| vendor_num | 4 | 500 | 2 |
| parven (composite) | 8 | 250 | 60 |

*Bild 13-7* ***Dokumentation der Größe der Tabellen und der Indizes in Pages***

Die sich daraus ergebende Anzahl der Plattenzugriffe kann wie folgt abgeschätzt werden:

- 400 Pages werden von der Tabelle **part** sequentiell gelesen
- 10.000 nicht-sequentielle Zugriffe auf die Tabelle **parven**, jeweils 2 Pages (eine Index Leaf-Page, eine Daten-Page) oder 20.000 Zugriffe auf Platten-Pages
- 15.000 nicht- sequentielle Zugriffe auf die Tabelle **vendor** oder 30.000 Zugriffe auf Platten-Pages

Sogar ein einfacher Join auf eine Tabelle, die gut indiziert ist, kann 50.400 Plattenzugriffe erfordern. Dies kann jedoch dadurch verbessert werden, daß die Abfrage in drei Schritte unterteilt wird; verwenden Sie hierfür temporäre Tabellen wie es in Bild 13-8 gezeigt wird. (Die folgende Lösung stammt von W. H. Inmon; sie ist einem Beispiel seines Buches *Optimizing Performance in DB2 Software*, Prentice-Hall 1988, entnommen.)

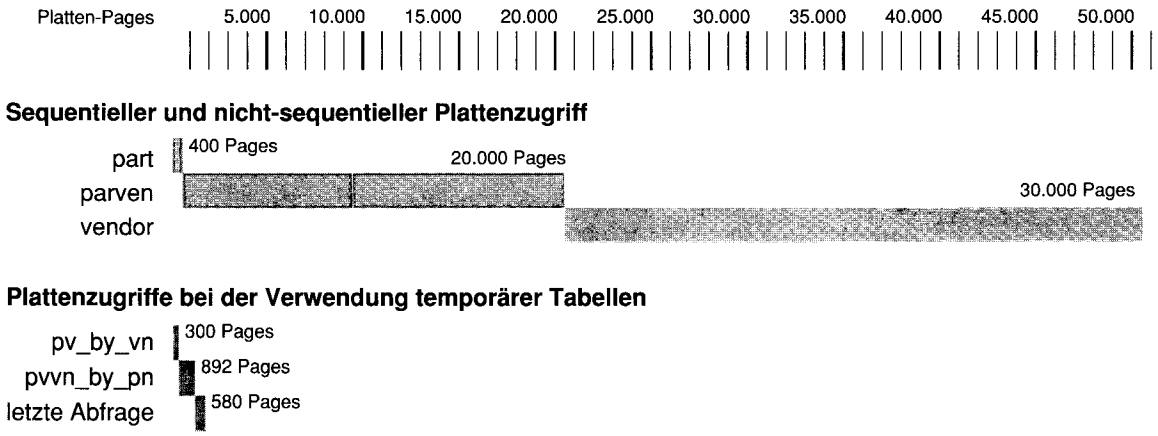



*Bild 13-8* *Eine Abfrage unter Verwendung temporärer Tabellen in drei Schritte unterteilen*

Zuerst müssen die Daten der Tabelle **parven** in die Reihenfolge der Werte der Spalte **vendor_num** gebracht werden:

```
SELECT part_num, vendor_num, price
    FROM parven
    ORDER BY vendor_num
    INTO TEMP pv_by_vn
```

Diese Anweisung liest die Tabelle **parven** sequentiell (50 Pages), schreibt eine temporäre Tabelle (50 Pages) und sortiert diese. Der Aufwand des Sortierens umfaßt bei einem Sortierdurchlauf 200 Pages, für die gesamte Sortierung 300 Pages.

Verknüpfen Sie diese temporäre Tabelle mit der Tabelle **vendor** und schreiben Sie das Ergebnis in eine weitere temporäre Tabelle; sortieren Sie dabei das Ergebnis nach der Spalte **part_num**:

```
SELECT pv_by_vn.*, vendor.vendor_name
    FROM pv_by_vn, vendor
    WHERE pv_by_vn.vendor_num = vendor.vendor_num
    ORDER BY pv_by_vn.part_num
    INTO TEMP pvvn_by_pn;
DROP TABLE pv_by_vn
```

Diese Abfrage liest **pv_by_vn** sequentiell (50 Pages). Sie greift auf die Tabelle **vendor** über den Index 15.000 mal zu. Da die Schlüssel aber in der Reihenfolge der Werte der Spalte **vendor_num** vorliegen, wird die Tabelle **vendor** sequentiell anhand des Index (42 Pages) gelesen.

Wenn das Feld **vendor_name** 32 Byte lang ist, enthält die Ausgabe-Tabelle 95 Datensätze pro Page und sie umfaßt ungefähr 160 Pages. Diese Pages werden geschrieben und anschließend sortiert; dies bedeutet, daß 5×160=800 Pages gelesen und geschrieben werden. Deshalb liest oder schreibt diese Abfrage insgesamt 892 Pages.Verknüpfen Sie nun diese Ausgabe mit der Tabelle **part**, um das Endergebnis zu erhalten:

```
SELECT pvvn_by_pn.*, part.part_desc
    FROM pvvn_by_pn, part
    WHERE pvvn_by_pn.part_num = part.part_num;
DROP TABLE pvvn_by_pn
```

Die obige Abfrage liest **pvvn_by_pn** sequentiell (160 Pages). Sie greift auf die Tabelle **part** über den Index 15.000 mal zu. Da die Schlüssel aber auch hier in der Reihenfolge der Werte der Spalte **part_num** vorliegen, wird die Tabelle **part** sequentiell (400 Pages) anhand des Index (20 Pages) gelesen. Die Ausgabe wird in sortierter Reihenfolge erstellt.

Durch das Aufteilen einer Abfrage in drei Schritte und dadurch, daß das Sortieren durch indizierten Zugriff ersetzt wurde, wurde folgendes erreicht. Eine Abfrage, die 50.400 Pages liest, in eine Abfrage umgewandelt, die ungefähr 1772 Pages schreibt und liest; dies ist ein Verhältnis von 30 zu 1.

***Hinweis:*** *Frühere Versionen als 4.1 des Informix Datenbankserver erlauben es nicht, die ORDER BY- und INTO TEMP-Klausel in derselben Abfrage zu verwenden. Bei diesen Datenbankservern können Sie die vorangegangene Lösung erhalten, indem Sie INTO TEMP ohne Sortierung verwenden.Um die gewünschte Sortierung zu erreichen, verwenden Sie anschließend einen Cluster-Index. Die Verbesserung beträgt meistens 15 zu 1 anstatt 30 zu 1.*

# Zusammenfassung

Schlechte Performance kann durch SQL-Anweisungen, aber auch durch eine Reihe von anderen Faktoren verursacht werden. Führen Sie die folgenden Schritte durch, bevor Sie sich den SQL-Anweisungen zuwenden:

- Untersuchen Sie die Anwendung in ihrem Zusammenspiel mit den Rechnern, Anwendern, Prozeduren und den Organisationsbereichen.
- Versuchen Sie, genau zu verstehen, was die Anwendung tut, für wen und warum.
- Suchen Sie im Umfeld nach nicht-technischen Lösungen.
- Entwickeln Sie eine Möglichkeit, um wiederholbare, quantitative Messungen der Performance einer Anwendung vornehmen zu können.
- Kristallisieren Sie die zeitaufwendigen Teile soweit wie möglich heraus.

Die Performance einer einzelnen Abfrage wird vom Optimierer bestimmt. Der Optimierer ist ein Teil des Datenbankservers, der einen Abfrageplan aufstellt. Der Optimierer listet alle möglichen Abfragepläne auf und schätzt für jeden den Arbeitsaufwand ab. Er leitet den Abfrageplan mit der niedrigsten Schätzung zur Ausführung an den Datenbankserver weiter.

Die folgenden Operationen sind in einer Abfrage sehr zeitaufwendig:

- Datensätze sequentiell von der Platte lesen; nicht-sequentiell anhand der ROWID; nicht-sequentiell über einen Index
- Datensätze über ein Netzwerk lesen
- Sortieren
- Einige schwierige Abfragen mit regulären Ausdrücken durchführen
- Indizieren und Sortieren von NCHAR und NVARCHAR-Spalten.

Die allgemeine Methode, eine Abfrage schneller zu machen, besteht zum einen darin, den Abfrageplan des Optimierers zu untersuchen; dieser wird von der Funktion SET EXPLAIN ON dokumentiert. Zum anderen besteht eine Möglichkeit darin, die Abfrage so zu verändern, daß der Datenbankserver

- weniger Daten-Pages von der Platte liest
- Pages sequentiell anstatt nicht-sequentiell liest
- das Sortieren vermeidet, weniger Sätze oder kleinere Spalten sortiert.

Auf diese Weise können Sie beachtliche Performance-Vorteile erzielen.

Kapitel

14

# Gespeicherte Prozeduren

# Kapitelüberblick

Mit SQL und einigen zusätzlichen Anweisungen der Stored Procedure Language (SPL) können Sie Prozeduren erzeugen und in der Datenbank speichern. Dieses Kapitel beschreibt, wie solche „gespeicherten Prozeduren" mit SQL und SPL erzeugt werden. Zahlreiche Beispiele für ablauffähige gespeicherte Prozeduren sollen Ihnen die Erstellung eigener gespeicherter Prozeduren erleichtern.

Die Syntaxbeschreibung der SPL-Anweisungen finden Sie in Kapitel 2 des Handbuchs *SQL-Sprachbeschreibung, Syntax*, ergänzt durch Hinweise zur Verwendung und Beispiele zu der jeweiligen Anweisung.

# Einführung in gespeicherte Prozeduren und SPL

Aus der Sicht von SQL ist eine gespeicherte Prozedur eine benutzerdefinierte Funktion. Jeder Benutzer mit RESOURCE-Berechtigung für eine Datenbank kann eine gespeicherte Prozedur erstellen. Ist die Prozedur einmal erstellt, wird sie in ausführbarer Form in der Datenbank als ein Objekt dieser Datenbank gespeichert. Mit einer gespeicherten Prozedur können Sie jede SQL-Funktion durchführen, darüberhinaus aber auch solche Funktionen, die mit SQL allein nicht durchführbar sind.

Gespeicherte Prozeduren enthalten SQL- und SPL-Anweisungen. SPL-Anweisungen können nur in den Anweisungen CREATE PROCEDURE und CREATE PROCEDURE FROM verwendet werden. Die CREATE PROCEDURE-Anweisung ist mit **DB-Access** verfügbar, die Anweisung CREATE PROCEDURE FROM mit **ESQL**-Produkten wie beispielsweise **INFORMIX-ESQL/C** und **INFORMIX-ESQL/COBOL**.

# Vorteile gespeicherter Prozeduren

Mit gespeicherten Prozeduren können Sie z. B. die Datenbank-Performance verbessern, das Schreiben von Anwendungen vereinfachen und den Zugriff auf Daten einschränken bzw. überwachen.

Eine gespeicherte Prozedur können Sie dazu nutzen, sich häufig wiederholende Aufgaben auszuführen, und so die Performance zu erhöhen. Die Verwendung einer gespeicherten Prozedur anstelle von reinem SQL-Code vermeidet wiederholte Syntaxprüfungen, Prüfen auf Gültigkeit und Abfrageoptimieren.

Eine gespeicherte Prozedur ist für jede Anwendung zugänglich. Mehrere Anwendungen können die gleiche Prozedur verwenden, wodurch sich die Entwicklungszeit von Anwendungen verringert.

Sie können eine gespeicherte Prozedur erstellen, die mit DBA-Berechtigung auch von Benutzern ausgeführt werden kann, die eigentlich keine DBA-Berechtigung besitzen. So können Sie den Zugriff auf die Daten in der Datenbank einschränken und steuern. Alternativ kann eine gespeicherte Prozedur überwachen, welche Benutzer auf Tabellen oder Daten zugreifen. Nähere Informationen darüber, wie Sie dies tun können, finden Sie im Abschnitt "Den Zugriff auf eine Datenbank kontrollieren" auf Seite 11-4.

# SQL und gespeicherte Prozeduren

Sie können gespeicherte Prozeduren innerhalb einer SQL-Anweisungen aufrufen. Sie können aber auch innerhalb gespeicherter Prozeduren SQL-Anweisungen ausführen. Nähere Informationen hierüber finden Sie in Kapitel 2 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

Sie rufen gespeicherte Prozeduren innerhalb einer SQL-Anweisung auf, um dieser Anweisung Werte zu übergeben. Beispielsweise können Sie mit einer gespeicherten Prozedur:

- Werte übergeben, die in eine Tabelle eingefügt werden sollen,
- Werte übergeben, die innerhalb der SELECT-, DELETE- oder UPDATE-Anweisung als Teil einer Bedingung verwendet werden.

Dies sind nur zwei Anwendungsbeispiele. Es gibt natürlich noch mehr. So kann jeder Ausdruck innerhalb einer Anweisung ein Aufruf einer gespeicherten Prozedur sein.

Sie können gespeicherte Prozeduren auch dazu verwenden, SQL-Anweisungen zu "verstecken". Eine gespeicherte Prozedur kann die Aufgaben eines Benutzers mit geringer SQL-Erfahrung vereinfachen. So kann ein erfahrener

SQL-Benutzer eine gespeicherte Prozedur schreiben, die anderen Benutzer müssen lediglich wissen, daß die Prozedur in der Datenbank existiert und daß sie sie ausführen können.

# Gespeicherte Prozeduren erzeugen und verwenden

Um eine gespeicherte Prozedur zu erzeugen, müssen Sie die SQL-Anweisungen, die als Teil der Prozedur ausgeführt werden sollen, in den Anweisungsblock einer CREATE PROCEDURE-Anweisung stellen. Mit SPL-Anweisungen wie IF, FOR und anderen können Sie den Prozedurablauf steuern. Die Beschreibung dieser Anweisungen finden Sie in Kapitel 2 des Handbuchs *SQL-Sprachbeschreibung, Syntax*. Dort finden Sie auch die Beschreibungen der SQL-Anweisungen.

## Eine Prozedur mit DB-Access erzeugen

Zum Erstellen einer Prozedur mit **DB-Access** müssen Sie eine CREATE PROCEDURE-Anweisung ausführen, die im Anweisungsblock alle Anweisungen enthält, die Teil der Prozedur sein sollen. Bild 14-1 zeigt, wie eine Prozedur erstellt wird, die eine Kundenadresse ausliest.

```
CREATE PROCEDURE read_address (lastname CHAR(15)) -- one argument
    RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),CHAR(2), CHAR(5); -- 6 items

    DEFINE p_lname,p_fname, p_city CHAR(15); --define each procedure variable
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);

    SELECT fname, address1, city, state, zipcode
        INTO p_fname, p_add, p_city, p_state, p_zip
        FROM customer
        WHERE lname = lastname;

    RETURN p_fname, lastname, p_add, p_city, p_state, p_zip; --6 items
END PROCEDURE

DOCUMENT 'This procedure takes the last name of a customer as', --brief description
    'its only argument. It returns the full name and address of',
    'the customer.'
WITH LISTING IN '/acctng/test/listfile' -- compile-time warnings go here
; -- end of the procedure read_address
```

*Bild 14-1* ***Prozedur, die aus der Tabelle*** **customer** ***liest***

# Eine Prozedur mit einem SQL API-Produkt erzeugen

Um eine gespeicherte Prozedur mit einem SQL API-Produkt zu erstellen, müssen Sie die CREATE PROCEDURE-Anweisung in eine Datei schreiben. Verwenden Sie die CREATE PROCEDURE FROM-Anweisung und verweisen Sie beim Compilieren der Prozedur auf diese Datei. Wenn Sie beispielsweise eine Prozedur erstellen wollen, die einen Kundennamen liest, dann können Sie eine Anweisung wie in Bild 14-1 verwenden und in einer Datei speichern. Wenn die Datei z. B. **read_add_source** heißt, dann compiliert die folgende Anweisung die darin enthaltene Prozedur **read_address**:

```
CREATE PROCEDURE FROM 'read_add_source';
```

Bild 14-2 zeigt, wie die vorhergehende SQL-Anweisung als **ESQL/C**-Programm geschrieben werden kann.

```
/* This program creates whatever procedure is in *
 * the file 'read_add_source'.
 */
#include <stdio.h>
$include sqlca;
$include sqlda;
$include datetime;
/* Program to create a procedure from the pwd */

main()
{
$database play;
$create procedure from 'read_add_source';
}
```

*Bild 14-2* *SQL-Anweisung zum Compilieren und Speichern der Prozedur* **read_address** **(ESQL/C)**

# Kommentieren und Dokumentieren einer Prozedur

Die Prozedur **read_address** in Bild 14-1 enthält Kommentare und eine DOCUMENT-Klausel. Die Kommentare werden in den Text der Prozedur aufgenommen. Kommentare werden stets durch zwei Bindestriche (--) eingeleitet, die beliebig am Zeilenanfang oder auch in der Mitte einer Zeile stehen können.

Mit der DOCUMENT-Klausel können Sie die Prozedur dokumentieren. Den Dokumentationstext können Sie abrufen, indem Sie die Systemtabelle **sysprocbody** abfragen. Der Abschnitt "Prozedurdokumentation lesen" auf Seite 14-9 enthält weitere Informationen über die Verwendung der DOCUMENT-Klausel.

# Fehler zur Compilier-Zeit ermitteln

Wenn Sie eine der Anweisungen CREATE PROCEDURE oder CREATE PROCEDURE FROM ausführen und der Prozedurtext einen Syntaxfehler enthält, dann mißlingt die Anweisung. Der Datenbankserver bricht die Verarbeitung des Textes ab und gibt die Stelle an, an der der Fehler aufgetreten ist.

## Syntaxfehler in einer Prozedur unter DB-Access ermitteln

Wenn eine unter **DB-Access** erstellte Prozedur einen Syntaxfehler enthält und Sie die Menüoption "Modify" des Menüs SQL wählen, dann wird die Schreibmarke an die fehlerhafte Stelle gesetzt.

## Syntaxfehler in einer Prozedur bei einem SQL API-Produkt ermitteln

Wenn eine mit einem der SQL API-Produkte erstellte Prozedur einen Syntaxfehler enthält, dann mißlingt die CREATE PROCEDURE-Anweisung. Der Datenbankserver trägt in das Feld SQLCODE von SQLCA eine negative Zahl und in das fünfte Element des Arrays SQLERRD die Zeichenposition in der Datei ein. Die folgende Tabelle enthält die genauen Bezeichnungen der Felder der Struktur SQLCA für jedes Produkt:

| ESQL/C | ESQL/FORTRAN | ESQL/COBOL |
|---|---|---|
| sqlca.sqlcode<br>SQLCODE | sqlcod | SQLCODE OF SQLCA |
| sqlca.sqlerrd[4] | sqlca.sqlerr(5) | SQLERRD[5] OF SQLCA |

Bild 14-3 zeigt, wie beim Erstellen einer Prozedur Syntaxfehler abgefangen werden können und wie man die Zeichenposition des Fehlers in der Datei anzeigt:

```
#include <stdio.h>
$include sqlca;
$include sqlda;
$include datetime;
/* Program to create a procedure from procfile in pwd */

main()
{
long char_num;

$ database play;
$create procedure from 'procfile';
if (sqlca.sqlcode != 0 )
        {
        printf('\nSqlca.sqlcode = %ld\n', sqlca.sqlcode);
        char_num = sqlca.sqlerrd[4];
      printf('\nError in creating read_address. Check character position %ld\n',
                        char_num);
}
.
.
.
```

*Bild 14-3* ***Fehlerprüfung beim Erstellen einer Prozedur in*** **ESQL/C**

Wenn die CREATE PROCEDURE FROM-Anweisung in Bild 14-3 mißlingt, dann gibt das Programm eine Meldung und die Zeichenposition aus, an der der Fehler aufgetreten ist.

# Berücksichtigen von Warnungen zur Compilier-Zeit

Wenn die Prozedur zwar syntaktisch korrekt ist, der Datenbankserver aber ein potentielles Problem feststellt, schreibt er eine Warnung in eine Datei. Anhand dieser Datei können Sie potentielle Probleme in der Prozedur abprüfen, bevor Sie sie ausführen.

Mit der Klausel WITH LISTING IN in der CREATE PROCEDURE-Anweisung geben Sie an, in welche Datei die Liste der Warnungen geschrieben werden soll, wie im folgenden Beispiel gezeigt.

```
CREATE PROCEDURE read_address (lastname CHAR(15)) -- one argument
    RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15), CHAR(2), CHAR(5); -- 6 items
    .
    .
    .
    WITH LISTING IN '/acctng/test/listfile' -- compile-time warnings go here
; -- end of the procedure read_address
```

Wenn Sie innerhalb eines Netzes arbeiten, dann wird die Datei mit der Liste der Warnungen auf dem Rechner angelegt, auf dem sich die Datenbank befindet. Wenn Sie einen absoluten Pfadnamen für die Datei angeben, dann wird die Datei an der angegebenen Stelle angelegt. Wenn Sie einen relativen Pfadnamen für die Datei angeben, dann wird die Datei in Ihrem Home-Dateiverzeichnis angelegt. (Wenn Sie kein Home-Dateiverzeichnis besitzen, dann wird die Datei im Root-Dateiverzeichnis angelegt.)

Nach dem Erstellen der Prozedur können Sie anhand der in der Klausel WITH LISTING IN angegebenen Datei feststellen, welche Warnungen darin aufgetreten sind.

# Text bzw. Dokumentation lesen

Eine gespeicherte Prozedur wird in der Systemtabelle **sysprocbody** gespeichert. Diese Systemtabelle enthält nicht nur die ausführbare Prozedur, sondern auch den Text der ursprünglichen CREATE PROCEDURE-Anweisung sowie die Dokumentation zur Prozedur.

## Prozedurtext lesen

Zum Lesen des Textes der Prozedur wählen Sie die Spalte **data** in der Systemtabelle **sysprocbody**. Die folgende SELECT-Anweisung liest den Text der Prozedur **read_address**.

```
SELECT data FROM informix.sysprocbody
    WHERE datakey = 'T'      -- find text lines
    AND
    procid = (SELECT procid FROM informix.sysprocedures
        WHERE informix.sysprocedures.procname = 'read_address')
```

## Prozedurdokumentation lesen

Mit der folgenden SELECT-Anweisung rufen Sie die Zeilen aus der DOCUMENT-Klausel der CREATE PROCEDURE-Anweisung ab:

```
SELECT data FROM informix.sysprocbody
    WHERE datakey = 'D'      -- find documentation lines
    AND
    procid = (SELECT procid FROM informix.sysprocedures
        WHERE informix.sysprocedures.procname = 'read_address')
```

# Ausführen einer Prozedur

Sie haben mehrere Möglichkeiten, eine Prozedur auszuführen: mit der SQL-Anweisung EXECUTE PROCEDURE oder mit einer der SPL-Anweisungen LET oder CALL. Darüberhinaus können Sie Prozeduren auch dynamisch ausführen, wie im Abschnitt "Gespeicherte Prozeduren dynamisch ausführen" auf Seite 14-12 gezeigt.

Die Prozedur **read_address** ermittelt den vollständigen Namen und die Adresse des Kunden mit dem Namen „Putnum". Diese Prozedur können Sie mit der folgenden Anweisung EXECUTE PROCEDURE ausführen lassen:

```
EXECUTE PROCEDURE read_address ('Putnum');
```

Die Prozedur **read_address** liefert Werte zurück. Daher müssen Sie, wenn die Prozedur **read_address** in einem **ESQL**- oder einem anderen Programm aufgerufen wird, mit der INTO-Klausel Host-Variablen bereitstellen, die die zurückgelieferten Werte aufnehmen. Bild 14-4 zeigt, wie die Prozedur **read_address** in einem **ESQL/C**-Programm ausgeführt wird:

```
#include <stdio.h>
$include sqlca;
$include sqlda;
$include datetime;
/* Program to execute a procedure in the database named "play" */

main()
{
$ char lname[16], fname[16], address[21];
$ char city[16], state[3], zip[6];

$ connect to play;
$EXECUTE PROCEDURE read_address ('Putnum')
    INTO $lname, $fname, $address, $city, $state, $zip;
if (sqlca.sqlcode != 0 )
     printf("\nFailure on execute");
}
```

*Bild 14-4* *Ausführen einer Prozedur in ESQL/C*

Wenn Sie eine Prozedur innerhalb einer anderen Prozedur aufrufen, dann verwenden Sie eine CALL- oder LET-Anweisung zum Ausführen der Prozedur . Bild 14-5 zeigt, wie Sie die Prozedur **read_address** mit einer CALL-Anweisung aufrufen :

```
CREATE PROCEDURE address_list ()

    DEFINE p_lname, p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    .
    .
    .
    CALL read_address ('Putnum') RETURNING p_fname, p_lname,
        p_add, p_city, p_state, p_zip;
    .
    .
    .
    -- use the returned data some way
END PROCEDURE;
```

*Bild 14-5* ***Prozeduraufruf innerhalb einer anderen Prozedur mit der CALL-Anweisung***

Bild 14-6 zeigt, wie mit der LET-Anweisung über einen Prozeduraufruf Werte an Prozedurvariablen übergeben werden:

```
CREATE PROCEDURE address_list ()

    DEFINE p_lname, p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    .
    .
    .
    LET p_fname, p_lname,p_add, p_city, p_state, p_zip = read_address ('Putnum');
    .
    .
    .
    -- use the returned data some way
END PROCEDURE;
```

*Bild 14-6* ***Übergeben von Werten aus einem Prozeduraufruf mit einer LET-Anweisung***

## Gespeicherte Prozeduren dynamisch ausführen

Mit den Anweisungen ALLOCATE DESCRIPTOR und GET DESCRIPTOR können Sie in einem **ESQL/C**-Programm eine EXECUTE PROCEDURE-Anweisung aufbereiten. Aufrufargumente können zur Lauf- wie zur Compilier-Zeit übergeben werden. Nähere Informationen zum dynamischen Ausführen gespeicherter Prozeduren finden Sie im Handbuch *INFORMIX-ESQL/C Programmer's Manual*. Nähere Informationen über dynamisches SQL und über die Verwendung aufbereiteter SELECT-Anweisungen finden Sie in Kapitel 5 "SQL in Programmen".

## Prozedurfehler beheben

Treten in der Prozedurlogik Fehler auf, können Sie mit der TRACE-Anweisung den Ablauf der Prozedur überprüfen und die Werte der folgenden Prozedur-Objekte verfolgen:

- Variablen
- Prozedurargumente
- Rückgabewerte
- SQL-Fehlercodes
- ISAM-Fehlercodes

Mit der SQL-Anweisung SET DEBUG FILE geben Sie zunächst den Namen der Datei an, in die die Liste der verfolgten Werte geschrieben werden soll. Beim Erstellen der Prozedur geben Sie dann die TRACE-Anweisung an.

Sie haben drei Möglichkeiten, die Art der TRACE-Ausgabe festzulegen:

| | |
|---|---|
| TRACE ON | verfolgt den Ablauf aller Anweisungen außer den der SQL-Anweisungen. Variablenwerte werden ausgegeben, bevor sie benutzt werden. Prozeduraufrufe und Rückgabewerte werden in die Ablaufverfolgung eingeschlossen |
| TRACE PROCEDURE | nur Prozeduraufrufe und Rückgabewerte werden in die Ablaufverfolgung eingeschlossen |
| TRACE *ausdruck* | gibt ein Literal oder einen Ausdruck aus. Falls erforderlich, wird der Wert des Ausdrucks berechnet, bevor er an die Datei übertragen wird. |

Bild 14-7 zeigt, wie die TRACE-Anweisung mit einer Version der Prozedur **read_address** verwendet werden kann. Zahlreiche der in diesem Beispiel verwendeten SPL-Anweisungen wurden noch nicht besprochen, aber das gesamte Beispiel verdeutlicht, wie mit der TRACE-Anweisung die Ausführung der Prozedur überwacht werden kann.

```
CREATE PROCEDURE read_many  (lastname CHAR(15))
    RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),CHAR(2), CHAR(5);

    DEFINE p_lname,p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    DEFINE lcount, i INT;

    LET lcount = 1;

    TRACE ON; -- Every expression will be traced from here on
    TRACE 'Foreach starts';          -- A trace statement with a literal
    FOREACH
    SELECT fname, lname, address1, city, state, zipcode
        INTO p_fname, p_lname, p_add, p_city, p_state, p_zip
        FROM customer
        WHERE lname = lastname
    RETURN p_fname, p_lname, p_add, p_city, p_state, p_zip WITH RESUME;
    LET lcount = lcount + 1;  -- count of returned addresses
    END FOREACH;

    TRACE 'Loop starts'; -- Another literal
    FOR i IN (1 TO 5)
        BEGIN
          RETURN i , i+1, i*i, i/i, i-1,i with resume;
        END
    END FOR;

END PROCEDURE

;
```

*Bild 14-7* ***Mit der TRACE-Anweisung die Ausführung einer Prozedur überwachen***

Bei jedem Ausführen einer Prozedur mit eingeschalteter Ablaufverfolgung werden weitere Einträge in die Datei aufgenommen, die in der SET DEBUG FILE-Anweisung angegeben ist. Sie können diese Einträge mit jedem beliebigen Editor lesen.

Im folgenden werden auszugsweise TRACE-Ausgaben aufgelistet, die von der in Bild 14-7 abgedruckten Prozedur erzeugt wurden.

| TRACE-Ausgabe: | Erläuterung: |
|---|---|
| `TRACE ON` | Einschalten der Ablaufverfolgung |
| `TRACE Foreach starts` | Angabe des zu überwachenden Ausdruck, hier: die Zeichenkette `Foreach starts` |
| `start select cursor` | Hinweis, daß ein Cursor für eine FOREACH-Schleife geöffnet wird |
| `select cursor iteration` | Hinweis über den Beginn eines jeden Schleifendurchlaufs des Select-Cursors |
| `expression:(+lcount, 1)` | Der angegebene Ausdruck `(lcount+1)` ergibt 2 |
| `let lcount = 2` | Jede LET-Anweisung wird mit dem zugewiesenen Wert ausgegeben. |

# Eine Prozedur erneut erstellen

Eine gespeicherte Prozedur müssen Sie explizit mit einer DROP PROCEDURE-Anweisung löschen, bevor Sie eine andere Prozedur mit gleichem Namen erstellen können. Beim Testen einer Prozedur können Sie die CREATE PROCEDURE-Anweisung mit dem gleichen Prozedurnamen erst dann wieder ausführen, nachdem Sie die bestehende Prozedur aus der Datenbank gelöscht haben.

# Berechtigungen für gespeicherte Prozeduren

Eine Prozedur wird in der Datenbank gespeichert, in der sie erzeugt wurde. Da es sich bei einer Prozedur um ein Datenbankobjekt handelt, werden zum Erstellen und Ausführen einer Prozedur bestimmte Berechtigungen benötigt.

Es gibt zwei Arten von Prozeduren, die beim Erzeugen der Prozedur festgelegt werden: Prozeduren mit DBA-Berechtigung und Prozeduren mit Eigentümer-Berechtigung. Je nachdem, welche Art gespeicherte Prozedur erzeugt wurde, unterscheiden sich die benötigten Berechtigungen. In der folgenden Übersicht werden diese Unterschiede erläutert:

| **Erläuterungen:** | **Prozedur mit DBA-Berechtigung** | **Prozedur mit Eigentümer-Berechtigung** |
|---|---|---|
| Darf erzeugt werden mit: | DBA Berechtigung | mindestens RESOURCE-Berechtigung |
| Voreinstellung für EXECUTE-Berechtigung: | jede(r) mit DBA-Berechtigung | ohne ANSI-Kompatibilität: PUBLIC (Jede(r) mit CONNECT-Berechtigung )<br><br>mit ANSI-Kompatibilität: Prozedur-Eigentümer und jede(r) mit DBA-Berechtigung |
| Berechtigungen, die der Prozedur-Eigentümer oder DBA anderen erteilen muß, um ihnen das Ausführen der Prozedur zu erlauben | EXECUTE-Berechtigung | ohne WITH GRANT OPTION: EXECUTE-Berechtigung sowie Berechtigungen für zugrundeliegende Datenbankobjekte<br><br>mit WITH GRANT OPTION: nur EXECUTE-Berechtigung |

*Bild 14-8* ***Unterschiede zwischen Prozeduren mit DBA-Berechtigung bzw. Eigentümer-Berechtigung***

# Berechtigungen zum Erzeugen

Zum Erstellen einer Prozedur mit DBA-Berechtigung benötigen Sie die DBA-Berechtigung für die Datenbank.

Zum Erstellen einer Prozedur mit Eigentümer-Berechtigung müssen Sie zumindest die RESOURCE-Berechtigung für die Datenbank besitzen.

Nähere Informationen hierzu finden Sie auch in Kapitel 11 "Zugriff auf Datenbaänken regeln".

# Berechtigungen zum Ausführ-Zeitpunkt

Eine Prozedur kann nur mit EXECUTE-Berechtigung oder DBA-Berechtigung ausgeführt werden. Abhängig davon, ob die Prozedur mit DBA-Berechtigung erstellt wurde und ob die Datenbank ANSI-kompatibel ist oder nicht, erteilt der Datenbankserver implizit bestimmte Berechtigungen.

Für Prozeduren mit Eigentümer-Berechtigung wird die EXECUTE-Berechtigung PUBLIC erteilt.Bei ANSI-Datenbanken wird die EXECUTE-Berechtigung nur dem Prozedureigentümer erteilt, sowie Benutzern mit DBA-Berechtigung.

Für Prozeduren mit DBA-Berechtigung wird die EXECUTE-Berechtigung nur Benutzern mit DBA-Berechtigung erteilt.

## Prozeduren mit Eigentümer-Berechtigung

Wird eine Prozedur mit Eigentümer-Berechtigung ausgeführt, dann wird überprüft, ob die darin angesprochenen Datenbankobjekte vorhanden sind. Darüber hinaus wird geprüft, ob die erforderlichen Berechtigungen für diese Objekte erteilt sind.

Wenn Sie eine Prozedur ausführen, die ausschließlich auf Ihre eigenen Objekte verweist, können keine Berechtigungen verletzt werden. Wenn Sie nicht Eigentümer der angesprochenen Objekte sind und eine Prozedur ausführen, die beispielsweise SELECT-Anweisungen enthält, so benötigen Sie EXECUTE-Berechtigung für die entsprechenden Tabellen. Sonst kann die Prozedur nicht fehlerfrei ablaufen .

Die erforderlichen Berechtigungen können auch mit der WITH GRANT OPTION-Anweisung weitergegeben werden.

Im Verlauf der Prozedur erzeugte Objekte, die nicht mit dem Namen eines Eigentümers gekennzeichnet sind, werden dem Eigentümer der Prozedur zugeordnet und nicht dem Benutzer, der die Prozedur ausführt. Beispielsweise

werden im folgenden Beispiel in einer Prozedur mit Eigentümer-Berechtigung zwei Tabellen erzeugt. Wenn **tony** der Eigentümer der Prozedur ist und **marty** der Benutzer, der sie ausführt, dann ist die erste Tabelle, **gargantuan**, das Eigentum von **tony**. Eigentümerin der zweiten Tabelle, **tiny**, ist **libby**.

```
CREATE PROCEDURE tryit()
    .
    .
    .
    CREATE TABLE gargantuan (col1 INT, col2 INT, col3 INT);
    CREATE TABLE libby.tiny (col1 INT, col2 INT, col3 INT);

END PROCEDURE
```

### Prozeduren mit DBA-Berechtigung

Wird eine Prozedur mit DBA-Berechtigung ausgeführt, so erhält der die Prozedur ausführende Benutzer DBA-Berechtigung, solange die Prozedur ausgeführt wird. Eine Prozedur mit DBA-Berechtigung verhält sich so, als ob der aktuelle Benutzer zuerst DBA-Berechtigung erhält, dann jede Anweisung der Prozedur einzeln ausführt und schließlich die DBA-Berechtigung wieder verliert.

Objekte, die im Verlauf einer Prozedur mit DBA-Berechtigung erzeugt werden, sind Eigentum des Benutzers, der die Prozedur ausführt. Anweisungen zur Datendefinition in der Prozedur können allerdings explizit einen anderen Eigentümer festlegen.

### Berechtigungen und geschachtelte Prozeduren

Die DBA-Berechtigung wird von einer aufgerufenen Prozedur nicht geerbt. Wenn eine Prozedur mit DBA-Berechtigung eine solche mit Eigentümer-Berechtigung aufruft, so erhält die aufgerufene Prozedur nicht den Status einer Prozedur mit DBA-Berechtigung. Wenn eine Prozedur mit Eigentümer-Berechtigung eine solche mit DBA-Berechtigung aufruft, dann werden die Anweisungen in der Prozedur mit DBA-Berechtigung genauso ausgeführt wie in jeder anderen Prozedur mit DBA-Berechtigung.

## Berechtigungen entziehen

Der Eigentümer einer Prozedur kann einem Benutzer die EXECUTE-Berechtigung entziehen. Wenn ein Benutzer die EXECUTE-Berechtigung verliert, dann wird es auch allen anderen Benutzern entzogen, die dieses Recht von jenem Benutzer erhalten haben.

# Variablen und Ausdrücke

Der vorliegende Abschnitt beschreibt Definition und Verwendung von Variablen in SPL. Darüberhinaus werden die Unterschiede zwischen SPL- und SQL-Ausdrücken aufgezeigt.

## Variablen

In einer gespeicherten Prozedur können Sie Variablen auf verschiedene Arten verwenden. Eine Variable kann in einer Datenbankabfrage oder in einer anderen SQL-Anweisung an allen Stellen verwendet werden, an denen eine Konstante stehen darf. Mit Variablen können Sie in SPL-Anweisungen Werte zuweisen und berechnen, die Anzahl der von einer Abfrage zurückgegebenen Datensätze ermitteln, eine Schleife ausführen und vieles mehr.

Der Wert einer Variablen wird gespeichert. Da die Variable kein Datenbankobjekt ist, werden durch das Zurücksetzen einer Transaktion nicht gleichzeitig die Werte der Prozedurvariablen wiederhergestellt.

### Formate von Variablen

Eine Variable muß den gültigen Regeln genügen. Diese Regeln sind im Handbuch *SQL-Sprachbeschreibung, Syntax* beschrieben. Einmal definiert, können Sie die Variable an jeder Stelle in der Prozedur verwenden.

Wenn Sie eines der **ESQL**-Produkte verwenden, müssen Sie die Variable (im Gegensatz zu Host-Variablen) nicht mit einem Sonderzeichen kennzeichnen.

### Globale und lokale Variablen

Sie können eine Variable entweder global oder lokal vereinbaren. Standardmäßig ist eine Variable lokal. Die folgende Aufstellung verdeutlicht die Unterschiede zwischen den beiden Typen:

| | |
|---|---|
| Lokal | lokale Variablen sind ausschließlich in der Prozedur verfügbar, in der sie definiert wurden. Lokalen Variablen kann zur Compilier-Zeit kein Standardwert zugewiesen werden. |
| Global | globale Variablen sind für andere Prozeduren verfügbar, die während derselben Sitzung in der gleichen Datenbank ausgeführt werden. Die Werte globaler Variablen werden global gespeichert, so daß alle Prozeduren auf sie zugreifen können, die in einer bestimmten Sitzung auf einem bestimmten Datenbankserver laufen. Dazu gehören beispielsweise alle Prozeduren, die von einem **ESQL**-Programm oder in einer |

DB-Access-Sitzung aufgerufen werden. Die Werte der Variablen gehen am Ende der Sitzung verloren.

Globalen Variablen muß zur Compilier-Zeit ein Standardwert zugewiesen werden.

Die erste Definition einer globalen Variable macht diese Variable global bekannt. Nachfolgende Definitionen derselben Variablen in anderen Prozeduren binden die Variable einfach an die globale Definition.

## Definieren von Variablen

Mit der DEFINE-Anweisung definieren Sie Variablen. Geben Sie jedoch eine Variable in der Argumentliste einer Prozedur an, so wird die Variable implizit definiert und es ist keine DEFINE-Anweisung erforderlich. Bevor Sie eine Variable verwenden, müssen Sie ihr einen Wert zuweisen. Es kann dies auch der NULL-Wert sein.

## Datentypen für Variablen

Außer SERIAL kann eine Variable jeden der Datentypen annehmen, die für die Spalten in einer Tabelle verfügbar sind. Das folgende Beispiel zeigt einige Definitionen für Prozedurvariablen:

```
DEFINE x INT;
DEFINE name CHAR(15);
DEFINE this_day DATETIME YEAR TO DAY ;
```

Wenn Sie den Datentyp einer Variablen als TEXT oder BYTE festlegen, dann enthält die Variable in Wirklichkeit nicht die Daten selbst, sondern vielmehr einen Zeiger auf die Daten. Sie verwenden diese Variable jedoch wie jede andere Prozedurvariable. Wenn Sie eine Variable vom Typ TEXT oder BYTE definieren, dann müssen Sie das Schlüsselwort REFERENCES verwenden. Dadurch wird deutlich gemacht, daß diese Variablen nicht die Daten, sondern lediglich einen Verweis auf diese Daten enthalten. Das folgende Beispiel zeigt die Definition einer TEXT- und einer BYTE-Variablen:

```
DEFINE ttt REFERENCES TEXT;
DEFINE bbb REFERENCES BYTE;
```

## Variablen subskribieren

Sie können eine Variable ebenso subskribieren wie in SQL einen Spaltennamen; Voraussetzung hierfür ist allerdings, daß die Variable von einem der Datentypen CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE oder TEXT ist. Subskripte markieren Anfangs- und Endezeichen einer Variablen und sind stets Konstanten. Variablen dürfen nicht als Subskripte verwendet werden. Die Verwendung wird im folgenden Beispiel verdeutlicht:

```
DEFINE name CHAR(15);
LET name[4,7] = 'Ream';
SELECT fname[1,3] INTO name[1,3] FROM customer
    WHERE lname = 'Ream';
```

Der durch die beiden Subskripte begrenzte Teil des Inhalts der Variablen wird als Teilzeichenkette bezeichnet.

## Gültigkeitsbereich von Variablen

Eine Variable ist innerhalb des Anweisungsblocks gültig, in dem sie definiert wurde. Sie ist auch in einem geschachtelten Anweisungsblock innerhalb dieses Anweisungsblocks gültig, sofern sie nicht von der Definition einer anderen Variablen mit gleichem Namen in dem geschachtelten Anweisungsblock ersetzt wird.

Am Anfang der in Bild 14-9 gezeigten Prozedur werden die ganzzahligen Variablen `x`, `y` und `z` definiert und initialisiert. Die Anweisungen BEGIN und END kennzeichnen den Anweisungsblock, in dem die ganzzahligen Variablen `x` und `q` sowie die CHAR-Variable `z` definiert sind. In diesem geschachtelten Anweisungsblock ersetzt die erneute Definition der Variablen `x` die

ursprüngliche Definition der Variablen x. Nach der END-Anweisung, die das Ende des geschachtelten Anweisungsblocks markiert, ist der ursprüngliche Wert der Variablen x wieder verfügbar.

```
CREATE PROCEDURE scope()
    DEFINE x,y,z INT;
    LET x = 5; LET y = 10;
    LET z = x + y; --z is 15
    BEGIN
        DEFINE x, q INT; DEFINE z CHAR(5);
        LET x = 100;
        LET q = x + y; -- q = 110
        LET z = "silly"; -- z receives a character value
    END
    LET y = x; -- y is now 5
    LET x = z; -- z is now 15, not "silly"
END PROCEDURE
```

*Bild 14-9* ***Ersetzen von Variablen in geschachtelten Anweisungsblöcken***

## Mehrdeutigkeiten von Variablen und Schlüsselwörtern

Wenn Sie einer Variablen den Namen eines Schlüsselworts geben, dann können Mehrdeutigkeiten auftreten. Die folgenden Regeln sollen Ihnen dabei helfen, Mehrdeutigkeiten bei Variablen, Prozedurnamen und Namen von Systemfunktionen zu vermeiden.

- Definierte Variablen haben den größten Vorrang.
- Prozeduren, die als solche in einer DEFINE-Anweisung definiert wurden, haben Vorrang über SQL-Funktionen.
- SQL-Funktionen haben Vorrang über Prozeduren, die existieren, aber *nicht* als Prozeduren angesprochen werden (unter Verwendung der DEFINE-Anweisung).

In manchen Fällen müssen Sie den Namen der Variablen ändern. Beispielsweise ist es nicht möglich, eine Variable mit dem Namen **count** oder **max** zu definieren, da dies die Namen von Mengenfunktionen sind. In der Beschreibung des Syntaxelements "Bezeichner" in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* finden Sie eine Liste der Schlüsselwörter, die zu Mehrdeutigkeiten führen können.

## Variablen und Spaltennamen

Wenn Sie für den Namen einer Prozedurvariablen denselben Namen wie für einen Spaltennamen verwenden, dann nimmt der Datenbankserver bei jedem Auftreten dieses Namens an, daß es sich um die Variable handelt. Kennzeichnen Sie den Spaltennamen mit dem Namen der Tabelle, wenn sich der Name auf die Spalte beziehen soll. Im folgenden Beispiel dient **lname** sowohl als Prozedur- als auch als Spaltenname. In der SELECT-Anweisung ist **customer.lname** ein Spaltenname und **lname** ein Variablenname:

```
CREATE PROCEDURE table_test()

    DEFINE lname CHAR(15);
    LET lname = 'Miller';
.
.
.
    SELECT customer.lname FROM customer INTO lname
        WHERE customer_num = 502;
.
.
.
```

## Variablen und SQL-Funktionen

Wenn Sie für eine Prozedurvariable denselben Namen wie für eine SQL-Funktion verwenden, dann nimmt der Datenbankserver bei jedem Auftreten dieses Namens an, daß es sich um die Variable handelt. Sie können die SQL-Funktion nicht in dem Programmblock verwenden, in dem die Variable definiert ist. Das folgende Beispiel zeigt einen Block in einer Prozedur, in dem eine Variable mit dem Namen **user** definiert ist. Diese Definition verhindert die Verwendung der Funktion USER innerhalb des BEGIN...END-Blocks:

```
CREATE PROCEDURE user_test()
    DEFINE name CHAR(10);
    DEFINE name2 CHAR(10);
    LET name = user; -- the SQL function

    BEGIN
    DEFINE user CHAR(15); -- disables user function
    LET user = 'Miller';
    LET name = user; -- assigns 'Miller' to variable name

    END
    .
    .
    .
    LET name2 = user; -- SQL function again
```

### Prozedurnamen und SQL-Funktionen

In der Beschreibung des Syntaxelements "Prozedurname" in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* finden Sie nähere Informationen darüber, wie Sie Verwechselungen von Prozedurnamen mit den Namen von SQL-Funktionen vermeiden können.

## SPL-Ausdrücke

In einer gespeicherten Prozedur können Sie einen beliebigen SQL-Ausdruck verwenden, abgesehen von Ausdrücken mit Mengenfuntionen. In der Beschreibung des Syntaxelements "Ausdruck" in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax* finden Sie die vollständige Beschreibung der Syntax von SQL-Ausdrücken.

Das folgende Beispiel zeigt einige typische SPL-Ausdrücke:

```
var1
var1 + var2 + 5
read_address('Miller')
read_address(lastname = 'Miller')
get_duedate(acct_num) + 10 UNITS DAY
fname[1,5] || ''|| lname
'(415)' || get_phonenum(cust_name)
```

## Werte an Variablen zuweisen

Es gibt vier Möglichkeiten, Werte an eine Prozedurvariable zuzuweisen:

- Verwenden einer LET-Anweisung
- Verwenden einer SELECT...INTO-Anweisung
- Verwenden einer CALL-Anweisung mit einer Prozedur, die eine RETURNING-Klausel enthält
- Verwenden einer EXECUTE PROCEDURE...INTO-Anweisung

Mit der LET-Anweisung weisen Sie einen Wert einer oder mehrere Variablen zu. Das folgende Beispiel zeigt verschiedene Formen der LET-Anweisung:

```
LET a = b + a;
LET a, b = c, d;
LET a, b = (SELECT fname, lname FROM customer
                WHERE customer_num = 101);
LET a, b = read_name(101);
```

Mit der SELECT-Anweisung können Sie einen Wert aus einer Datenbank direkt einer Variablen zuweisen. Die Anweisung im folgenden Beispiel hat dieselbe Auswirkung wie die dritte LET-Anweisung im vorhergehenden Beispiel:

```
SELECT fname, lname into a, b FROM customer
    WHERE customer_num = 101
```

Mit den Anweisungen CALL- oder EXECUTE PROCEDURE können Sie Werte, die von einer Prozedur zurückgegeben werden, einer oder mehreren Prozedurvariablen zuweisen. Beide Anweisungen weisen die vollständige Adresse, die von der Prozedur **read_address** zurückgegeben wird, den angegebenen Prozedurvariablen zu:

```
EXECUTE PROCEDURE read_address('Smith')
    INTO p_fname, p_lname, p_add, p_city, p_state, p_zip;

CALL read_address('Smith')
    RETURNING p_fname, p_lname, p_add, p_city, p_state, p_zip;
```

# Programmablauf-Kontrolle

SPL enthält zahlreiche Anweisungen, mit denen Sie den Programmablauf in Ihrer gespeicherten Prozedur regeln und auf der Grundlage der abgerufenen Daten zur Laufzeit Entscheidungen treffen können. Die Anweisungen zur Programmablauf-Kontrolle werden in diesem Abschnitt kurz beschrieben. Die Syntax dieser Anweisungen und ausführliche Beschreibungen finden Sie in Kapitel 2 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

## Verzweigungen

Mit einer IF-Anweisung formulieren Sie in einer gespeicherten Prozedur eine logische Verzweigung. Eine IF-Anweisung wertet zunächst eine Bedingung aus. Ist die Bedingung wahr, so führt sie den THEN-Zweig der Anweisung aus. Ist die Bedingung nicht wahr, dann führt das Programm die darauffolgenden Anweisung aus, sofern nicht eine der Klauseln ELSE oder ELIF *(else if)* in der IF-Anweisung angegeben ist. Bild 14-10 zeigt das Beispiel einer IF-Anweisung.

```
CREATE PROCEDURE str_compare (str1 CHAR(20), str2 CHAR(20))
    RETURNING INT;
    DEFINE result INT;

    IF str1 > str2 THEN
        result = 1;
    ELIF str2 > str1 THEN
        result = -1;
    ELSE
        result = 0;
    END IF
    RETURN result;
END PROCEDURE -- str_compare
```

*Bild 14-10 IF-Anweisung*

# Schleifen

Es gibt in SPL drei Arten von Schleifen, die mit jeweils einer der folgenden Anweisungen eingeleitet werden:

| Schleife: | Erläuterung: |
|---|---|
| FOR | beginnt eine kontrollierte Schleife; Endlosschleifen sind ausgeschlossen. |
| FOREACH | ermöglicht die Auswahl und Bearbeitung mehr als eines Satzes aus der Datenbank. Die Anweisung deklariert und öffnet implizit einen Cursor. |
| WHILE | beginnt eine Schleife; Endlosschleifen sind *nicht* ausgeschlossen. |

Mit einer der folgenden vier Anweisungen kann eine Schleife verlassen werden:

| Aussprung: | Erläuterung: |
|---|---|
| CONTINUE | überspringt die verbleibenden Anweisungen in der aktuellen Schleife und beginnt mit der Ausführung eines neuen Schleifendurchlaufs. |
| EXIT | verläßt die aktuelle Schleife. Die Programmausführung wird bei der ersten Anweisung nach der Schleife fortgesetzt. |
| RETURN | verläßt die Prozedur. Ist ein Rückgabewert angegeben, dann wird dieser beim Verlassen der Prozedur zurückgegeben. |
| RAISE EXCEPTION | verläßt die Schleife, wenn die Ausnahme *(exception)* nicht in der Schleife abgefangen wird |

Nähere Informationen über Syntax und Verwendung dieser Anweisungen finden Sie in Kapitel 2 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

# Funktionen in SPL

Sie können aus einer Prozedur andere Prozeduren aufrufen und Betriebssystemkommandos ausführen.

## Prozeduren aus einer Prozedur aufrufen

Mit einer CALL-Anweisung oder der Anweisung EXECUTE PROCEDURE können Sie aus einer Prozedur eine andere Prozedur aufrufen. Bild 14-11 zeigt den Aufruf der Prozedur **read_address** mit einer CALL-Anweisung.

```
CREATE PROCEDURE call_test()
    RETURNING CHAR(15), CHAR(15);

    DEFINE fname, lname CHAR(15);
    CALL read_name('Putnum') RETURNING fname, lname;

    IF fname = 'Eileen' THEN RETURN 'Jessica', lname;
    ELSE RETURN fname, lname;
    END IF
END PROCEDURE
```

*Bild 14-11 Prozedur mit der CALL-Anweisung aufrufen*

## Betriebssystemkommandos aus einer Prozedur ausführen

Mit der SYSTEM-Anweisung können Sie aus einer Prozedur einen Systemaufruf absetzen. Bild 14-12 zeigt den Aufruf eines **echo**-Kommandos.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);

DELETE FROM customer
    WHERE customer_num = cnum;

IF username = 'acctclrk' THEN
    SYSTEM 'echo ''Delete from customer by acctclrk'' >> /mis/records/updates' ;
END IF
END PROCEDURE -- delete_customer
```

*Bild 14-12 Systemaufruf aus einer Prozedur mit der SYSTEM-Anweisung absetzen*

## Prozeduren rekursiv aufrufen

Eine Prozedur kann sich selbst aufrufen. Es gibt keine Einschränkungen für den rekursiven Aufruf von Prozeduren.

# Daten mit einer Prozedur austauschen

Beim Erstellen einer Prozedur legen Sie mit einer Argumentliste fest, ob die Prozedur Daten erhalten soll. Geben Sie für jeden Wert, den die Prozedur erwartet, ein Argument und einen Datentyp für das Argument an.

Wenn eine Prozedur beispielsweise einen einzelnen ganzzahligen Wert erhalten soll, dann können Sie den Prozedurkopf folgendermaßen festlegen:

```
CREATE PROCEDURE safe_delete(cnum INT)
```

## Ergebnisse zurückgeben

Eine Prozedur, die einen oder mehrere Werte zurückgibt, muß zwei Programmzeilen enthalten, in denen der Datenaustausch vollzogen wird. Sie müssen den zurückzugebenden Datentyp angeben und die Werte explizit zurückgeben.

### Angeben der Datentypen von Rückgabewerten

Unmittelbar nachdem Sie den Namen und die Eingabeparameter der Prozedur festgelegt haben, müssen Sie in einer RETURNING-Klausel die Datentypen der Werte angeben, die von der Prozedur zurückgegeben werden sollen. Das folgende Beispiel zeigt den Kopf einer Prozedur (Name, Parameter und RETURNING-Klausel), die eine Ganzzahl als Eingabe erwartet und eineGanzzahl und eine Zeichenkette von 10 Byte zurückgibt:

```
CREATE PROCEDURE get_call(cnum INT)
    RETURNING INT, CHAR(10);
```

### Zurückgeben von Werten

Haben Sie in der RETURNING-Klausel die Datentypen der Rückgabewerte festgelegt, dann können Sie mit der RETURN-Anweisung an einer beliebigen Stelle in der Prozedur Werte zurückgeben. Die Werte in der RETURN-Anwei-

sung müssen in Anzahl und Datentyp mit den Angaben in der RETURNING-Klausel übereinstimmen. Das folgende Beispiel zeigt, wie die Prozedur **get_call** Informationen zurückgibt:

```
CREATE PROCEDURE get_call(cnum INT)
    RETURNING INT, CHAR(10);
    DEFINE ncalls INT;
    DEFINE o_name CHAR(10);
    .
    .
    .
    RETURN ncalls, o_name;
    .
    .
    .
END PROCEDURE
```

## Prozeduren mit mehreren Rückgabewerten

Wenn Ihre Prozedur eine SELECT-Anweisung ausführt, die mehr als einen Datensatz aus der Datenbank zurückgeben kann, oder wenn Werte aus einer Schleife zurückgegeben werden, dann müssen Sie in der RETURN-Anweisung die Schlüsselwörter WITH RESUME angeben. Mit einer RETURN...WITH RESUME-Anweisung werden die Rückgabewerte an das aufrufende Programm zurückgegeben. Anschließend wird die Ausführung bei der Anweisung fortgesetzt, die auf die RETURN...WITH RESUME-Anweisung folgt.

Bild 14-13 zeigt ein Beispiel einer solchen "Cursor-Prozedur". Die Prozedur gibt die Werte von einer FOREACH- und einer FOR-Schleife zurück. Man nennt diese Prozedur eine Cursor-Prozedur, da sie eine FOREACH-Schleife enthält und so implizit einen Cursor verwendet:

```
CREATE PROCEDURE read_many  (lastname CHAR(15))
    RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),CHAR(2),
    CHAR(5);

    DEFINE p_lname,p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    DEFINE lcount INT ;
    DEFINE i INT ;

    LET lcount = 0;
    TRACE ON;
    CREATE VIEW myview AS SELECT * FROM customer;
```

```
    TRACE 'Foreach starts';
    FOREACH
    SELECT fname, lname, address1, city, state, zipcode
        INTO p_fname, p_lname, p_add, p_city, p_state, p_zip
        FROM customer
        WHERE lname = lastname
    RETURN p_fname, p_lname, p_add, p_city, p_state, p_zip
        WITH RESUME;
    LET lcount = lcount +1;
    END FOREACH;

    FOR i IN (1 TO 5)
        BEGIN
         RETURN 'a', 'b', 'c', 'd', 'e' WITH RESUME;
        END
    END FOR;
END PROCEDURE
```

*Bild 14-13* ***Prozedur, die Werte aus einer FOREACH-Schleife und einer FOR-Schleife zurückgibt***

Mit dieser Prozedur werden Name und Adresse für jede Person mit dem angegebenen Nachnamen zurückgegeben. Die Prozedur gibt darüberhinaus eine Reihe von Buchstaben zurück. Die aufrufende Prozedur bzw. das aufrufende Programm muß mehrere Rückgabewerte erwarten und einen Cursor oder eine FOREACH-Anweisung zum Verarbeiten mehrerer Rückgabewerte verwenden.

# Fehlerbehandlung in SPL

Sie können alle Fehler- und Ausnahmesituationen, die vom Datenbankserver an Ihre Prozedur gemeldet bzw. von Ihrer Prozedur erzeugt werden, mit der ON EXCEPTION-Anweisung abfangen. Mit der RAISE EXCEPTION-Anweisung können Sie in Ihrer Prozedur eine solche Situation herbeiführen.

## Auf Fehler reagieren

Die ON EXCEPTION-Anweisung gibt Ihnen die Möglichkeit, mit einer Anweisung oder Anweisungsfolge auf Fehler zu reagieren.

Um auf einen Fehler reagieren zu können, müssen Sie die Anweisung(sfolge) in einen Anweisungsblock setzen und dem Block eine ON EXCEPTION-Anweisung voranstellen. Wenn in diesem Anweisungsblock ein Fehler auftritt, dann können Maßnahmen zur Fehlerbehebung ergriffen werden.

Bild 14-14 zeigt das Beispiel einer ON EXCEPTION-Anweisung in einem BEGIN...END-Block.

```
BEGIN
    DEFINE c INT;
    ON EXCEPTION IN
        (
        -206, -- table does not exist
        -217  -- column does not exist
        ) SET err_num

    IF err_num = -206 THEN
            CREATE TABLE t (c INT);
            iNSERT INTO t VALUES (10);
            -- continue after the insert statement
        ELSE
            ALTER TABLE t ADD(d INT);
            LET c = (SELECT d FROM t);
            -- continue after the select statement.
        END IF
    END EXCEPTION WITH RESUME

INSERT INTO t VALUES (10);  -- will fail if t does not exist

LET c = (SELECT d FROM t);  -- will fail if d does not exist
END
```

*Bild 14-14 ON EXCEPTION-Anweisung in einem BEGIN...END-Block*

Wenn ein Fehler auftritt, dann sucht der SPL-Interpreter nach der innersten ON EXCEPTION-Anweisung, die auf den Fehler reagiert. Wenn der Fehler behoben ist und die ON EXCEPTION-Anweisung die Schlüsselwörter WITH RESUME enthält, dann wird die Prozedur automatisch direkt *nach* der Anweisung fortgesetzt, die den Fehler ausgelöst hat. Wenn die ON EXCEPTION-Anweisung nicht die Schlüsselwörter WITH RESUME enthält, dann wird der aktuelle Block ganz verlassen.

# Gültigkeitsbereich einer ON EXCEPTION-Anweisung

Eine ON EXCEPTION-Anweisung ist gültig in dem Anweisungsblock, der auf die ON EXCEPTION-Anweisung folgt, in allen in diesem Anweisungsblock geschachtelten Anweisungsblöcken sowie in allen Anweisungsblöcken, die auf die ON EXCEPTION-Anweisung folgen. Sie ist *nicht* gültig in dem Anweisungsblock, der die ON EXCEPTION-Anweisung enthält. Bild 14-15 zeigt, an welchen Stellen in der Prozedur die Ausnahme gültig ist, wo also beim Auftreten des Fehlers `201` in einem der angegebenen Blöcke die Aktion mit der Bezeichnung `a201` ausgeführt wird:

```
CREATE PROCEDURE scope()
    DEFINE i INT;
    .
    .
    .
    BEGIN   -- begin statement block A
    .
    .
    .
        ON EXCEPTION IN (201)
        -- do action a201
        END EXCEPTION
        BEGIN -- statement block aa
            -- do action, a201 valid here
        END
        BEGIN -- statement block bb
            -- do action, a201 valid here
        END
        WHILE i < 10
            -- do something, a201 is valid here
        END WHILE

    END
    BEGIN   -- begin statement block B
        -- do something
        -- a201 is NOT valid here
    END
END PROCEDURE
```

*Bild 14-15 Gültigkeitsbereich einer ON EXCEPTION-Anweisung in einer Prozedur*

# Benutzerdefinierte Ausnahmen

Mit der RAISE EXCEPTION-Anweisung können Sie selbst einen Fehler erzeugen, wie im folgenden Beispiel gezeigt. In diesem Beispiel verwendet die ON EXCEPTION-Anweisung zwei Variablen mit den Namen **esql** und **eisam**, die die vom Datenbankserver zurückgegebenen Fehlernummern aufnehmen sollen. Wenn ein SQL-Fehler mit der Nummer -206 auftritt, dann werden die in der IF-Klausel angegebenen Aktionen ausgeführt. Wenn irgend ein anderer SQL-Fehler auftritt, dann wird der Fehler an den umschließenden Anweisungsblock weitergereicht.

```
BEGIN
    ON EXCEPTION SET esql, eisam  -- trap all errors
        IF esql = -206 THEN       -- table not found
            -- recover somehow
        ELSE
            RAISE exception esql, eisam ; -- pass the error up
        END IF
    END EXCEPTION
        -- do something
END
```

## Simulieren von SQL-Fehlern

Sie können Fehler erzeugen, um SQL-Fehler zu simulieren, wie im folgenden Beispiel gezeigt. Wenn in dem Beispiel der Benutzer den Namen **pault** hat, dann verhält sich die gespeicherte Prozedur so, als ob dieser Benutzer keine UPDATE-Berechtigung hat, selbst wenn er es in Wirklichkeit besitzt.

```
BEGIN
    IF user = "paul" THEN
        RAISE EXCEPTION -273;  -- deny Paul update privilege
    END IF
END
```

## Verlassen einer Schleife mit RAISE EXCEPTION

Mit der RAISE EXCEPTION-Anweisung verlassen Sie eine tief verschachtelten Block, wie in Bild 14-16 gezeigt. Ist die innerste Bedingung wahr (wenn also aa negativ ist), dann tritt die Ausnahme ein und die Prozedur wird nach der END-Anweisung des Blocks fortgesetzt, in diesem Fall mit der TRACE-Anweisung.

```
BEGIN
    ON EXCEPTION IN (1)
    END EXCEPTION WITH RESUME -- do nothing significant (cont)

    BEGIN
        FOR i IN (1 TO 1000)
            FOREACH select ..INTO aa FROM t
                IF aa < 0 THEN
                    RAISE EXCEPTION 1 ;     -- emergency exit
                END IF
            END FOREACH
        END FOR
    RETURN 1;
    END

    --do something;          -- emergency exit to
                                    -- this statement.
    TRACE "Negative value returned"
    RETURN -10;
END
```

*Bild 14-16* ***Verlassen einer geschachtelten Schleife mit der RAISE EXCEPTION-Anweisung***

Vergessen Sie nicht, daß ein BEGIN...END-Block eine *einzelne* Anweisung ist. Wenn irgendwo in einem Block ein Fehler auftritt, der außerhalb des Blocks abgefangen wird, dann wird der Rest des Blocks übersprungen und die Ausführung bei der nächsten Anweisung fortgesetzt.

Sofern nicht irgendwo geschachtelt in dem Block eine Anweisung eine Reaktion auf diesen Fehler vorsieht, wird die Fehlerbedingung an den Block durchgereicht, der den Aufruf enthält sowie an alle Blöcke, die den Block enthalten. Wenn keine ON EXCEPTION-Anweisung zur Reaktion auf den Fehler vorgesehen ist, dann wird die Ausführung der Prozedur beendet und eine Fehlermeldung an das aufrufende Programm bzw. die aufrufende Prozedur weitergeleitet.

# Zusammenfassung

Mit gespeicherten Prozeduren können Sie auf einfache Weise Datenbankanwendungen programmieren. Zu den Vorteilen gespeicherter Prozeduren zählen vor allem bessere Performance, einfacheres Erstellen von Anwendungen sowie die Möglichkeit, den Datenzugriff einzuschränken bzw. zu überwachen. Die Syntax und Verwendung der einzelnen SPL-Anweisungen finden Sie in Kapitel 2 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

# Trigger erzeugen und verwenden

15

# Kapitelüberblick

Ein SQL-Trigger ist ein Datenbank-Mechanismus, der im Anschluß an eine INSERT-, DELETE-, UPDATE- oder EXECUTE PROCEDURE-Anweisung die Ausführung einer weiteren Anweisung anstößt. Der Trigger ist jedem Benutzer zugänglich, der über die entsprechende Berechtigung verfügt.

In diesem Kapitel werden die einzelnen Komponenten der CREATE TRIGGER-Anweisung erklärt. Es werden auch einige Möglichkeiten vorgestellt, Trigger anzuwenden. Es wird außerdem beschrieben, welche Vorteile es hat, gespeicherte Prozeduren von Triggern auslösen zu lassen.

# Einsatzgebiete für Trigger

Trigger sind in der Datenbank definiert und können von jedem Benutzer mit der entsprechenden Berechtigung aufgerufen werden. Mit Triggern können Sie SQL-Anweisungen schreiben. Diese können Sie dann aus verschiedenen Anwendungsprogrammen heraus aufrufen. Wenn mehrere Programmteile die gleichen Anweisungen ausführen sollen, können durch die Programmierung von Triggern Redundanzen vermieden werden.

Unter anderem können Sie mit Triggern folgende Aktionen ausführen:

- Ein AUDIT-Protokoll zur Datenbank erzeugen, um z. B. alle Änderungen in der Tabelle **orders** der Beispiel-Datenbank zu protokollieren.
- Datenbankverfahren implementieren. Beispielsweise können Sie festlegen, ab welcher Höhe eine Bestellung die Kreditlinie eines Kunden übersteigt und dann eine entsprechende Meldung erscheinen lassen.
- Zusatzinformationen ableiten, die nicht direkt in einer Tabelle oder der Datenbank vorhanden sind. Beispielsweise können Sie bei einer Änderung in der Spalte **quantity** automatisch eine entsprechende Änderung in der Spalte **total_price** vornehmen lassen.
- Referentielle Integrität erzwingen. Wenn Sie beispielsweise die Daten eines Kunden löschen, können Sie mit einem Trigger dafür sorgen, daß

auch die zugehörigen Datensätze (also die mit der gleichen Kundennummer) in der Tabelle **orders** gelöscht werden.

# Trigger erzeugen

Trigger werden mit der Anweisung CREATE TRIGGER erzeugt. Diese Anweisung verknüpft SQL-Anweisungen zu einer Trigger-Aktion. Wenn die Aktion ausgeführt wird, werden damit die SQL-Anweisungen initiiert, die in der Datenbank gespeichert sind. Das folgende Bild verdeutlicht den Zusammenhang zwischen Trigger-Auslöser und Trigger-Aktion.

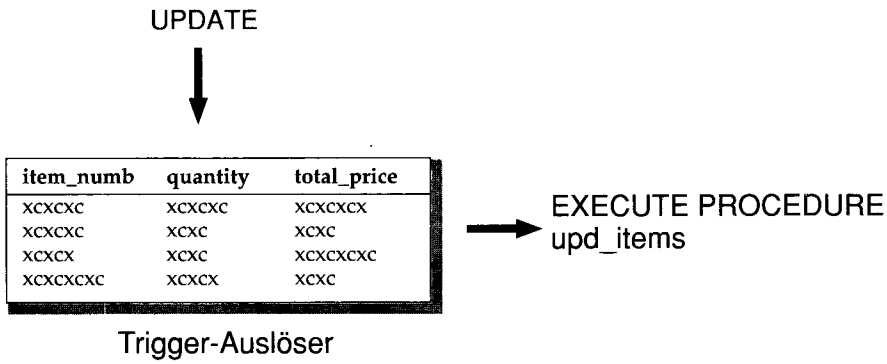



*Bild 15-1* ***Trigger bestehend aus Trigger-Auslöser und Trigger-Aktion***

Die Anweisung CREATE TRIGGER besteht aus einzelnen Klauseln, mit denen Sie einen Trigger in mehreren Schritten definieren. Die folgenden Einzelschritte werden in diesem Kapitel näher erläutert:

- Sie müssen einem Trigger einen Namen geben.
- Sie müssen den Trigger-Auslöser definieren, also die Tabelle und die zugehörige Aktion, die den Trigger auslöst.
- Sie müssen die Aktionen festlegen, die der Trigger auslöst.

Die optionale REFERENCING-Klausel wird später im Abschnitt "Trigger-Aktion FOR EACH ROW" auf Seite 15-9 beschrieben.

Sie können Trigger mit dem Dienstprogramm **DB-Access** oder mit einem der **ESQL**-Produkte **INFORMIX-ESQL/C** oder **INFORMIX-ESQL/COBOL** erzeugen. In diesem Abschnitt wird die CREATE TRIGGER Anweisung so beschrieben, als würden Sie einen Trigger interaktiv mit **DB-Access** erzeugen. Wenn Sie

ein ESQL-Produkt einsetzen, müssen Sie der Anweisung das jeweilige Symbol oder Schlüsselwort voranstellen, mit dem ESQL-Anweisungen gekennzeichnet sind.

## Trigger benennen

Ein Trigger wird durch seinen Namen identifiziert. Der Name wird unmittelbar hinter der Anweisung CREATE TRIGGER eingegeben. Der Name darf aus maximal 18 Zeichen bestehen. Er muß mit einem Buchstaben beginnen und darf aus Buchstaben, den Ziffern 0 bis 9 sowie dem Unterstrich (_) bestehen. Im folgenden Beispiel wird der Teil der CREATE TRIGGER Anweisung gezeigt, in der der Name **upqty** vergeben wird.

```
CREATE TRIGGER upqty              -- assign trigger name
```

## Trigger-Auslöser festlegen

Unter einem *Trigger-Auslöser* versteht man die Anweisung, die den Trigger startet. Die Trigger-Aktion wird ausgelöst, sobald die Anweisung selbst ausgeführt wird. Der Trigger-Auslöser kann eine der Anweisungen INSERT, DELETE oder UPDATE sein. Bei einer UPDATE-Anweisung können Sie eine oder mehrere Spalten der Tabelle festlegen, die den Trigger starten sollen. Wenn Sie keine Spalten angeben, aktiviert jedes Aktualisieren der Tabelle den Trigger. Pro Tabelle kann jeweils nur ein INSERT- und ein DELETE -Trigger definiert werden. Dagegen können mehrere UPDATE-Trigger für die Spalten einer Tabelle definiert werden, solange es sich dabei jeweils um verschiedene Spalten handelt.

Der folgende Ausschnitt aus einer CREATE TRIGGER Anweisung zeigt, wie eine UPDATE-Anweisung für die Spalte **quantity** der Tabelle **items** als Trigger-Auslöser definiert wird.

```
CREATE TRIGGER upqty
UPDATE OF quantity ON items      -- an UPDATE trigger event
```

Beachten Sie, daß in der Klausel der Anweisung die Tabelle angegeben wird, für die der Trigger erzeugt wird. Soll ein Trigger durch eine INSERT- oder eine DELETE-Anweisung ausgelöst werden, muß das Schlüsselwort INSERT bzw. DELETE angegeben werden, wie Sie im folgenden Beispiel sehen können:

```
CREATE TRIGGER ins_qty
INSERT ON items                    -- an INSERT trigger event
```

## Trigger-Aktion festlegen

Als *Trigger-Aktionen* werden diejenigen SQL-Anweisungen bezeichnet, die ausgeführt werden, wenn der Trigger-Auslöser auftritt. Trigger-Aktionen können aus den Anweisungen INSERT, UPDATE, DELETE oder EXECUTE PROCEDURE bestehen. Außerdem müssen Sie festlegen, *wann* die Trigger-Aktionen ausgeführt werden sollen in Bezug zum Trigger-Auslöser. Hier können Sie zwischen den folgenden Möglichkeiten wählen:

- Bevor der Trigger-Auslöser ausgeführt wird.
- Nachdem der Trigger-Auslöser ausgeführt wird.
- Für jeden Datensatz, der vom Trigger-Auslöser betroffen wird.

Ein einzelner Trigger kann Aktionen für alle drei Fälle vorsehen.

Wenn Sie eine Trigger-Aktion festlegen, geben Sie zuerst an, wann sie auftreten soll und definieren dann, aus welchen SQL-Anweisungen sie bestehen soll. Mit den Schlüsselwörtern BEFORE, AFTER und FOR EACH ROW legen Sie fest, wann die Trigger-Aktion ausgeführt werden soll. Anschließend folgt, in Klammern eingeschlossen, die Trigger-Aktion. Die folgende Definition einer Trigger-Aktion gibt an, daß die gespeicherte Prozedur **upd_items_p1** vor der Trigger-Auslöser ausgeführt wird.

```
BEFORE(EXECUTE PROCEDURE upd_items_p1)  -- a BEFORE action
```

## Eine vollständige CREATE TRIGGER Anweisung

Wenn Sie die Klauseln zur Namensdefinition, zur Definition des Trigger-Auslösers sowie zur Definition der Trigger-Aktion zusammennehmen, haben Sie eine vollständige CREATE TRIGGER-Anweisung. Die folgende Anweisung faßt die einzelnen bisherigen Teilbeispiele zusammen. Dieser Trigger startet die gespeicherte Prozedur **upd_items_p1**, sobald die Spalte **quantity** in der Tabelle **items** aktualisiert wird.

```
CREATE TRIGGER upqty
UPDATE OF quantity ON items
BEFORE(EXECUTE PROCEDURE upd_items_p1)
```

Falls ein Objekt in der Trigger-Definition nicht vorhanden ist, wie die gespeicherte Prozedur **upd_items_p1** in unserem Beispiel, meldet der Datenbankserver einen Fehler.

# Trigger-Aktionen verwenden

Um effizient mit Triggern arbeiten zu können, müssen Sie die Beziehung zwischen Trigger-Auslösern und Trigger-Aktionen verstehen. Sie legen die Art dieser Beziehung fest, wenn Sie den Zeitpunkt für die Trigger-Aktion definieren: also BEFORE, AFTER oder FOR EACH ROW.

## Trigger-Aktionen BEFORE und AFTER

Trigger-Aktionen, die vor oder nach dem Trigger-Auslöser stattfinden, laufen jeweils nur einmal ab. Die Trigger-Aktion BEFORE wird *vor* dem Trigger-Auslöser angestoßen, die Trigger-Aktion AFTER entsprechend danach. Die Trigger-Aktionen BEFORE und AFTER werden auch dann ausgeführt, wenn der Trigger-Auslöser selbst keine Datensätze verarbeitet.

Neben anderen Anwendungen können Sie BEFORE und AFTER Trigger-Aktionen verwenden, um die Auswirkungen des Trigger-Auslösers festzulegen. Beispielsweise können Sie mit der gespeicherten Prozedur **upd_items_p1** die Gesamtanzahl aller Bestellungen für alle Artikel berechnen lassen, bevor Sie

die Spalte **quantity** in der Tabelle **items** aktualisieren. Beachten Sie, daß die Prozedur die Summe in einer globalen Variablen mit dem Namen **old_qty** ablegt.

```
CREATE PROCEDURE upd_items_p1()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    LET old_qty = (SELECT SUM(quantity) FROM items);
END PROCEDURE;
```

Nachdem der Trigger-Auslöser UPDATE ausgeführt wurde, können Sie die Summe erneut bilden, um zu sehen, wie sie sich geändert hat. Die im folgenden gezeigte gespeicherte Prozedur **upd_items_p2** berechnet die Summe von **quantity** neu und legt das Ergebnis in der lokalen Variablen **new_qty** ab. Dann vergleicht sie den Inhalt der Variablen **old_qty** mit **new_qty** um festzustellen, ob sich die Gesamtzahl aller Bestellungen um mehr als 50 Prozent vermehrt hat. Falls dies der Fall ist, verwendet die Prozedur die RAISE EXCEPTION-Anweisung, um einen SQL-Fehler zu simulieren:

```
CREATE PROCEDURE upd_items_p2()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    DEFINE new_qty INT;
    LET new_qty = (SELECT SUM(quantity) FROM items);
    IF new_qty > old_qty * 1.50 THEN
        RAISE EXCEPTION -746, 0, 'Not allowed - rule violation;'
    END IF
END PROCEDURE;
```

Der folgende Trigger ruft die gespeicherten Prozeduren **upd_items_p1** und **upd_items_p2** auf, um einen ungewöhnlich hohen Aktualisierung der Spalte **quantity** der Tabelle **items** zu verhindern.

```
CREATE TRIGGER up_items
UPDATE OF quantity ON items
BEFORE(EXECUTE PROCEDURE upd_items_p1())
AFTER(EXECUTE PROCEDURE upd_items_p2());
```

Wenn eine Aktualisierung eine Steigerung der Gesamtzahl aller Bestellungen um mehr als 50 Prozent zur Folge hat, führt die Anweisung RAISE EXCEPTION in **upd_items_p2** dazu, daß der Trigger mit einem Fehler beendet wird. Wenn ein Trigger unter **INFORMIX-OnLine Dynamic Server** scheitert und für die Datenbank die Protokollierung eingeschaltet ist, werden sämtliche Änderungen zurückgesetzt, die sowohl von dem Trigger-Auslöser als auch von den Trigger-Aktionen durchgeführt worden sind. Nähere Infor-

mationen darüber, was geschieht, wenn ein Trigger scheitert, finden Sie in der Beschreibung der Anweisung CREATE TRIGGER in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax*.

# Trigger-Aktion FOR EACH ROW

Eine FOR EACH ROW Trigger-Aktion wird für jeden Datensatz einzeln ausgeführt, der vom Trigger-Auslöser betroffen ist.

Mit dem folgenden Trigger-Auslöser wird eine FOR EACH ROW Trigger-Aktion für jeden Satz in der Tabelle **items** ausgeführt, bei dem die Spalte **manu_code** den Wert „KAR" hat:

```
UPDATE items SET quantity = quantity * 2 WHERE manu_code = 'KAR'
```

Wenn der Trigger-Auslöser keine Datensätze verarbeitet, wird die Trigger-Aktion FOR EACH ROW *nicht* ausgeführt.

## Die REFERENCING Klausel

Wenn Sie eine FOR EACH ROW-Trigger-Aktion erstellen, müssen Sie angeben, ob Sie sich auf einen Spaltenwert vor oder nach der Auswirkung des Trigger-Auslösers beziehen. Um z. B. Änderungen in der Spalte **quantity** der Tabelle **items** zu protokollieren, können Sie die folgende Tabelle erzeugen:

```
CREATE TABLE log_record
    (item_num     SMALLINT,
    ord_num       INTEGER,
    username      CHARACTER(8),
    update_time   DATETIME YEAR TO MINUTE,
    old_qty       SMALLINT,
    new_qty       SMALLINT);
```

Um Werte für die Spalten **old_qty** und **new_qty** zu ermitteln, müssen Sie auf die alten und neuen Werte der Spalte **quantity** in der Tabelle **items** zugreifen können, also auf die Werte vor und nach der Auswirkung des Trigger-Auslösers. Mit der Klausel REFERENCING ist dies möglich.

Mit dieser Klausel können Sie zwei Spaltenpräfixe generieren, die Sie mit einem Spaltennamen kombinieren können. Ein Spaltenpräfix bezieht sich auf den alten Wert der Spalte, der andere auf den neuen. Je nach Bedarf können Sie einen oder zwei Spaltenpräfixe erzeugen. Mit den Schlüsselwörtern OLD

und NEW zeigen Sie an, welchen Sie erzeugen wollen. Die folgende REFERENCING-Klausel erzeugt die Spaltenpräfixe **pre_upd** und **post_upd**, um sich auf alte und neue Werte einer Spalte zu beziehen.

```
REFERENCING OLD AS pre_upd NEW AS post_upd
```

Die folgende Trigger-Anweisung erzeugt einen Datensatz in der Tabelle **log_record**, sobald in irgendeinem Datensatz der Tabelle **items** die Spalte **quantity** verändert wird. Die INSERT Anweisung führt die alten Werte der Spalten **item_num** und **order_num** sowie die alten *und* neuen Werte der Spalte **quantity** ein.

```
FOR EACH ROW(INSERT INTO log_record
    VALUES (pre_upd.item_num, pre_upd.order_num, USER, CURRENT,
            pre_upd.quantity, post_upd.quantity));
```

Die in der Klausel REFERENCING definierten Spaltenpräfixe beziehen sich auf alle Datensätze, die vom Trigger-Auslöser verändert werden.

In SQL-Anweisungen innerhalb von Trigger-Aktionen des Typs FOR EACH ROW brauchen Sie einer Spalte nur dann keinen Spaltenpräfix voranzustellen, wenn die Anweisung unabhängig von der Trigger-Aktion gültig ist. Nähere Informationen zu diesem Thema finden Sie in der Beschreibung der CREATE TRIGGER-Anweisung in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

## Die WHEN-Klausel

Sie können einer Trigger-Aktion eine WHEN-Klausel voranstellen, um die Aktion abhängig zu machen vom Ergebnis einer Prüfung. Diese Klausel besteht aus dem Schlüsselwort WHEN, gefolgt von einer Bedingung, die in Klammern angegeben wird. Innerhalb der CREATE TRIGGER-Anweisung folgt die WHEN-Klausel den Schlüsselwörtern BEFORE, AFTER oder FOR EACH ROW und steht vor der Liste der Trigger-Anweisungen.

Falls die WHEN-Bedingung erfüllt ist, werden die Trigger-Aktionen der Reihe nach abgearbeitet. Falls die WHEN- Bedingung *falsch* oder *unbekannt* ist, werden die Trigger-Aktionen in der Liste nicht ausgeführt. Falls es sich um eine Trigger-Aktion des Typs FOR EACH ROW handelt, wird die Bedingung für jeden Datensatz geprüft.

Im folgenden Beispiel wird die Trigger-Aktion nur ausgeführt, wenn die Bedingung der WHEN-Klausel erfüllt ist, wenn also der Stückpreis nach dem Aktualisieren mehr als das Doppelte des ursprünglichen Preises beträgt:

```
CREATE TRIGGER up_price
UPDATE OF unit_price ON stock
REFERENCING OLD AS pre NEW AS post
FOR EACH ROW WHEN(post.unit_price > pre_unit_price * 2)
    (INSERT INTO warn_tab VALUES(pre.stock_num, pre.order_num,
        pre.unit_price, post.unit_price, CURRENT))
```

Nähere Informationen über die WHEN-Klausel finden Sie in der Beschreibung der CREATE TRIGGER-Anweisung in Kapitel 1 des Handbuchs *SQL-Sprachbeschreibung, Syntax.*

# Gespeicherte Prozeduren als Trigger-Aktionen

Die wohl mächtigste Funktionalität von Triggern ist die Möglichkeit, gespeicherte Prozeduren als Trigger-Aktionen aufzurufen. Mit der Anweisung EXECUTE PROCEDURE können Sie sowohl Daten aus der Trigger-Tabelle an eine gespeicherte Prozedur übergeben als auch die Trigger-Tabelle mit den Daten aktualisieren, die von einer gespeicherten Prozedur zurückgeliefert werden. Mit der Sprache für gespeicherte Prozeduren SPL („Stored Procedure Language") können Sie Variablen definieren, ihnen Werte zuweisen, Vergleiche durchführen und komplexe Aufgaben innerhalb einer Trigger-Aktion ausführen.

## Daten an gespeicherte Prozeduren übergeben

Sie übergeben Daten an gespeicherte Prozeduren in der Argumentliste der Anweisung EXECUTE PROCEDURE. In dem unten gezeigten Trigger übergibt die Anweisung EXECUTE STATEMENT Werte aus den Spalten **quantity** und **total_price** der Tabelle **items** an die gespeicherte Prozedur **calc_totpr**.

```
CREATE TRIGGER upd_totpr
UPDATE OF quantity ON items
REFERENCING OLD AS pre_upd NEW AS post_upd
FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
    post_upd.quantity, pre_upd.total_price) INTO total_price)
```

Die überg

ebenen Daten können Sie in den Berechnungen der Prozedur verwenden.

### Arbeiten mit der Stored Procedure Language (SPL)

Die Anweisung EXECUTE PROCEDURE im oben gezeigten Trigger ruft die unten gezeigte Prozedur auf. Die Prozedur verwendet die Sprache SPL, um die nötigen Änderungen zu berechnen, die in der Spalte **total_price** vorgenommen werden müssen, nachdem in der Tabelle **items** die Spalte **quantity** aktualisiert wurde. Die Prozedur erhält dabei sowohl den alten als auch den neuen Wert von **quantity** und den alten Wert von **total_price**. Sie dividiert den alten Gesamtpreis durch den alten Wert von **quantity**, um den Stückpreis zu berechnen. Anschließend wird der Stückpreis mit dem neuen Wert von **quantity** multipliziert und so der neue Gesamtpreis ermittelt.

```
CREATE PROCEDURE calc_totpr(old_qty SMALLINT, new_qty SMALLINT,
    total MONEY(8)) RETURNING MONEY(8);
    DEFINE u_price LIKE items.total_price;
    DEFINE n_total LIKE items.total_price;
    LET u_price = total / old_qty;
    LET n_total = new_qty * u_price;
    RETURN n_total;
END PROCEDURE;
```

In diesem Beispiel ermöglicht SPL es dem Trigger, Daten zu ermitteln, die nicht *direkt* in der Trigger-Tabelle enthalten sind.

### Nicht-getriggerte Spalten mit Daten aus gespeicherten Prozeduren aktualisieren

Innerhalb einer Trigger-Aktion können Sie mit der Klausel INTO der EXECUTE PROCEDURE-Anweisung Aktualisierungen von nicht-getriggerten Spalten der Trigger-Tabelle durchführen. So enthält die EXECUTE PROCEDURE-Anweisung, die die gespeicherte Prozedur **calc_totpr** aufruft, eine INTO-Klausel, die sich auf die Spalte **total_price** bezieht:

```
FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
    post_upd.quantity, pre_upd.total_price) INTO total_price);
```

Der Wert, der in der Spalte **total_price** aktualisiert wird, wird mit einer RETURN-Anweisung am Ende der gespeicherten Prozedur zurückgegeben. Die Spalte **total_price** wird für jeden Datensatz aktualisiert, der vom Trigger-Auslöser betroffen ist.

# Ablaufverfolgung von Trigger-Aktionen

Falls eine Trigger-Aktion nicht wie erwartet abläuft, können Sie sie in eine gespeicherte Prozedur einfügen und mit der SPL-Anweisung TRACE ihren Ablauf überwachen. Bevor Sie die Ablaufverfolgung starten, müssen Sie mit der Anweisung SET DEBUG FILE TO das Ergebnis in eine Datei umlenken. Im folgenden Beispiel wurde die gespeicherte Prozedur **items_pct** um TRACE -Anweisungen erweitert. Die Anweisung SET DEBUG FILE TO lenkt das Ergebnis in die Datei **/usr/mydir/trig.trace**. Die Anweisung TRACE ON schaltet die Ablaufverfolgung von Anweisungen und Variablen an:

```
CREATE PROCEDURE items_pct(mac CHAR(3))
DEFINE tp MONEY;
DEFINE mc_tot MONEY;
DEFINE pct DECIMAL;
SET DEBUG FILE TO '/usr/mydir/trig.trace';
TRACE 'begin trace';
TRACE ON
LET tp = (SELECT SUM(total_price) FROM items);
LET mc_tot = (SELECT SUM(total_price) FROM items
    WHERE manu_code = mac);
LET pct = mc_tot / tp;
IF pct > .10 THEN
    RAISE EXCEPTION -745;
END IF
TRACE OFF;
END PROCEDURE;

CREATE TRIGGER items_ins
INSERT ON items
REFERENCING NEW AS post_ins
FOR EACH ROW(EXECUTE PROCEDURE items_pct (post_ins.manu_code));
```

*Bild 15-2* ***Ablaufverfolgung einer Trigger-Aktion innerhalb einer gespeicherten Prozedur***

Bild 15-3 zeigt beispielhaft das Ergebnis der Ablaufverfolgung der Prozedur **items_pct**, das in die Datei **/usr/mydir/trig.trace** umgelenkt wurde. Der Ausdruck zeigt den Wert von Prozedurvariablen, Prozedurargumenten, Rückgabewerten und Fehlercodes.

```
trace expression :begin trace
trace on
expression:
  (select (sum total_price)
    from items)
evaluates to $18280.77 ;
let  tp = $18280.77
expression:
  (select (sum total_price)
    from items
    where (= manu_code, mac))
evaluates to $3008.00 ;
let  mc_tot = $3008.00
expression:(/ mc_tot, tp)
evaluates to 0.16
let  pct = 0.16
expression:(> pct, 0.1)
evaluates to 1
expression:(- 745)
evaluates to -745
raise exception :-745, 0, ''
exception : looking for handler
SQL error = -745 ISAM error = 0  error string =  = ''
exception : no appropriate handler
```

*Bild 15-3* *Ergebnis der Ablaufverfolgung*

Nähere Informationen darüber, wie Sie mit der TRACE-Anweisung logische Fehler in Ihren gespeicherten Prozeduren finden können, finden Sie in Kapitel 14 "Gespeicherte Prozeduren".

# Fehlermeldungen erzeugen

Wenn ein Trigger aufgrund einer SQL-Anweisung scheitert, gibt der Datenbankserver die SQL-Fehlernummer zurück, die die Ursache des jeweiligen Fehlers anzeigt.

Wenn die Trigger-Aktion eine gespeicherte Prozedur ist, können Sie für andersartige Fehlerursachen *eigene* Fehlermeldungen erzeugen, indem Sie eine von zwei reservierten Fehlernummern verwenden. Die erste, -745, enthält einen allgemeinen unveränderbaren Meldungstext. Bei der zweiten Nummer, -746, können Sie einen eigenen Meldungstext eingeben, der bis zu 71 Zeichen lang sein darf.

# Eine vorgegebene Fehlermeldung vergeben

Sie können die Fehlernummer -745 bei jedem Trigger-Fehler ausgeben lassen, der kein SQL-Fehler ist. Dann erscheint folgende Fehlermeldung:

```
-745 Trigger execution has failed
```

Mit der SPL-Anweisung RAISE EXCEPTION können Sie diese Meldung erzwingen. Im folgenden Beispiel wird die Fehlermeldung -745 erzeugt, wenn der Wert von **new_qty** das Eineinhalbfache von **old_qty** übersteigt:

```
CREATE PROCEDURE upd_items_p2()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    DEFINE new_qty INT;
    LET new_qty = (SELECT SUM(quantity) FROM items);
    IF new_qty > old_qty * 1.50 THEN
        RAISE EXCEPTION -745;
    END IF
END PROCEDURE
```

Falls Sie mit **DB-Access** arbeiten, erscheint der Text dieser Fehlermeldung unten auf dem Bildschirm, wie Sie in Bild 15-4 sehen:

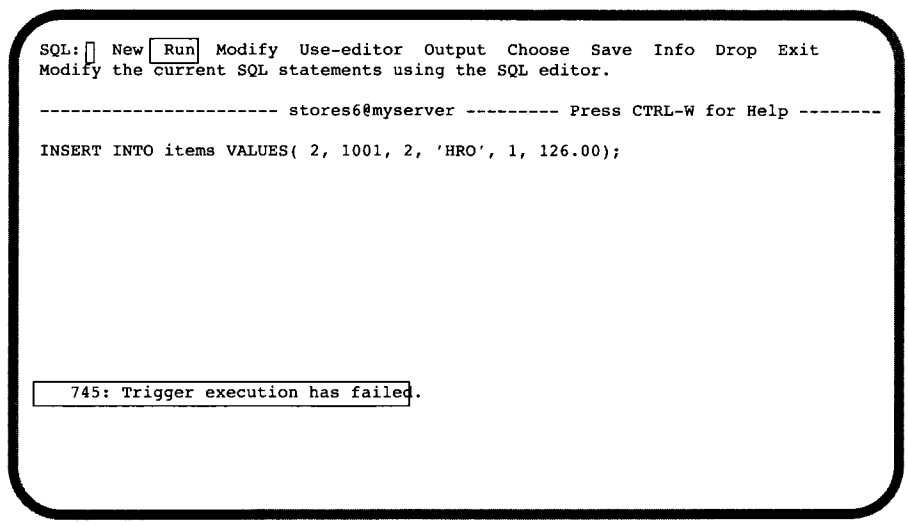



*Bild 15-4* ***Fehlermeldung mit vorgegebenem Meldungstext***

Wird die fehlerhafte Prozedur mit einer SQL-Anweisung in einem **SQL-API** ausgelöst, setzt der Datenbankserver die SQL-Fehlerstatusvariable auf -745 und liefert sie an Ihr Programm zurück. Wie Sie den Text der SQL-Fehlermel-

dung anzeigen lassen können, können Sie der Dokumentation zu dem von Ihnen eingesetzten Informix-Werkzeug zur Anwendungsentwicklung entnehmen.

## Eine variable Fehlermeldung vergeben

Mit der Fehlernummer -746 können Sie den Text der Fehlernummer selbst festlegen. Wie schon das vorherige Beispiel erzeugt auch das folgende einen Fehler, wenn der Wert von **new_qty** das Eineinhalbfache von **old_qty** übersteigt. In diesem Fall ist die Fehlernummer aber -746, und als drittes Argument in der Anweisung RAISE EXCEPTION erscheint der folgende Meldungstext:

```
- Too many items for Mfr. -
```

Nähere Informationen über Syntax und Verwendung der Anweisung RAISE EXCEPTION finden Sie in Kapitel 14 dieses Handbuchs.

```
CREATE PROCEDURE upd_items_p2()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    DEFINE new_qty INT;
    LET new_qty = (SELECT SUM(quantity) FROM items);
    IF new_qty > old_qty * 1.50 THEN
        RAISE EXCEPTION -746, 0, 'Too many items for Mfr.';
    END IF
END PROCEDURE
```

Falls Sie mit **DB-Access** arbeiten, wird das Ergebnis der gespeicherten Prozedur ausgegeben wie in Bild 15-5 gezeigt:

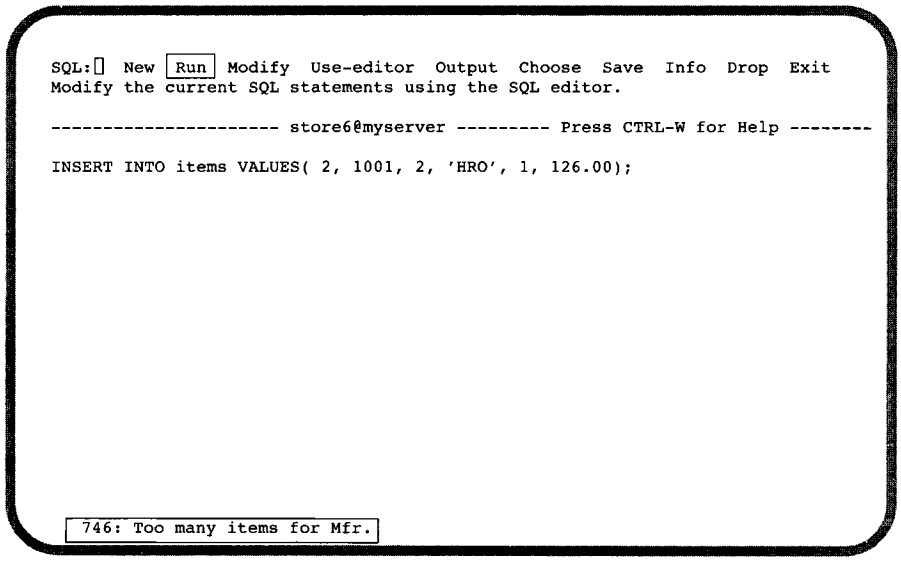



*Bild 15-5* ***Fehlermeldung -746 mit individuellem Meldungstext***

Falls Sie den Trigger durch eine SQL-Anweisung in einem **ESQL**-Programm auslösen, setzt der Datenbankserver das Feld **sqlcode** der Struktur SQLCA auf den Wert -746 und liefert den Meldungstext an das Feld **sqlerrm** der Struktur SQLCA. Nähere Informationen über die SQLCA finden Sie in "SQL in Programmen" ab Seite 5-3 oder in der Dokumentation zu dem von Ihnen eingesetzten **SQL-API**.

# Zusammenfassung

Dieses Kapitel erläuterte, wie Sie Trigger erzeugen und verwenden können. Dabei wurden die folgende Informationen vermittelt:

- Die Funktion der einzelnen Elemente der Anweisung CREATE TRIGGER wurde erklärt.
- Es wurde beschrieben, wie Trigger-Aktionen der Typen BEFORE und AFTER erzeugt werden und wie man sie verwendet, um die Auswirkung des Trigger-Auslösers festzulegen.
- Es wurde beschrieben, wie eine Trigger-Aktionen des Typs FOR EACH ROW erzeugt wird und wie man die Klausel REFERENCING einsetzt, um Spaltenwerte sowohl vor als auch nach dem Ausführen der Trigger-Aktion anzusprechen.
- Die Vorzüge gespeicherter Prozeduren als Trigger-Aktionen wurden geschildert.
- Die Ablaufverfolgung bei fehlerhaften Trigger-Aktionen wurde erklärt.
- Es wurde erklärt, wie man zwei Arten von Fehlermeldungen in Trigger-Aktionen implementiert.

# Stichwörter

## B

## C

## D

## E

## F

## G

## H



## I

## J

## K



## L

## M



## N

## O

## P

## R

## S

## Z

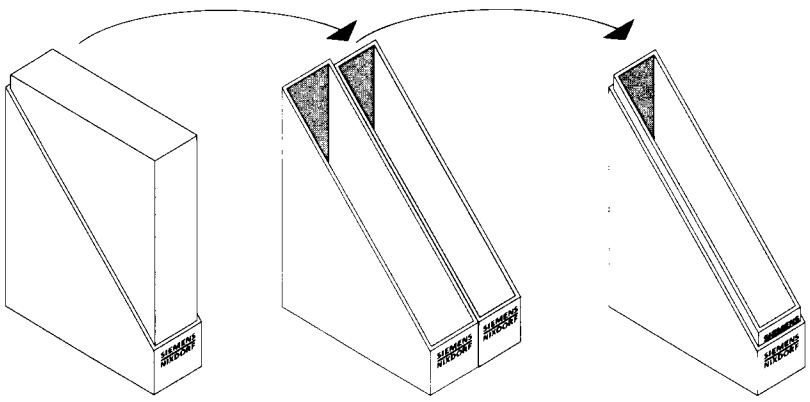


## Symbole

# Sammelboxen / Organizer cases

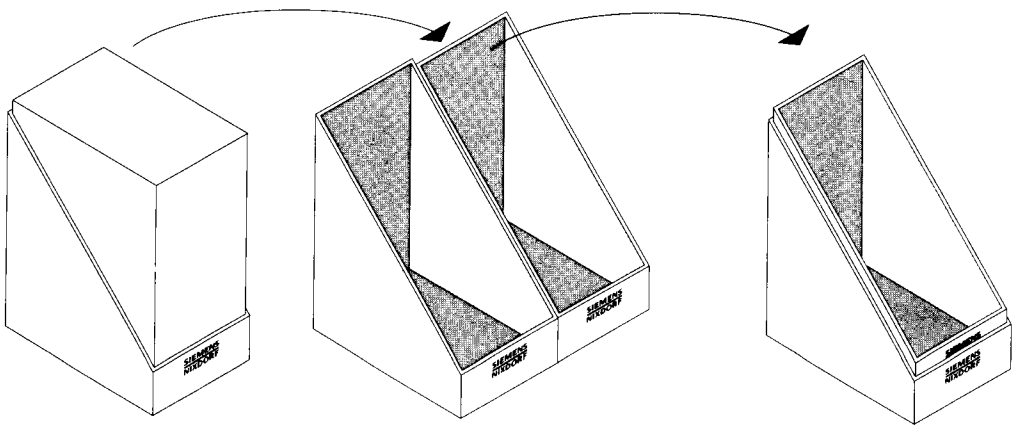
Für Handbücher des vorliegenden Formates bieten wir zweiteilige Sammelboxen in zwei unterschiedlichen Größen an. Der Bestellvorgang entspricht dem für Handbücher.

Two-part organizer cases are available in two sizes for storing manuals in the present format. The ordering procedure is the same as for manuals.



Breite: ca. 5 cm
Bestellnummer: U3775-J-Z18-1

Width: approx. 5 cm
Order No.: U3775-J-Z18-1



Breite: ca. 10 cm
Bestellnummer: U3776-J-Z18-1

Width: approx. 10 cm
Order No.: U3776-J-Z18-1

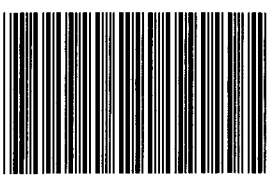
945673

Herausgegeben von / Published by
Siemens Nixdorf Informationssysteme AG
D-33094 Paderborn
D-81730 München

Bestell-Nr./Order No. ***U9634-J-Z265-2***
Printed in the Federal Republic of Germany
4850 AG 7942. (8060) F

Das Papier dieser Broschüre erfüllt unsere Forderungen nach einem umweltfreundlichen Papier. Das Rohpapier wird aus chlorfrei gebleichtem Zellstoff hergestellt.

This brochure is printed on environmentally friendly paper, cellulose treated with chlorine-free bleach.



9Y505955