Einführung in die Datenbanksprache

SQL

Eine Schulungsunterlage von Volker König

Stand: 22.04.04

[volker könig - freier journalist]

Postfach 2141 47913 Tönisvorst

fon 02156 491007_mobil 0163 7031025_fax 02156 491008

www.volkerkoenig.de_vk@volkerkoenig.de

© 1994-2004 by Volker König
Eichenstraße 20
47918 Tönisvorst
http://www.volkerkoenig.de
vk@volkerkoenig.de

Dieses Dokument wurde mit OpenOffice erstellt und gepflegt. OpenOffice ist eine kostenlose Office-Suite nach dem OpenSource-Standard. Sie können die aktuellen Versionen unter www.openoffice.org erhalten.

Inhaltsverzeichnis

1 EINLEITUNG 7

1.1 Diese Schulung 7
1.2 Dieses Dokument 7
1.3 Voraussetzungen 7
1.4 Lehrgangsziel 8
1.5 MySQL and more 8

2 DATENBANKEN 9

2.1 Warum Datenbanken? 9
2.1.1 Datenmodelle 9
2.1.1.1 Unstrukturierte Datenbanken (Flat Files) 10
2.1.1.2 Sequentielles File 10
2.1.1.3 ISAM-Dateien 10
2.1.2 Strukturierte Datenbanken 10
2.1.2.1 Hierarchische Datenbanken 11
2.1.2.2 Netzwerkmodelle 13
2.1.2.3 Relationale Datenbanken 13
2.1.2.4 Objektorientierte Datenbanken 13

3 RELATIONALE DATENBANKSYSTEME 15

3.1 Entstehung 15
3.2 IBM und System/R 15
3.3 Probleme bestehender Systeme 15
3.4 Symbiose 16
3.5 Beschreiben statt Navigieren 16
3.6 Implementationen 17
3.7 Normierungen 17
3.8 Erweiterungen 18
3.9 Grundlagen 18
3.10 Verteilte Daten 19

4 'ZWÖLF REGELN' 21

5 DATENMODELLIERUNG 24

5.1 Entitäten 25
5.2 Relationships 27
5.3 Relationen 30
5.4 Normalisieren 32
5.4.1 Erste Normalform 32
5.4.2 Zweite Normalform 32
5.5 Dritte Normalform 33
5.6 Tabellen definieren 34

6 DB2, SQL UND SPUFI 35

6.1 SQL 35
6.2 DB2/MVS: SPUFI 35
6.3 DB2/UDB Server: db2-Commandline 35
6.4 ORACLE: SQL/Plus und svrmgrl 36
6.5 MySql: der mysql-Befehl und mySqlManager 36
6.6 ODBC & Co 36
6.7 Tabellen anlegen 37

7 EINFACHE ABFRAGEN 38

7.1 Selektieren 38
7.2 Qualifiziertes SELECT 39

8FUNKTIONEN 43
8.1Skalare Funktionen 43
8.1.1CHR 43
8.1.2CHAR() 44
8.1.3CONCAT 44
8.1.4DATE() 45
8.1.5DATEPART() 45
8.1.6DAY 46
8.1.7DAYOFMONTH 46
8.1.8DAYOFYEAR 46
8.1.9DAYS 47
8.1.10DECIMAL 47
8.1.11DIGITS 48
8.1.12FLOAT 48
8.1.13HEX 48
8.1.14HOUR 49
8.1.15INTEGER 49
8.1.16LENGTH 49
8.1.17MICROSECOND 49
8.1.18MINUTE 50
8.1.19MONTH 50
8.1.20RAND 50
8.1.21SECOND 50
8.1.22SUBSTR 51
8.1.23TIME 51
8.1.24TIMESTAMP 51
8.1.25TO_CHAR (ORACLE) 51
8.1.26VALUE 52
8.1.27VARGRAPHIC 52
8.1.28YEAR 53
8.2Spaltenfunktionen 53
8.2.1SUM 53
8.2.2MIN 53
8.2.3MAX 54
8.2.4AVG 54
8.2.5COUNT 54
8.2.6COUNT DISTINCT 54
8.3Abfragen mit Spaltenfunktionen 55
9EINSCHRÄNKEN DER ERGEBNISMENGE 57
9.1Vergleiche mit Funktionen und 'virtuellen' Spalten 59
9.1.1Skalare Funktionen 59
9.1.1.1Kleine Scherzartikelkunde für DB2-Anwender 61
10SORTIEREN 63
10.1Sortierfolge nach 'Original-Spalten' 63
11GRUPPIEREN 66
11.1DISTINCT 67
11.2Auswählen mit HAVING 68
12VERBUNDENE ABFRAGEN 71
12.1Tabellen mit Join verknüpfen 71
12.2Mengen zusammenführen 71
12.2.1Performanceüberlegungen 74
12.3UNION 76
12.4UNION ALL 80

12.5 SUBSELECT 81

13 VERÄNDERN VON DATEN 84

13.1 INSERT 84
13.1.1 Unqualifiziertes INSERT 84
13.1.2 Qualifiziertes INSERT 85
13.1.3 „BULK INSERT“ 86
13.2 DELETE 86
13.3 UPDATE 87

14 SQL IN DER PROGRAMMIERUNG 89

14.1 PRINZIP DES EMBEDDED SQL 89
14.1.1 Programmumwandlung 89
14.1.2 Binden des Planes 90
14.1.3 Form der Einbettung 91
14.1.4 DELETE 92
14.1.4.1 EINBETTUNG DER STATEMENTS 93
14.2 ERWEITERUNGEN DER SYNTAX 94
14.2.1 INTO 95
14.2.2 CURSOR-Verarbeitung 95
14.2.2.1 DECLARE CURSOR 95
14.2.2.2 OPEN und CLOSE 96
14.2.2.3 FETCH 97
14.2.2.4 DECLARE CURSOR 97
14.2.3 FOR UPDATE OF 98
14.2.4 FOR FETCH ONLY 99
14.2.5 OPTIMIZE FOR 99
14.2.6 WITH HOLD 100
14.3 TRANSAKTIONSGRENZEN 100
14.3.1 COMMIT 101
14.3.2 ROLLBACK 102
14.4 DYNAMISCHES SQL 102
14.4.1 DECLARE CURSOR 103
14.4.2 EXECUTE 103
14.4.3 EXECUTE IMMEDIATE 104

15 PERFORMANCE 105

15.1 DATENMODELLIERUNG 105
15.1.1 Datentypen 105
15.1.1.1 Numerische Datentypen 105
15.1.1.2 Character-Spalten fester Länge 105
15.1.1.3 VARCHAR-Felder 106
15.1.1.4 Schlüsselspalten (Primary Key) 107
15.2 NICHT-RELATIONALE STRUKTUREN 107
15.3 INDEX-DEFINITIONEN 108
15.3.1 Optimieren von Suchbedingungen durch Indizes 110
15.3.2 Indizes und Sortierfolgen 110
15.4 ÜBERLISTEN DES OPTIMIZERS 111
15.5 PROGRAMMDESIGN 112
15.5.1 Statisches, eingebettetes SQL 112
15.5.2 Dynamisches SQL 113
15.6 CODIERUNG DER STATEMENTS 114
15.6.1 Bewertung von Statements mittels EXPLAIN 115
15.6.1.1 Explain beim Binden 116

1 ANHANG A: TABELLENDEFINITIONEN 117

1 Einleitung

1.1 Diese Schulung

Datenbanken waren bis vor einigen Jahren Angelegenheit von Spezialisten. Eine Datenbank war eine Black Box, ein Kasten, in dem Programme Daten speichern und aus dem sie diese wieder abrufen. Gleich, ob am Arbeitsplatz oder im privaten Bereich – beispielsweise bei der Verwaltung von Vereinsdaten – Datenbanken waren der Safe, in den wir unsere Daten gesperrt haben, und die Anwendungsprogramme waren die Schlüssel dazu.

Aber die Programme konnten nicht auf alle Fragen Antworten geben – wie viele Vereinsmitglieder vom Jahrgang 1971 haben wir eigentlich? Schwierig, aber die teure Vereinsverwaltung hat vielleicht eine Statistikfunktion, die das beantworten kann. Notfalls kann man bei 200 Mitgliedern auch die Liste durchgehen und manuell zählen.

Schwieriger wird es bei dieser Frage: Gibt es Zusammenhänge zwischen der Zahlungsmoral und der Dauer der Mitgliedschaft? Solche Fragen können wichtig sein, wenn Sie wissen wollen, ob die Mitgliedsbeiträge besser per Bankeinzug oder weiterhin bar entrichtet werden sollen.

Auch EDV-Fachleute ohne Datenbankausbildung, die mit der Abfragesprache SQL in Berührung kommen, können von dieser Schulung profitieren. Da es sich bei relationalen Datenbanken um ein Konzept handelt, das Grundlage der meisten zeitgenössischen Datenbanken ist, wurde auch das Konzept der Relationalen Datenbanken und das Design einer Datenbank Bestandteil dieses Kurses.

1.2 Dieses Dokument

Die vorliegende Schulungsunterlage wurde als kursbegleitendes Dokument entwickelt und ist dem Aufbau der Schulung entsprechend gegliedert. Jedoch wurde darauf Wert gelegt, dass notfalls auch ein Selbststudium auf der Basis der Schulungsunterlage möglich ist.

1.3 Voraussetzungen

Voraussetzungen für die Teilnahme an dieser Schulung sind Grundkenntnisse in der Handhabung von Windows. Weiterhin sind grundsätzliche Kenntnisse der Programmierung von

Nutzen. Es wird jedoch keinerlei Kenntnis einer bestimmten Programmiersprache vorausgesetzt.

1.4 Lehrgangsziel

Am Ende der Schulung sind Sie in der Lage, relationale Datenstrukturen zu verstehen und selbständig Datenmodelle zu entwickeln. Sie werden diese mit geeigneten Hilfsmitteln und der Abfragesprache SQL in komplexen Abfragen auswerten können.

1.5 MySQL and more

Diese Schulung wurde ursprünglich für DB2 von IBM konzipiert. Dieser SQL-basierte Datenbankmanager ist in den letzten Jahren auf alle von IBM vertriebenen Hardwareplattformen portiert worden und eignet sich durch seine einfache Administration gut für die Entwicklung von Client-/Serveranwendungen im kommerziellen Bereich.

Als kostenlose SQL-Datenbank steht unter http://www.mysql.com ein freier (also für Privatpersonen kostenlos nutzbarer) SQL-Server zur Verfügung. Übrigens ist Microsoft keineswegs „Erfinder" des SQL-Servers, sondern hat lediglich seinem Produkt die Bezeichnung „SQL-Server" gegeben. Das ist ähnlich frech, wenn ein Opel einen neuen Fahrzeugtyp einfach „Auto" nennen würde.

ORACLE wiederum ist im UNIX- und Serverbereich seit Jahren Marktführer (wenngleich ein ziemlich harter Preiskampf mit IBM ausgebrochen ist und DB2 Universal Database ziemlich viel Terrain gewonnen hat)

Aber auch Desktop-Datenbanken wie Paradox oder MX-ACCESS sind in der Lage, Daten per SQL abzufragen. SQL stellt insofern eine Art „Datenbankesperanto" dar, wenngleich die Dialekte mitunter abweichen.

Bei der Arbeit mit einer spezifischen Datenbank ist es daher empfehlenswert, ein Handbuch zur SQL-Syntax der Datenbank griffbereit zu haben. Insbesondere die unterstützten Datentypen und die SQL-Funktionen variieren von System zu System.

Ich habe versucht, bei Abweichungen zwischen den verschiedenen Servern in einer Synopse die Unterschiede aufzuführen.

2 Datenbanken

2.1 Warum Datenbanken?

Die letzten Jahrzehnte haben unsere Gesellschaft zu einer ausgeprägten Informationsgesellschaft gemacht. Max Vetter formulierte die Aufgabe der Informatik so[1]:

> Das Jahrhundertproblem der Informatik besteht in:
>
> 1. der Bewältigung des Datenchaos, das infolge unkontrolliert gewachsener Datenbestände fast überall entstanden ist.
> 2. Der Schaffung einer Datenbasis, die für die effiziente Nutzung zukunftsträchtiger Möglichkeiten der Informatik - gemeint sind benützerfreundliche, auch Nichtinformatikern zumutbare Anwendungsgeneratoren und höhere Datenbanksprachen - unerlässlich ist.

Diese Aufgabe gilt es durch IT-Verfahren zu bewältigen. Derartige Verfahren müssen zunächst in der Lage sein, die relevanten Daten in standardisierter Form auf Speichermedien abzulegen, wiederzufinden und zu interpretieren. Es müssen standardisierte Möglichkeiten geschaffen werden, die Daten auszuwerten. Auswerten bedeutet, auf Grund der gespeicherten Daten Rückschlüsse auf die Vorgänge und Sachverhalte in der 'realen' Welt ziehen zu können, welche sie beschreiben.

2.1.1 Datenmodelle

Wenn man durch IT-Verfahren Rückschlüsse auf die 'reale' Welt ermöglichen will, dann ist dafür Sorge zu tragen, dass diese möglichst naturgetreu im Verfahren abgebildet wird. Da die Realität nur durch ihre Daten abgebildet werden kann, müssen eben diese ein möglichst zutreffendes Modell der Realität ergeben.

Im Laufe der Zeit haben sich mehrere Ansätze zur Datenmodellierung entwickelt, von denen die wichtigsten hier kurz angeschnitten werden sollen.

[1] Vetter, Max; Aufbau betrieblicher Informationssysteme; Teubner Verlag; Stuttgart 1990.

2.1.1.1 Unstrukturierte Datenbanken (Flat Files)

Bei diesem flachen oder auch 'zweidimensionalen' Modellen handelt es sich schlicht um sequentielle Dateien, die nach einem bestimmten Muster aufgebaute Datensätze enthalten. Sie haben nur zwei Dimensionen: eine 'Breite', nämlich die Größe des einzelnen Datensatzes, und eine 'Länge', nämlich die Anzahl der Datensätze in der Datei.

2.1.1.2 Sequentielles File

Ein Programm, das in einer derartigen Datei nach einem bestimmten Datensatz sucht, muss sie Satz für Satz lesen und mit dem Suchkriterium vergleichen, bis Übereinstimmung gefunden wurde. Statistisch gesehen muss für jede Suche durchschnittlich die Hälfte aller Datensätze gelesen und ausgewertet werden.

Beim Anfügen neuer Sätze kann man die Sortierfolge nicht einhalten, da neue Sätze nur am Ende der Datei angefügt werden können. Benötigt man sortierte Daten, so ist die Datei vorher durch ein spezielles Sortierprogramm zu bearbeiten.

2.1.1.3 ISAM-Dateien

ISAM ist die Abkürzung für 'Index-Sequential-Access-Method'. Bei dieser Zugriffsmethode setzt sich die Datenbank aus zwei sequentiellen Dateien zusammen, einem Datenteil und einem Index. Der Datenteil entspricht völlig dem gerade beschriebenen sequentiellen File.

Auch diese Speicherungsmethode ist unstrukturiert. Lediglich bei der Suche nach definierten Schlüsselfeldern der Datensätze, die in einem oder mehreren Indizes geführt werden, kann die schnellere Suchmethode über die wesentlich kleinere Index-Datei benutzt werden. Diese kann, da sie sehr klein ist, einfacher in der korrekten Sortierfolge gehalten werden, als der Datenteil. Meistens kann man mehrere Indexe pro Datei definieren, was mehrere Suchkriterien für schnelle indizierte Suche bietet.

Allerdings muss der Index bei jedem Einfügen, Ändern und Löschen von Sätzen gepflegt werden, was Zeit kostet.

2.1.2 Strukturierte Datenbanken

Flat Files haben spätestens dann erhebliche Mängel, wenn man tatsächliche, natürlich angefallene Daten darin abbilden will. Nehmen wir als Beispiel eine Datenbank über Ihre CDs: Pro Datensatz in einem derartigen Flat File

werden Sie zunächst je ein Feld für den CD-Titel, den Interpreten, die Gesamtlaufzeit etc. definieren.

Wenn Sie auch die Musikstücke, die sich auf der CD befinden, speichern wollen, zeigen sich die Mängel der unstrukturierten Datenbanken. Da die Länge für jeden Satz gleich ist, können Sie nur so viele Musiktitel pro CD abspeichern, wie Sie entsprechende Felder im Datensatz definiert haben. Definieren Sie 10 Felder, werden Sie spätestens bei der ersten Doppel-CD nicht mehr alle Musikstücke erfassen können.

Definieren Sie 40 Felder, so verschenken Sie bei den meisten CDs unnötig Speicherplatz. Bei zweidimensionalen Modellen fehlt es an der Möglichkeit, das wiederholte Auftreten von Daten, die zu demselben Datensatz gehören, in der Datenbank abzubilden. Man müsste dafür zu den zwei Dimensionen 'Breite' und 'Länge' noch wenigstens eine dritte Dimension hinzufügen.

2.1.2.1 Hierarchische Datenbanken

Bei diesen Datenbanken werden die Informationen in Segmente aufgeteilt, die in einer Hierarchie angeordnet sind. Die Hierarchie kann normalerweise bis zu 15 Stufen umfassen. In der obersten Ebene kann nur ein Segmenttyp stehen; man nennt ihn das ROOT-Segment (Root, engl. für Wurzel). In allen anderen Stufen können Elemente vom gleichen Typ pro Datensatz mehrfach vorkommen (Twin-Segmente, von Twin, engl. für Zwilling).

In diesen Ebenen kann es mehrere unterschiedliche Segmenttypen geben (Sibling-Segmente, von Sibling, engl. für Geschwister). Segmenttypen, von denen ein anderer Segmenttyp abgeleitet wird, heißen Parent- (engl. für Eltern), die abgeleiteten Typen Child-Segmente (engl. für Kind). Abbildung 1 zeigt ein Beispiel für eine hierarchische Datenbank. Bei sämtlichen Zugriffen auf die Daten müssen Sie über das ROOT- Segment einsteigen, d.h., sie müssen zunächst den CD-Titel suchen, zu dem Sie weitere Daten abfragen möchten.

Von dort aus können Sie alle Musiktitel, die sich auf der jeweiligen CD befinden, abfragen. Von den Musikstücken ausgehend können Sie deren Komponisten, Texter und Interpreten feststellen. Ein direktes Suchen nach Musikstücken, um die CDs zu ermitteln, auf denen sie sich befinden, unterstützt eine hierarchische Datenbank nicht. Vielmehr müssten

Sie die ROOT-Segmente aller CDs lesen, um von dort aus in den Musikstücken nach dem jeweiligen Titel zu suchen.

Weiterhin kann diese Datenbank nur '1:n-Beziehungen' (sprich: Eins zu En) abbilden. Viele Musikstücke werden sich auf mehreren CDs befinden, haben aber immer denselben Titel, denselben Texter, denselben Komponisten. Trotzdem werden diese Daten pro Auftreten des Titels einmal abgespeichert. Das führt zu Datenredundanzen, die neben der Speicherplatzverschwendung als Gefahr mit sich bringen, dass die Daten zu einem Musikstück an unterschiedlichen Stellen in der Datenbank voneinander abweichen.

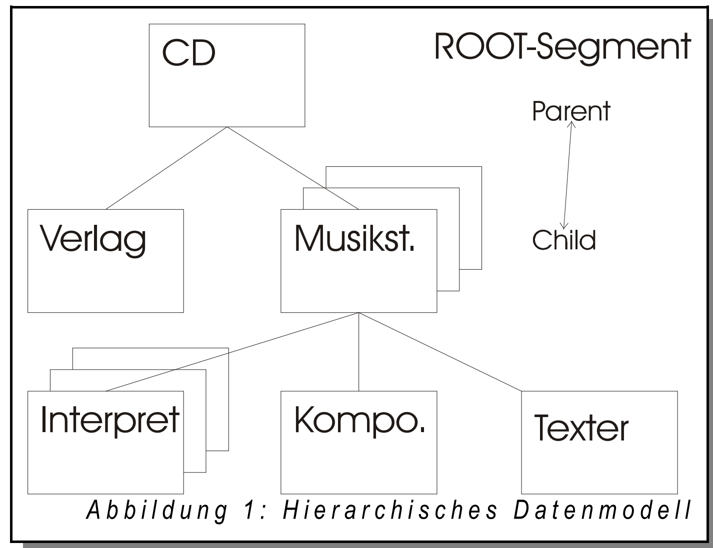


Abbildung 1: Hierarchisches Datenmodell

IMS, das Information Management System von IBM, das auch in der Verwaltung und bei Banken genutzt wird, ist ein solches hierarchisches System. Um das 'Navigieren' zu den gesuchten Daten über das ROOT-Segment für Standardverarbeitungen umgehen zu können, wurden von IBM Sekundärindexe oder Sekundäreinstiege geschaffen. Durch sie kann man beispielsweise direkt bestimmte Musikstücke ermitteln, um von dort auf die zugehörigen CD-Titel zuzugreifen. Hierarchische Datenbanken stellen die Verknüpfung zwischen Parent- und Childsegmenten über direkte 'Pointer' her. Ein Pointer ist nichts anderes als der Verweis auf die exakte Position, die ein Datensatz in einer Datei hat.

Das hat den Vorteil, dass einem gefundenen Parentsegment die Position der zugehörigen Childsegmente 'bekannt' sind. Abhängige Daten können direkt gelesen werden; ein Zusammensuchen ist unnötig. Die Performancevorteile hierbei liegen auf der Hand.

2.1.2.2 Netzwerkmodelle

Die Erweiterungen von IMS durch Sekundäreinstiege bringt es einen Schritt an das Netzwerkmodell heran. Dieses basiert ebenfalls auf der Segmentierung von Datensätzen und der direkten Verbindung zusammenhängender Daten durch Pointer. Jedoch ist ein Einstieg in die Datenstruktur über jeden Segmenttyp möglich. Datenredundanzen werden vermieden, da - in unserem Beispiel - jeder Musiktitel nur noch einmal abgespeichert wird und Verweise auf die CD-Titel, denen er zugeordnet wird, enthält. Man spricht hierbei auch von einer m:n-Beziehung ('Em zu En').

Netzwerkmodelle sind die Grundlage von Datenbanken nach CODASYL, beispielsweise UDS von SIEMENS.

2.1.2.3 Relationale Datenbanken

Diese Datenbanken basiert auf Gedanken von E. F. Codd[2], der 1970 eine wissenschaftliche Arbeit zum Thema veröffentlichte. Sie bestehen aus Tabellen, die für sich genommen, wieder einfache, zweidimensionale Gebilde sind. Sie werden jedoch durch Integritätsregeln so miteinander verknüpft, dass eine redundanzfreie, dem Netzwerkmodell nicht unähnliche Struktur entsteht.

Da die Verknüpfung der Tabellen jedoch nur über abstrakte Werte und nicht über Pointer erfolgt, bestehen sehr flexible Möglichkeiten der Abfrage. Gleichzeitig ist der Ressourenverbrauch höher als z.B. bei hierarchischen Systemen. Das Relationale Modell ist die Grundlage von SQL und wird daher in einem eigenen Hauptkapitel eingehend beschrieben.

2.1.2.4 Objektorientierte Datenbanken

Bei objektorientierten Datenbanken handelt es sich um eine sehr neuartige Technologie, welche die strikte logische Trennung von Daten und Programmanweisungen aufhebt. Datensätze bekommen die Programme, mit denen sie bearbeitet werden, als Methoden 'beigeheftet'. Datenobjekte werden nicht mehr durch direkte Abfrage oder Manipulation der Datenfelder bearbeitet, sondern durch das Versenden von 'Nachrichten' an ihre Methoden.

Diese Kapselung, die das Programm völlig von der tatsächlichen Speicherform von Informationen loslöst, macht "Vererbung" möglich. Auf die CD-Datenbank bezogen ließe sich beispielsweise eine Objektklasse 'CD' definieren, die alle

[2] Edgar F. Codd, Preisträger des Turing Award 1999 für seine Grundlagenarbeit in der Datenbanktheorie.

grundlegenden Informationen über CDs beinhaltet (CD-Titel, Anzahl und Name der Musikstücke, Verlag etc.). Davon könnte man die Klassen 'Mehrfach-CD', 'Einzel-CD' und 'Maxi- Single' ableiten, die zusätzliche eigene Methoden implementiert bekommen, beispielsweise für die Abfrage und Pflege von Länge und Titelanzahl der jeweiligen Einzel-CDs.

Diese abgeleiteten Typen können nun als 'CD' angesprochen und auch zusammengefasst bearbeitet werden. Das Programm, das diese Verarbeitung auslöst, braucht die Definitionen der abgeleiteten Klassen nicht zu kennen. Andererseits besteht die Möglichkeit, alle nur für eine der vererbten Klassen anfallenden Daten über die Ansprache als 'Mehrfach-CD' oder 'Maxi- Single' abzuarbeiten. Es gibt seit einiger Zeit verschiedenartige Ansätze, Kapselung und Vererbung in Datenbanken zu implementieren. Interessant sind Tendenzen, relationale Datenbanken einschließlich der Abfragesprache SQL objektorientiert zu erweitern.

Da sich aber noch keine unabhängigen Standards abzeichnen, soll auf diese neue Technologie hier nicht eingegangen werden.

Sie kennen nun die Unterschiede, die zwischen verschiedenen Datenbankkonzepten bestehen und können einzelne Systeme den Konzeptionen zuordnen.

3 Relationale Datenbanksysteme

3.1 Entstehung

Wie schon erwähnt basiert das relationale Datenmodell auf einer wissenschaftlichen Arbeit von E. F. Codd aus dem Jahre 1970. Eine Relation in diesem Modell kann man als simple, zweidimensionale Tabelle darstellen. Die Datenbank besteht aber aus einer Vielzahl von Tabellen, die in der Datenbankdefinition über Werteinhalte der Spalten verknüpft werden.

Die meisten Datenbank-Management-Systeme (DBMS), die sich 'relational' nennen, sind streng genommen nur tabellenorientiert. Denn die relationale Datenbank ist erst das, was auf dem DBMS für ein Anwendungsprogramm konkret implementiert wird - sofern es den Regeln relationaler Datenbanken entspricht.

Es ist ebenso möglich, ein solches DBMS zur Simulation komfortabler ISAM-Dateien oder zur Abbildung eines hierarchischen Systems zu verwenden. Allerdings ist das nicht zu empfehlen, da im ersten Fall nur ein Bruchteil der Möglichkeiten genutzt wird und im zweiten Fall gravierende Unverträglichkeiten der beiden Systeme zu Tage kommen.

3.2 IBM und System/R

In den 70er Jahren wurde in den Labors von IBM auf der Basis von Codds Überlegungen ein DBMS entwickelt, mit dem man die Realisierbarkeit und den Nutzen relationaler Datenbanken ermitteln wollte. Es trug den Namen System/R. Gleichzeitig wollte man dem Endanwender für individuelle *ad hoc*-Abfragen und dem Programmierer für die Programmentwicklung weitgehend dieselbe Abfragesprache zur Verfügung stellen.

3.3 Probleme bestehender Systeme

Ein Handlingproblem der bisherigen DBMS (sei es hierarchisch oder als Netzwerkmodell) war die Implementation ihrer Aufrufe in den benutzten Hochsprachen. Beim Umwandeln der Programme wurde höchstens festgestellt, ob den Datenbankaufrufen die Parameter des korrekten Typs in der korrekten Zahl und Reihenfolge übergeben wurden.

Die inhaltliche Konsistenz der Abfragen blieb den Compilern verborgen. Somit konnten viele logische Fehler in den benutzten Aufrufen erst zur Laufzeit des fertig umgewandelten und gelinkten Programms festgestellt werden. Auch das Ergebnis der Abfragen wurde erst zur Laufzeit überprüfbar, was die Produktivität der Entwickler natürlich beeinträchtigte.

3.4 Symbiose

IBM entwickelte für System/R eine Abfragesprache, die komplett in jede beliebige höhere Sprache - selbst in Assembler - eingebettet werden kann. Durch einen Precompiler wird die inhaltliche Konsistenz der Abfragen überprüft. Gleichzeitig werden sie in Aufrufe von Datenbankschnittstellen umgewandelt, die der eigentliche Compiler der Hochsprache übersetzen kann.

Weiterhin bekam der Entwickler die Möglichkeit, die Abfragen vor der Codierung im Programmsource in einem interaktiven Tool auszuprobieren. Auch von anderen Seiten - der ETH Zürich beispielsweise - wurden Versuche unternommen, relationale Abfragesprachen in eine höhere Programmiersprache zu integrieren. Aber die hierbei entstandenen Ergebnisse wie PASCAL/R oder Modula/R haben keinen nennenswerten Einfluss gefunden.

Die relationale Sprache von IBM hieß zunächst Sequel, nämlich structured english query language. Aber das Ziel, eine in verschiedene Kultursprachen übersetzbare Datenbanksprache zu schaffen, ließ sich nicht erreichen. Sehr bald wurde die Sprache in SQL, also structured query language, umbenannt.

3.5 Beschreiben statt Navigieren

Im Gegensatz zur Sprache DL/I, die IBM für das hierarchische IMS entwickelt hat, ist SQL eine deskriptive Sprache. Der Programmierer muss sich nicht mehr mit GET-UNIQUE und GET-NEXT-Befehlen durch die Datenbankstruktur hindurch navigieren, sondern er beschreibt, aufgrund welcher Tabellen und welcher Beziehungen der Tabellen zueinander welche Informationen vom DBMS geliefert werden sollen.

Das DBMS entscheidet nun selbst, u.U. erst zur Laufzeit, welche Zugriffe in welcher Reihenfolge durchgeführt werden müssen, um die Informationen zusammenzutragen. Früher waren viele Datenbankabfragen notwendig, um mit vielen

Programmanweisungen in Schleifenstrukturen mit logischen Verzweigungen und weiteren Abfragen Ergebnisse über komplexe Zusammenhänge zu ermitteln.

Bei SQL wird im Idealfall eine komplexe Datenbankabfrage eine Ergebnismenge zur Verfügung stellen, die nur noch in einer einfachen Schleife Element für Element abgearbeitet werden muss.
Standards

SQL entwickelte sich recht schnell zu einer sehr beliebten und werbewirksamen Abfragesprache. Viele Softwarehäuser erkannten die Zeichen der Zeit und ließen ihre altbekannten DBMS in Richtung 'Relational' mutieren, indem sie ein SQL-Tool aufpfropften.

3.6 Implementationen

Andere Firmen entwickelten völlig neue Relationale DBMS (RDBMS) auf der Basis von SQL. Dabei entwickelten sich verschiedene Erweiterungen dessen, was IBM in den ersten Implementationen vorlegte. Da die Ära der PCs und Workstations erst nach der Entwicklung der relationalen Datenbanken begann, wurden auf diesen Hardwareplattformen keine nennenswerten hierarchischen Datenbanksysteme mehr realisiert.

Die wesentlich flexiblere relationale Struktur setzte sich dort als Stand der Technik durch, wenngleich SQL auf der PC-Ebene in der Vergangenheit nur eine untergeordnete Rolle gespielt hat. Viele Hersteller begannen, ihre RDBMS möglichst kompatibel auf möglichst viele Plattformen zu migrieren. So entwickelten sich einzelne DBMS bald zu de-facto-Standards, zum Beispiel das vielzitierte ORACLE.

In einer Reihe von Fällen wird heutzutage eine Programmiersprache der 4. Generation (4GL) zusammen mit einem RDBMS im Paket verkauft. Man kann aber - SQL sei dank - die 4GL meistens auch mit anderen Datenbanken kombinieren oder die Datenbank von klassischen Programmiersprachen wie COBOL aus ansprechen.

3.7 Normierungen

Die ersten Sprachbeschreibungen, damals noch unter dem Namen SEQUEL, wurden Mitte der 70-er Jahre veröffentlicht. Auf deren Basis erschien 1987 ein ISO-Dokument, mit dem die 'offizielle' Sprachdefinition (ISO 9075:1987, auch

SQL-87 genannt) erstellt wurde. Sie wird oft als 'SQL Level 1' bezeichnet.

1989 erschien die nächste Fassung (ISO 9075:1989), die als SQL-89 noch heute den gängigsten Standard setzt. Sie beschreibt unter dem Namen 'SQL Level 2' die in der Norm definierten Erweiterungen. Die in der letzten Beschreibung SQL-92 (ISO/IEC 9075:1992) festgelegten Standards, die unter DIN 66315 auch in Deutschland angenommen wurden, sind jedoch heute in den meisten RDBMS noch Zukunftsmusik.

3.8 Erweiterungen

Die Hersteller versuchten seit SQL-87, einerseits die Forderungen der Norm nach Möglichkeit zu erfüllen, aber gleichzeitig durch Erweiterungen der Sprache leistungsfähigere Systeme anzubieten, als die Konkurrenz es tut. Die jeweils folgenden ISO-Normierungen griffen von diesen Erweiterungen immer einzelne, zweckmäßige Implementationen heraus und erhoben sie zum Standard.

Daher mussten Anbieter, die einen anderen Weg zur Implementierung beispielsweise von Datumstypen gegangen sind, versuchen, irgendwie zugleich kompatibel zu ihren eigenen Vorversionen und zum neuen Standard zu sein. Viele Hersteller werben für ihre Produkte aber mit 'ORACLE-Kompatibilität' oder 'DB2-Kompatibilität'.

3.9 Grundlagen

Der 'Erfinder' der relationalen Datenbanksysteme, E. F. Codd, hat in seinen Veröffentlichungen[3] 12 Regeln definiert, die ein relationales DBMS zu erfüllen hat. In der Praxis bleibt die vollständige Erfüllung schon aus Gründen der Realisierbarkeit unerreichbar, aber anhand der Regeln kann man leicht das Konzept der RDBMS verstehen.
Datentransparenz

Ein wichtiges Erkennungsmerkmal für RDBMS ist die Datentransparenz, die in verschiedenen Aspekten schon in den 12 Regeln erwähnt wurde. Datentransparenz ist eigentlich ein irreführender Begriff, denn es sind nicht die Daten, die transparent, also durchsichtig, sind, sondern ihre physische Speicherung.

[3] Codd, E. F.; A relational model for large shared data banks; CACM, Vol. 13, no. 6; June 1970

Das Programm übergibt dem DBMS die Daten in einer standardisierten Form und erhält sie in derselben Form zurück. Wie und wo sie abgespeichert werden ist beim Codieren völlig uninteressant. Bei DB2 wird dies besonders bei Spalten in Datums- und Uhrzeitformaten deutlich. Ein Datumstyp in wird als Textkette wie eine STRING-Variable übergeben und empfangen, beispielsweise als '17.08.1994'.

Es ist dabei auf die genaue Einhaltung einer gültigen Form zu achten: z.B. Tag und Monat zweistellig, Jahr vierstellig, mit Punkten getrennt. DB2 empfängt diesen String und unterzieht ihn einer Plausibilitätsüberprüfung: Das Datum '29.02.1973' würde abgewiesen, da 1973 kein Schaltjahr war.

Ist das Datum in sich korrekt, wird es als numerisch codierter Wert abgespeichert und beim Auslesen anhand des definierten Standard- Datumsformates wieder in einen String gewandelt. Weiterhin versteckt sich hinter dem Begriff Transparenz, dass der Name und Aufbau der Datei, in der die Tabellen gespeichert werden, völlig irrelevant sind. Bei DB2/MVS kann beispielsweise ein sog. 'Tablespace' als physische Datei zum Aufnehmen von Tabellen eine oder viele Tabellen beinhalten, bestimmte Segmentierungen für die Tabellen aufweisen und bestimmte Freiräume zum Einfügen neuer Daten aufweisen.

Zum schieren Funktionieren der Anwendungen ist es völlig gleichgültig, wie diese Werte im Einzelnen aussehen. Die Daten werden in abstrakten Tabellen abgelegt, die von der physischen Speicherform unabhängig sind.

Bei DB2 for AIX und anderen Plattformen der DB2 Universal Database geht die Transparenz sogar noch weiter: Beim Anlegen der Datenbank braucht der Administrator nur noch den zu benutzenden Verzeichnispfad zu bezeichnen.

Wieviele Dateien mit welchen Namen und welchen internen Strukturen angelegt werden, kann das Datenbanksystem völlig selbständig entscheiden.

3.10 Verteilte Daten

Im Zuge der verteilten Datenbanken geht der Transparenzbegriff noch einen Schritt weiter: Durch die *Distributed Relational Data Architecture* (DRDA) können - beispielsweise - in einem DB2/2- System Tabellen definiert werden, die in

einem ganz anderen Trägersystem auf einem ganz anderen Computer physisch gespeichert sind.

Für Anwender und Programmierer verhalten sich diese Tabellen aber ganz genau so, wie die lokal vorhandenen Daten. DRDA ermöglicht durch die implementierte Datentransparenz, dass z.B. ein DB2/2-System als 'Gateway' zwischen einer PC-Anwendung als Client und einem DB2/MVS als Server fungiert. Andererseits können in zukünftigen Versionen die Daten vom MVS-System in Abfragen beliebig mit lokalen Daten der OS/2-Ebene kombiniert werden.

Die Verteilung der Daten zwischen den Ebenen ist hierdurch so flexibel, dass sogar bestehende Datenbanken zwischen den Systemebenen umverteilt werden können, ohne dass im Normalfall Änderungen an Anwendungsprogrammen notwendig sind.

Sie kennen nun das Konzept und die Vorteile relationaler Datenbanksysteme und können diese anhand der 'Zwölf Regeln' erkennen.

4 'Zwölf Regeln'

Codds 12 Regeln ist zunächst, oft auch als 'Regel 0', die Bedingung vorangestellt, dass jedes als RDBMS vertriebene System in der Lage sein muss, seine Datenbanken alleine mit seinen relationalen Regeln zu verwalten.

Die weiteren Regeln lauten wie folgt:

1. **Die Informationsregel**
   Die gesamte Information einer relationalen Datenbank wird auf logischer Ebene in genau einer Art und Weise dargestellt - nämlich als Wertetabelle.

2. **Regel des garantierten Zugriffs**
   Für jeden atomaren Wert einer relationalen Datenbank wird ein logischer Zugriff garantiert: Mittels Rückgriff auf eine Kombination aus Tabellennamen, Primärschlüsselwerten und Spaltennamen.

3. **Systematische Behandlung von NULL-Werten**
   NULL-Werte (also nicht die leere Zeichenkette bzw. die Zahl Null) werden in vollständigen RDBMS systematisch zur Darstellung fehlender oder nicht verwendbarer Informationen benutzt - unabhängig von deren Datentyp.

4. **Auf dem relationalen Modell basierender dynamischer online- Katalog**
   Die Datenbankbeschreibung wird auf logischer Ebene wie andere Daten dargestellt, so dass zugriffsberechtigte Anwender mit derselben relationalen Sprache auf sie zugreifen können wie auf normale Daten.

5. **Verständliche Sprachenuntermengen-Regel**
   Ein relationales Modell kann mehrere Sprachen und verschiedene Bildschirmanweisungen unterstützen. Allerdings muss es mindestens eine Sprache geben, deren Sätze nach einer wohldefinierten Syntax als Zeichenketten konstruiert werden können, und welche die folgenden Möglichkeiten vollständig unterstützt: Datendefinition, Viewdefinition, Datenmanipulation, Integritätsbeschränkungen, Zugriffsberechtigung, Transaktionsgrenzen.

6. **View-Updating-Regel**

Sofern ein Update einer View theoretisch möglich ist, wird er auch vom System zugelassen.

7. **Einfügen, Löschen und Update auf hochsprachlicher Ebene**
   Die Fähigkeit, eine Basisrelation oder eine abgeleitete Relation als einen einzigen Operanden zu behandeln, bezieht sich nicht allein auf den Datenzugriff, sondern auch auf das Löschen, Einfügen und Aktualisieren von Daten.

8. **Unabhängigkeit von der physischen Datenspeicherung**
   Anwenderprogramme und Bildschirmaktivitäten bleiben logisch unbeeinträchtigt von der Änderung in der Speicherrepräsentation und in den Zugriffsmethoden.

9. **Unabhängigkeit logischer Daten**
   Anwenderprogramme und Bildschirmaktivitäten bleiben logisch unbeeinträchtigt, wenn in den Basistabellen informationserhaltende Veränderungen vorgenommen werden, die es theoretisch ermöglichen, eine Beeinträchtigung auszuschließen.

10. **Integritätsunabhängigkeit**
   Spezifische Integritätsbeschränkungen für eine bestimmte relationale Datenbank müssen in einer relationalen Datensprachenuntermenge definiert und im Speicher abgelegt werden können, nicht in den Anwendungsprogrammen.

11. **Verteilungsunabhängigkeit**
   Ein RDBMS besitzt Verteilungsunabhängigkeit.

12. **Regel der Unzerstörbarkeit**
   Verfügt ein relationales System über eine Sprache auf niedriger Ebene (single-record-at-a-time), so kann diese Sprache nicht dazu verwendet werden, die in einer relationalen Sprache auf höherer Ebene (multiple-record-at-a-time) definierten Integritätsregeln zu untergraben oder zu umgehen.

Diese zwölf Regeln werden von den meisten RDBMS mehr oder weniger erfüllt.

Manche Fachleute behaupten zwar, ein Datenbanksystem sei erst dann relational, wenn alle Regeln erfüllt sind, aber die Grenzen sind wohl fließend. Schließlich sind die hier aufge-

führten Regeln schon 'abgeschwächt', damit sie überhaupt erfüllbar werden.

5 Datenmodellierung

Bei der Entwicklung eines Programmes kann man grob zwischen der daten- und der ablauforientierten Vorgehensweise unterscheiden.

Im letzteren Fall werden die rein funktionalen Anforderungen an das zu entwickelnde Programm herangezogen, um auf dieser Basis ein Sollkonzept zu erstellen. Das Datenmodell wird erst in Anlehnung an die Grobstruktur des Programmes entwickelt.

Diese Vorgehensweise schafft immer dann Probleme, wenn sich durch Veränderungen in den Anforderungen die Struktur des Programmes nachträglich ändert. Bei hierarchischen Datenbanksystemen war sie teilweise unumgänglich, da das Navigieren in den Daten anhand der häufigsten Zugriffswege optimiert werden mußte.

Relationale Datenbanken sind jedoch in ihren Abfragemöglichkeiten so flexibel, dass man das Datenmodell nur in absoluten Ausnahmefällen an die Programmstruktur anpassen sollte.

Schließlich heißt es Datenverarbeitung und nicht Maskenspeicherung. Außerdem ist ein Datenmodell, das sich möglichst nah an der abzubildenden Realität orientiert, langlebiger, denn es unterliegt im Laufe der Zeit weniger Veränderungen als die Anforderungen an die Software.

Datenorientierte Vorgehensweise hat folgende Vorteile:

- ⇩ Bei der datenorientierten Vorgehensweise resultieren fast zwangsläufig integrierte, vielfach benutzbare, redundanzfreie Datenbanken.
- ⇩ Die Vorgehensweise führt auch dann zum Ziel, wenn Funktionen vorerst einmal gar nicht ermittelt werden können. Dies ist immer dann der Fall, wenn eine Basis für die Beantwortung von spontanen, nicht voraussehbaren Fragestellungen geschaffen werden soll.
- ⇩ Die Vorgehensweise vermag jederzeit einen Überblick bezüglich der datenspezifischen Aspekte einer Unternehmung zu gewährleisten. Dieser Sachverhalt wird in Zukunft mit der verteilten Datenverarbeitung außerordentlich an Bedeutung gewinnen.

Daher sollte die Erstellung des Datenmodells in der Realisierungsphase eines Projektes der erste Schritt sein. Die Datenmodellierung ist üblicherweise eine Aufgabe, die vom Anwendungsentwickler zusammen mit der Datenbankadministration erfüllt wird.

Sorgfältig ausgeführt schafft man sich so eine Datenbasis, die während der Programmentwicklung nur noch in Details geändert werden muss und vielseitig nutzbar ist. Eine beispielhafte Datenmodellierung mit einem Entity-Relationship-Model wird in den folgenden Abschnitten kurz erläutert.

5.1 Entitäten

Der erste Schritt besteht aus dem Suchen von Entitäten. Eine Entität ist ein Element einer Entitätsmenge. Eine Entität ist ein individuelles und identifizierbares Exemplar von Dingen, Personen oder Begriffen der realen oder der Vorstellungswelt, für welches anwendungsbezogene Informationen von Bedeutung sind.

Eine Entitätsmenge vereint diejenigen Entitäten, die über die gleichen Attributarten beschrieben und über die gleichen Schlüsselbegriffe identifiziert werden. Auf die schon angedachte CD-Datenbank bezogen wäre beispielsweise jedes Musikstück eine Entität, ebenso wie jeder Komponist und jede CD.

Die gesamten Musikstücke würden eine eigene Entitätsmenge bilden, da sie über denselben Schlüssel zu bezeichnen sind (Titel, ggfls. Komponist) und mit denselben Attributarten beschrieben werden (wieder Titel und Komponist, weiterhin Texter, Dauer, Jahr der ersten Veröffentlichung etc.). Weitere Entitätsmengen wären beispielsweise die Menge aller CDs, die Menge aller Komponisten, die Menge aller Interpreten usw.

Es macht auch bei kleinen und 'überschaubaren' Datenmodellen Sinn, sie in Form eines Diagrammes darzustellen. Das ist spätestens dann notwendig, wenn man anderen Personen - vielleicht sogar EDV-Laien - die Zusammenhänge erklären will. Die Entität wird als Kasten dargestellt.

Der Name befindet sich im Symbol. An der Seite werden die Attribute der Entität notiert, allen voran die Schlüsselattribute, die durch einen senkrechten Strich gekennzeichnet

werden. Aus Gründen der Übersichtlichkeit kann man in Diagrammen diese komplette Darstellung auch verkürzen, indem man die Attribute wegläßt oder auf die Schlüsselattribute beschränkt.

Abbildung 2: Die Entität "CD"

Unabhängig davon sollte man einen Entitäten- bzw. Attributekatalog erstellen, aus dem die Entitäten und Attribute textlich beschrieben hervorgehen. Häufig wird an dieser Stelle schon Rücksicht auf spätere physische Strukturen im DBMS genommen. Bei pragmatischer Behandlung ist das auch korrekt, denn man kann sich so teilweise Arbeitsschritte ersparen.

Aber in diesem Beispiel wird bewusst strikt zwischen (abstrakten) Attributen und (konkreten) physischen Datentypen unterschieden. Der Entitätenkatalog könnte für unsere CD-Verwaltung etwa so aussehen:

Tabelle1: Entitätenkatalog

Entität	Schlüssel	Attribute	Beschreibung
CD	**CD-Titel, Best. Nr**	Verlag, Interpret, Komponist, Titelzahl, Musikart, Preis	Je ein Element pro CD, Doppel-CD oder Maxi-Single
Musikstück	**Musiktitel,**	Texter, Länge, Musikart, Erschein-Datum	Je ein Element pro Musiktitel
Komponist	**Name**	Geb.Dat, Todestag	Je ein Element pro Komponist, von dem Musikstücke erfaßt sind
Texter	**Name**	Geb.Dat, Todestag	Je ein Element pro Texter
Interpret	**Name**	Geb.Dat, Todestag, Fanclub, Adresse, Instrument_MM Sänger_MM	Je ein Element pro Musiker, von dem Aufnahmen erfaßt sind.

Diese Aufzählung ist keinesfalls abschließend und soll nur ein Denkmodell darstellen.

5.2 Relationships

Als nächster Schritt werden die Beziehungen (engl. relationships) zwischen den Entitätsmengen ermittelt und in Form eines Diagramms dargestellt. Bei den Beziehungen muss man grob zwischen zwei Arten mit je zwei Unterarten, also insgesamt vier Ausprägungen, differenzieren:

Tabelle 2: Relationsarten

einfache Beziehung	**-->**	einfache muss Beziehung
	--)	einfache kann Beziehung
mehrfache Beziehung	-->>	mehrfache muss Beziehung
	--))	mehrfache kann Beziehung

Eine Beziehung besteht immer zwischen genau zwei Entitäten. Sie besitzt in jeder Richtung (von Entität A nach Entität B und umgekehrt) eine der genannten Ausprägungen. Im Diagramm werden die Beziehungen durch Linien dargestellt, welche die Entitäten miteinander verbinden.

Die Ausprägung der Beziehung wird durch Symbole an den Enden verdeutlicht. Eine einzelne Pfeilspitze steht für 'einfache muss-Beziehung', zwei Pfeilspitzen für 'mehrfache muss-Beziehung'. Analog zu den Pfeilspitzen werden Bögen benutzt, um 'kann-Beziehungen' darzustellen. Abbildung 3 zeigt die Beziehung zwischen Komponist und Musiktitel.

Jeder Musiktitel muss zwangsläufig einen Komponisten haben (wobei wir der Einfachheit des Denkmodells halber unterstellen, dass es keine Musiktitel mit mehreren

Komponisten gibt). Daher trägt der Pfeil, der vom Musiktitel auf den Komponisten verweist, eine einzelne Spitze. In die andere Richtung gilt, dass ein Komponist erst dann für uns interessant wird, wenn wir wenigstens einen seiner Musiktitel in der Sammlung haben. Daher auch hier die 'muss-Beziehung'.

Gleichzeitig kann er aber auch eine Vielzahl von Titeln geschrieben haben, die sich auf unseren CDs befinden, daher die doppelte Spitze. Die Texte, die immer in Uhrzeigerrichtung gelesen werden (das soll durch die gestrichelten Bögen deutlich gemacht werden), beschreiben dem Betrachter - wenn nötig - den Inhalt der Beziehung. Das gleiche Spiel machen wir nun mit der Beziehung zwischen Musiktitel und Texter.

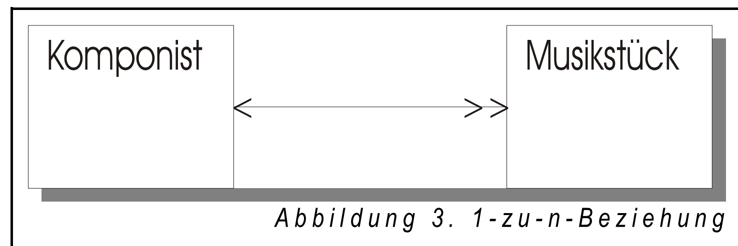


Abbildung 3. 1-zu-n-Beziehung

Da es Musiktitel gibt, die rein instrumental ohne Gesang komponiert wurde, muss ein Musiktitel keine Beziehung zu einem Texter haben. Falls ein Text existiert, stammt dieser (wieder der Einfachheit halber) aus der Feder einer einzelnen Person. Wir haben also eine 'einfache kann-Beziehung'. Sie wird durch einen Bogen am Ende der Linie dargestellt. Umgekehrt gilt wieder, dass ein Texter erst dann in der Datenbank erfaßt wird, wenn wenigstens ein Titel von ihm getextet wurde.

Möglicherweise hat jedoch auch er eine Vielzahl von Titeln geschrieben. Somit haben wir auch hier eine 'mehrfache muss- Beziehung', die durch zwei Pfeilspitzen dargestellt wird. Die Vorteile des relationalen Systems kommen besonders deutlich bei der Beziehung zwischen CD-Titel und Musiktitel zum Vorschein. Jeder Musiktitel kann auf mehreren CDs vorhanden sein, jede CD kann mehrere Musiktitel enthalten. Da es keine leeren CDs gibt und uns nur die Musiktitel interessieren, die wir auf CD haben, handelt es sich in beiden Richtungen um eine 'mehrfache muss-Beziehung'.

Diese m:n-Beziehung (sprich: Em zu En) ist etwas, das gerade relationale DBMS darstellen können. Allerdings wird auch hier nur mit Wasser gekocht und die Beziehung muss,

damit man sie darstellen kann, 'aufgelöst' werden. Hierzu schafft man eine neue Entitätsmenge, man nennt sie auch assoziative Entität, deren einziger Lebenszweck darin besteht, mit jedem Element die Verbindung zwischen genau einem Musiktitel und einer CD abzubilden.

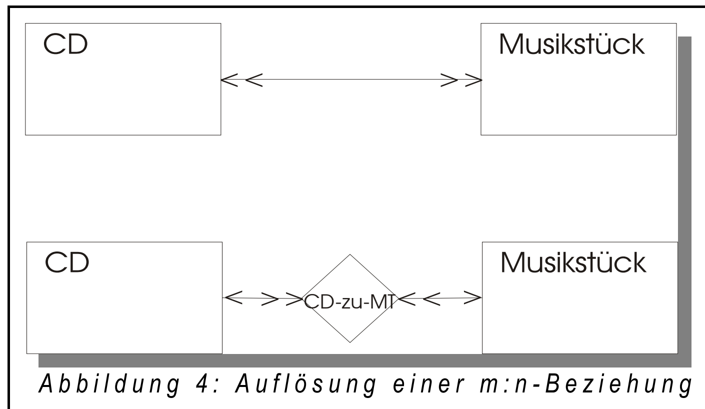


Abbildung 4: Auflösung einer m:n-Beziehung

Sie wird durch die Raute im Diagramm dargestellt. Da sie komplett von den beiden anderen Entitäten abgeleitet wird, braucht man sich - bis auf den Namen - in dieser Phase noch keine Gedanken über sie zu machen. Alternativ kann man diese assoziative Entität auch als 'vollwertige' Entität darstellen; dann muss man die Beziehungen in diesem Beispiel aber als 1:n-Beziehungen zwischen CD und 'CD-zu-MT' bzw. Musiktitel und 'CD-zu-MT' darstellen.

Die Beziehung zwischen Musiktitel und Interpret ist weniger trivial, als die bisherigen Beziehungen. Jeder Musiktitel kann von mehreren Interpreten aufgenommen worden sein. Jeder Interpret kann auch mehrere Musikstücke auf unseren CDs gespielt oder gesungen haben. Wenn man unter 'Interpret' auch 'Orchester' oder 'Band' verstehen will, dann kann man behaupten, dass jede Aufnahme exakt einem Interpreten zugeordnet werden kann, jeder Interpret aber mehrere Musikaufnahmen in unserer Sammlung haben kann.

Uns ist also bei der Beschreibung der Beziehungen aufgefallen, dass wir eine Entität übersehen haben: Die Musikaufnahme auf der CD selbst. Denn auf der CD auf der CD befindet sich nicht der Musiktitel, sondern eine Aufnahme des Titels (man könnte auch sagen: Eine Instanz des Titels). Also ist anstelle der gezeigten m:n-Beziehnung zwischen Musiktitel und CD eine neue Entität Aufnahme zu setzen. Da sich die Aufnahme nur noch auf einer CD befindet, nur einen Musiktitel gleichzeitig umfasst und nur ein Interpret (Sänger, Instrumentalist, Band, Orchester) ihr zugeordnet werden muss, hat sie nur 1:n-Beziehungen, die wir problemlos darstellen können.

Das gesamte Datenmodell der CD-Datenbank als Entity-Relationship- Diagramm (ERD) ist in Abbildung 5 dargestellt. Mit der Nummer 1. ist eine simple 1:n-Beziehung markiert. Da die Enden Pfeilspitzen tragen, ist sie in beide Richtungen eine 'muss- Beziehung'. Jeder Musiktitel hat genau einen Komponisten, jeder Komponist hat einen oder viele Musiktitel geschrieben. Eine Variante davon ist in Beziehung Nummer 2. zu sehen: Jeder Texter hat zu einem oder mehreren Musikstücken den Text geschrieben (sonst würden wir ihn nicht abspeichern), jedes Musikstück hat aber keinen oder genau einen Texter.

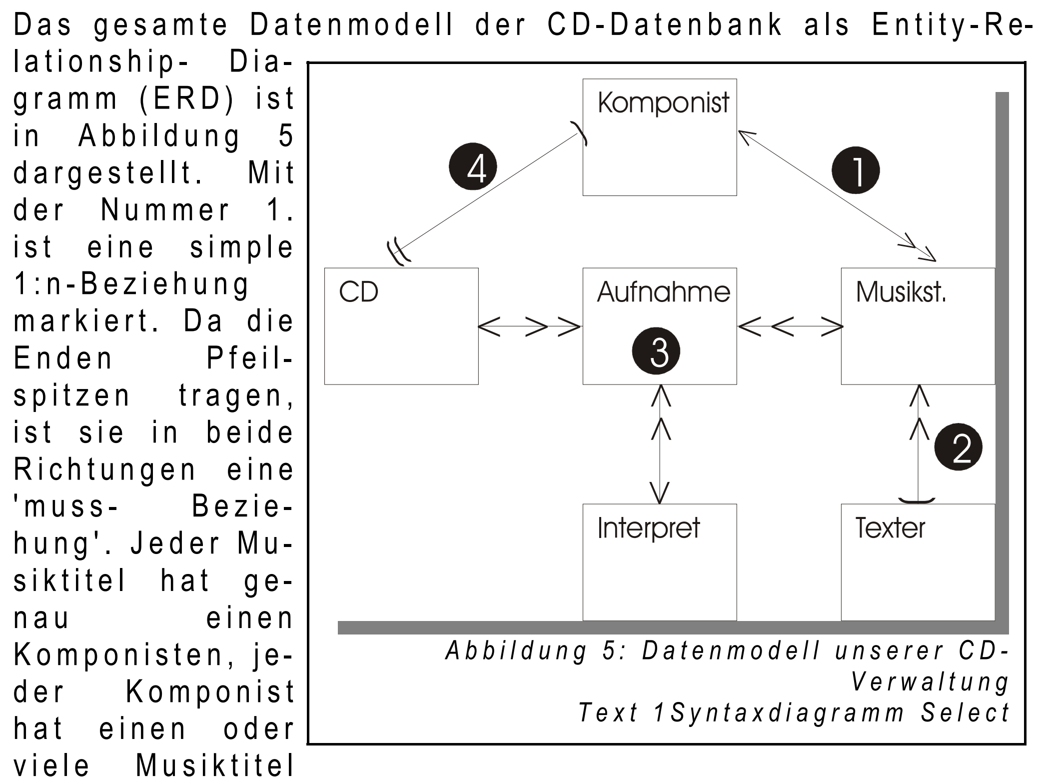


Abbildung 5: Datenmodell unserer CD-Verwaltung
Text 1Syntaxdiagramm Select

Die m:n-Beziehung Nummer 3., die sich zu einer vollwertigen Entität gemausert hat, ist bereits besprochen worden. Unter Nummer 4. ist eine Besonderheit zu sehen: Jede CD kann keinem oder genau einem Komponisten gewidmet sein. Jeder Komponist kann in der Datenbank keine, eine oder viele CDs haben, die ihm gewidmet sind. Die gleiche Konstellation ergäbe sich, wenn man die Beziehung zwischen Interpret und CD herstellen wollte.

5.3 Relationen

Der nächste Schritt bei der Erstellung einer Datenbank ist das Ableiten von Relationen aus den Entitäten. Vorher kann man jedoch schon einmal die Entitäten ansehen und sich erste Gedanken über die physische Implementation einzelner Attribute machen. Die Schlüsselbegriffe sind bisher immer die Namen bzw. Titel von CD, Musikstück, Interpret, Komponist oder Texter.

Personennamen sind von der Eindeutigkeit her nur eingeschränkt geeignete Schlüssel, aber als lange Textketten

performancemäßig ungünstig. Denn über die Schlüssel müsste das DBMS später die Beziehungen der Objekte ermitteln, und das geht über möglichst kurze und binäre Zahlenformate am schnellsten.

Hinzu kommt, dass Musikstücke und CDs häufig keine eindeutigen Titel haben und der Schlüssel daher aus dem Titel und der Bestellnummer bzw. dem Namen des Komponisten zusammengesetzt sein müsste, was die Performance noch weiter verschlechtern würde.

Es empfiehlt sich daher in diesen Fällen einen eigenen Primärschlüssel zu erfinden, der in Form von INTEGER oder SMALLINT-Spalten dargestellt wird. Da wir deutlich weniger als je 32000 CDs, Musikstücke, Komponisten etc. erwarten können wir ruhig SMALLINTEGER benutzen. Der Inhalt dieser Schlüssel müsste zwar beim Aufnehmen neuer Daten vom Anwendungsprogramm gepflegt werden, aber SQL bietet hier komfortable Lösungen.

Als Suchbegriff und Schlüsselkandidaten behalten wir die alten Schlüsselattribute natürlich im Auge. Das Ableiten der Relationen ist nun einfach. Man schreibt lediglich die Namen der Entitäten auf und listet dahinter in Klammern die Namen ihrer Attribute. Die Schlüsselattribute werden unterstrichen. Man kann in unserem Fall diesen einfachen Formalismus erweitern.

Wir kennzeichnen die 'künstlichen' Schlüssel (die SMALLINT-Werte) durch Unterstreichung und Fettdruck, die 'natürlichen Schlüssel' oder Schlüsselkandidaten durch Unterstreichung.. Es resultieren folgende Relationen:

CD(CD-Nr, CD-Titel, BestNr, Verlag, Interpret, Komponist, Titelzahl, Musikart, Preis)

MUSIKSTUECK(M-Nr, Musiktitel, Komponist, Texter, Laenge, Musikart, Erschein-Datum)

KOMPONIST(K-Nr, K-Name, K-GebDat, K-Todestag)

TEXTER(T-Nr, T-Name, T-GebDat, T-Todestag, Schriftsteller-MM)

INTERPRET(I-Nr, I-Name, I-GebDat, I-Todestag, I-Adresse, I-Fanclub, Instrument)

Neu hinzu kommt an dieser Stelle die neue Entität Aufnahme, die wir bei Auflösung der beiden m:n-Beziehungen schaffen mussten. Da jede Aufnahme in einem bestimmten

Kalenderjahr stattgefunden hat und die einzelnen Interpreten die Länge der Musikstücke variieren, können wir als Attribute der Aufnahme das Entstehungsjahr und die Länge festlegen.

Als Schlüssel definieren wir zunächst CD-Nr und M-Nr, da wir die Aufnahmen in unserer Datenbank anhand von CD-Nr. und Musiktitel indentifizieren können. Wir erweitern den Schlüssel noch um die I-Nr, damit der seltene Fall abgedeckt ist, dass ein und derselbe Titel auf derselben CD von verschiedenen Interpreten eingespielt wurde.

`AUFNAHME(CD-Nr, M-Nr, I-Nr, Track, Jahr, Laenge)`

5.4 Normalisieren

Im nächsten Schritt, dem Normalisieren, werden verschiedene Datenredundanzen erkannt und eliminiert. Insgesamt hat Codd fünf Stufen der Normalisierung entwickelt, auch 'Normalform' genannt, aber im täglichen Leben reichen die ersten drei völlig aus.

5.4.1 Erste Normalform

> In einer in der 1. Normalform (1NF) befindlichen Relation sind nur einfache, keine mengenmäßigen Attribute zugelassen.

Bei Darstellung in Tabellenform enthält jede Zelle, also jeder Schnittpunkt einer Spalte mit einer Zeile, genau einen definierten Wert.

Diese 1NF haben wir zumindest teilweise bereits bei der Entwicklung der Entitäten und der Schaffung der neuen Entität beachtet. Bezogen auf die Entität CD läge ein Verstoß gegen die 1NF vor, wenn wir ein Attribut 'Musiktitel' einführen würden, in dem alle auf der CD enthaltenen Titel über ihren Schlüssel bezeichnet werden.

Denkbare Verstöße gegen die 1NF lägen auch bei 'Interpret' vor, wenn das Attribut 'Fanclub' mehr als einen Fanclub pro Person bezeichnen können soll. Wollte man dies, so müsste man im relationalen Modell eine eigene Entität bzw. Relation 'Fanclub' entwickeln und in eine 1:n- Beziehung zu 'Interpret' stellen.

5.4.2 Zweite Normalform

> Eine Relation ist in der 2NF, wenn sie die 1NF erfüllt und jedes nicht dem Schlüssel angehörige Attribut vom gesamten Schlüssel voll funktional abhängig ist und nicht nur von Schlüsselteilen.

Bei einem derart einfachen Modell ist die 2NF schnell erfüllt. Ein Verstoß läge nur vor, wenn ein Schlüssel aus mehreren Attributen vorkäme.

In der betreffenden Relation müsste dann eine direkte Abhängigkeit zwischen einem Teil der Schlüsselattribute und einem nicht zum Schlüssel gehörenden Attribut bestehen. Sofern man in der Entität Aufnahme noch den Namen des Musikstücks oder des Komponisten abspeichern wollte, läge ein Verstoß vor.

Der Titel und Komponist sind schließlich bei ein und demselben Musikstück immer gleich und der Inhalt diese Attributes daher direkt von dem Teil des Schlüssels, der das Musikstück bezeichnet, abhängig.

5.5 Dritte Normalform

> Eine Relation ist in der 3NF, wenn sie die 2NF erfüllt und keine Attribute funktional von anderen Attributen abhängig sind, die nicht Teile des Schlüssels sind.

Ein solcher Verstoß kann auch in unserer Mini-Datenbank vorliegen. Die Attribute 'Adresse' in den Personen-Entitäten können und sollten in Teilattribute aufgespalten werden. Sie würden z.B. in die danach eigenständigen Attribute Straße, Plz und Ort zerfallen.

Spätestens seit dem 1.7.1993 ist aber - bezogen auf Adressen in Deutschland - hier ein Verstoß gegen die 3NF gegeben, denn jede Postleitzahl ist seit diesem Tag genau einem Ort zugeordnet. Ein Ort kann zwar mehrere Postleitzahlen haben, aber aus der Zahl lässt sich umgekehrt immer eindeutig der Ort ableiten.

Daher müsste man, wenn man nur inländische Adressen abspeichern wollte, eine neue Entität 'Ort' definieren und als Schlüssel die fünfstellige Postleitzahl festlegen. In den gespeicherten Adressen würde das Attribut 'Ort' wegfallen und aus der Postleitzahl und der neuen Entität abgeleitet werden. Es gibt aber mehrere Punkte, die eine solche kom-

plette Normalisierung in diesem Fall unnötig machen: Die Autogrammadresse und die Anschriften der Fanclubs sind wirklich nicht Sinn und Zweck der Datenbank, sondern eher nebensächliche Dinge, auf die man auch verzichten kann.

Weiterhin werden zwangsläufig auch ausländische Anschriften erfaßt, bei denen man sich nicht mehr auf die Eindeutigkeit der Postleitzahl verlassen kann. Also kann auf eine Einhaltung der 3NF hier mit gutem Gewissen verzichtet werden. Die vierte und fünfte Normalform sind im täglichen Leben verzichtbar und werden somit im Rahmen dieser Schulung nicht angesprochen.

5.6 Tabellen definieren

Im letzten Schritt müssen aus den Relationen Tabellen gemacht werden. Hierzu müssen einzelne Attribute aufgespalten werden, in unserem Fall die Adressen in der Entität 'Interpret'. Für die anderen Attribute muss man - wie in allen Datenbanken - die geeigneten Datentypen ermitteln. Bisher haben wir in den Relationen nur die Primärschlüssel definiert, also die Schlüssel, über die wir die Elemente selbst identifizieren.

Um die Relationships, also die Verbindungen zwischen den Entitäten, darstellen zu können, müssen wir zu den Attributen noch Fremdschlüssel aufnehmen. Zu diesem Zweck wird in der Relation, die eine 'zu-eins-Beziehung' zu einer anderen hat, der Schlüssel dieser anderen Relation als Attribut hinzugegeben. Die Beziehung 'CD zu Aufnahme' ist eine 'zu-n-Beziehung', da jede CD mehrere Aufnahmen enthalten kann.

Umgekehrt, als 'Aufnahme zu CD' ist es eine 'zu-eins-Beziehung', da jede Aufnahme genau einer CD zugeordnet wird. In der komplexen Beziehung zwischen CD, Musikstück und Interpret gehen von der 'Aufnahme', die den Knoten darstellt, drei solcher 'zu-eins-Beziehungen' aus. Da wir diese Relation aus einer assoziativen Entität entwickelt haben, sind die drei Schlüssel bereits in ihrer Zusammensetzung als Schlüssel der 'Aufnahme' benutzt und brauchen nichts hinzuzufügen.

Für die anderen Beziehungen müssen wir jedoch Fremdschlüssel hinzufügen: Der Musiktitel muss den Schlüssel des Komponisten als Pflichtfeld erhalten; ebenso der des Texters der aber auch leer bleiben kann. Eine ebensolche 'freiwillige' Verknüpfung besteht von der CD zum Komponis-

ten für den Fall, dass die CD einem Komponisten gewidmet ist. Der Rest ist Fleißarbeit.

Die Tabellendefinitionen als DB2 'Create'-Statements bekommen Schulungsteilnehmer in mehreren Dateien zur Verfügung gestellt; für das Selbststudium finden Sie die Definitionen im Anhang A. Die in DB2 verfügbaren Datentypen finden Sie im Anhang B.

6 DB2, SQL und SPUFI

Wenn gleich die Datenbank erfolgreich angelegt und gefüllt wurde, können wir mit den ersten Abfragen beginnen. SQL basiert auf der englischen Sprache. Mit EDV-Grundkenntnissen und Schulenglisch sind einfache Abfragen fast schon klarsprachlich verständlich.

6.1 SQL

Zunächst noch ein paar Worte über die Gliederung der Sprache.

Abbildung 6 zeigt die Dreigeteiltheit von SQL. Jeder der Sprachanteile erfüllt genau definierte Zwecke, die in den folgenden Abschnitten beleuchtet werden sollen.

- **DDL- Data Definition Language** Elemente dieses Sprachanteils werden Sie gleich sehen, denn die CREATE- Statements, mit denen Sie die Tabellen für unsere Schulung anlegen werden, sind sein Hauptbestandteil. Er dient zum Definieren und Modifizieren der Datenstrukturen.
- **DCL - Data Control Language** Dieser Sprachanteil dient in erster Linie der Verwaltung von Zugriffsrechten auf die Datenstrukturen. Allerdings gehören auch einzelne Befehle zur Steuerung der Zugriffe und zur Sicherung der Datenintegrität hier her.
- **DML - Data Manipulation Language** Mit einem Statement aus der DML wollen wir gleich die ersten SQL-Erfahrungen sammeln. Die DML stellt den wichtigsten Anteil dar, denn sie enthält alle Elemente zum Verändern und Abfragen von Dateninhalten.

6.2 DB2/MVS: SPUFI

DB2/MVS wird standardmäßig mit einem Programm namens SPUFI mit ausgeliefert. SPUFI bedeutet 'SQL-Ausführer mit Dateieingabe' (SQL processor using file input).

Der Aufruf von SPUFI erfolgt über den entsprechenden ISPF-Menüpunkt *DB2 interaktiv*. Näheres erfahren Sie bei den in Ihrem Rechenzentrum zuständigen SystembetreuerInnen.

6.3 DB2/UDB Server: db2-Commandline

Die DB2/UDB-Server werden mit einer DB2-Kommandozeile ausgeliefert. Diese Kommandozeile wird abhängig vom

benutzten Betriebssystem (UNIX, Linux, Windows etc.) aufgerufen. Es gibt zwei Betriebsmöglichkeiten: Als eigenständiger Befehlsprozessor oder als Befehl des Betriebssystems.

Typischerweise wird die db2-Commandline als Betriebssystembefehl aufgerufen. Alle Befehle, die DB2 auf diese Weise gegeben werden, werden in der normalen Eingabezeile mit dem Befehl „db2“ voran eingegeben.

Auf diese Weise ist auch das Ausführen von komplexen Scripten möglich, in denen möglicherweise hunderte von Befehlen zum Anlegen einer neuen Datenbank mit ihren Tabellen enthalten sind.

6.4 ORACLE: SQL/Plus und svrmgrl

Die typische Befehlsoberfläche für ORACLE ist SQL/Plus. Auch SQL/Plus ist ein Befehlszeileninterpreter, der zusätzlich zu SQL-Befehlen auch Dateien mit Kommandos an die Datenbank übergeben kann.

Bis Version 8 gab es noch den Servermanager-Linemode (svrmgrl) – Systembefehle wie das Stoppen und Starten des Servers wurden über den Servermanager abgesetzt, Datenbankabfragen über SQL/Plus. Seit Version 8 wuchsen diese Tools mehr und mehr zusammen und ab Version 9 ist der svrmgrl Vergangenheit.

6.5 MySql: der mysql-Befehl und mySqlManager

Mysql ist ein Kommando, dem in der Kommandzeile SQL-Statements übergeben werden können. Auch hier können reichlich Optionen angegeben und auf diese Weise Scripte zum Erzeugen von Datenbanken ausgeführt werden.

Das Management der Datenbank – Einblick in den Online-Katalog, Eingabe von einzelnen SQL-Statements – wird jedoch komfortabler mit dem graphischen Tool mySqlManager durchgeführt

6.6 ODBC & Co

ODBC steht für **O**pen **D**ata**b**ase **C**onnectivity. Es handelt sich dabei um eine von Microsoft erdachte Schnittstelle zur standardisierten Programmierung von Datenbankzugriffen.

Auch ODBC beherrscht SQL – und zwar für alle unterstützten Datenbanken. Sie benötigen „lediglich“ den passenden Treiber (was schonmal zu einer langwierigen Suche werden

kann). ODBC unterstützt sowohl Serverdatenbanken – sowohl für DB2 als auch für ORACLE und mySql sind Treiber im Internet verfügbar. MS-SQL-Server basiert ausschließlich auf ODBC.

Es sind auch für die Dateien der gängigen Desktop-Datenbanken (dBase, Paradox, MS-ACCESS) ODBC-Treiber mit SQL-Unterstützung verfügbar.

6.7 Tabellen anlegen

Vor den Abfragen müssen Sie die Tabellen anlegen und füllen, die sich aus dem besprochenen Datenmodell ergeben.

Viel Tipparbeit haben Sie damit als Schulungsteilnehmer nicht, denn die gesamten SQL-Befehle hierzu sind in einer Datei bereits erfasst. Wie die Datei heisst hängt vom gewählten Datenbanksystem ab:

System	**Dateiname**
MySql	**tablecreate.mysql.ddl**
DB2	**tablecreate.db2.ddl**
ORACLE	**tablecreate.oracle.ddl**

Im zweiten Schritt müssen die Daten in die Schulungsdatenbank geladen werden. Ein Script mit den notwendigen INSERT-Kommandos liegt unter dem Namen

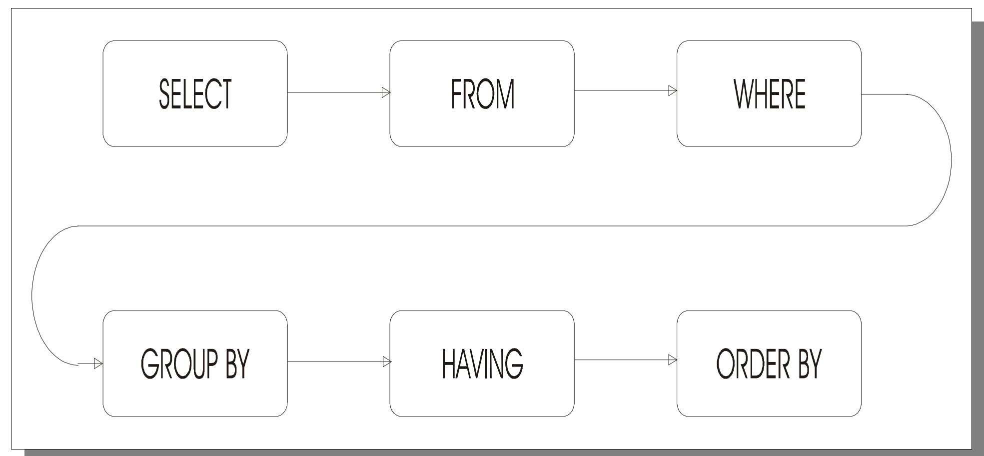
insert.sql

bereit. Hier muss nicht mehr zwischen den einzelnen Systemen unterschieden werden – SQL ist nunmal eine Art Esperanto.

7 Einfache Abfragen

7.1 Unqualifiziertes Select

Das häufigste SQL-Statement dient zum Auswählen von Daten aus der Datenbank. Das englische Wort hierfür heißt SELECT.



Um alle Daten aus der Tabelle CD zu selektieren codieren Sie

```
SELECT * FROM CD ;
```

SQL-Statements, wie vollständige Anweisungen genannt werden, bestehen aus einer oder mehreren Klauseln (englisch: clause). Das SELECT-Statement beginnt immer mit der SELECT-Klausel, die definiert, welche Spalten aus einer Tabelle selektiert werden sollen.

In unserem Fall wird durch das Jokerzeichen angegeben, dass alle Spalten gewünscht sind. Der zweite, zwingende Bestandteil des SELECT-Statements ist die FROM-Klausel, die den Namen der Tabelle(n) angibt, aus der die in der SELECT-Klausel bezeichneten Spalten stammen. Der Name einer Tabelle kann bis zu 18 alphanumerische Zeichen umfassen, von denen das erste ein Buchstabe sein muss.

Die in unserem Hause genutzten Namenskonventionen nach AKD-Richtlinien schreiben jedoch vor, dass nur acht Stellen genutzt werden, die nach einem bestimmten Muster aufzu-

bauen sind. Der Name einer Tabelle ist innerhalb einer Datenbank, d.h. er darf nur einmal benutzt werden.

Wenn Sie die Tabelle ohne Angabe eines Besitzers anlegen, so wird automatisch Ihre UserID als Owner benutzt. Das gleiche gilt für Zugriffe, bei denen dem Tabellennamen kein Owner vorangestellt wird.

Führen Sie nun die gezeigte Abfrage mit SPUFI aus und sehen Sie sich das Ergebnis an.

Wenn Sie sich die Ausgabe der Query ansehen, werden Sie feststellen, dass Sie nicht einen einzelnen Satz, sondern eine wiederum in Tabellenform dargestellte Ergebnismenge erhalten. Bei dieser Abfrage ist die Menge weder nach den CD-Namen noch nach deren Nummern sortiert (wenn sich die Ausgabe an eine der beiden Sortierfolgen hält, so ist das reiner Zufall und liegt u.a. an der Reihenfolge, in der die Sätze eingefügt wurden).

SQL ist eine mengenorientierte Abfragesprache. Später, in Ihren Anwendungsprogrammen, beschreiben Sie auch lediglich eine Ergebnismenge, die Sie dann durch weitere SQL-Anweisungen Element für Element abarbeiten können.

Sehen Sie sich nun analog zur ersten Abfrage die Inhalte der Tabellen MUSIKSTUECK, INTERPR, KOMPONIST und TEXTER an.

7.2 Qualifiziertes SELECT

Die gerade durchgeführten unqualifizierten Abfragen sind typisch für interaktives SQL, das für ad hoc-Abfragen benutzt wird. In Programmen hat ein 'SELECT *' nichts verloren. Codds 9. Regel besagt, dass RDBMS die Möglichkeit schaffen müssen, ein Programm so zu codieren, dass es von Änderungen der Datenstrukturen unberührt bleibt, die seine Funktionen nicht berühren.

Das bedeutet auch, dass in einer Tabelle Spalten hinzugefügt und entfernt werden können, ohne dass Programme, die diese Spalten nicht nutzen, geändert werden müssen. Wenn man ein unqualifiziertes SELECT in einem Programm nutzt, werden immer alle Spalten angesprochen, die zur Laufzeit der Abfrage in der Tabelle vorhanden sind.

Da im Programm aber für jede Spalte eine Variable definiert sein muss, kommt man nicht umhin, ein solches Programm

bei Änderungen der Tabellendefinition anzupassen. Wenn man nur die jeweils benötigten Spalten anspricht, so braucht man lediglich die Programme bzw. Programmteile zu ändern, die von Tabellenänderungen wirklich betroffen sind.

Qualifizierte Zugriffe auf Tabellen codieren Sie, indem Sie anstelle des Jokerzeichens in der SELECT-Klausel den oder die Namen der Spalten angeben, deren Inhalte sie sehen möchten. Wenn Sie mehrere Namen angeben, dann müssen Sie diese durch Kommata trennen.

Lassen Sie sich nur die Spalten CD_Nr und CD_Titel aus der Tabelle CD anzeigen. Sie sind jetzt in der Lage, sich die Inhalte beliebiger DB2-Tabellen, auf die Sie die Zugriffsrechte besitzen, anzeigen zu lassen.

'Virtuelle' Spalten

Die SELECT-Klausel kann nicht nur simple Spaltennamen als Definition für die Ergebnistabelle enthalten, sondern regelrechte mathematische Ausdrücke. Sie können mit allen vier Grundrechenarten und einigen anderen, in der Programmierung bekannten Operatoren Spalten und Konstanten miteinander verknüpfen.

In die Tabelle CD haben wir eine Spalte aufgenommen, die den Kaufpreis der CDs beeinhaltet, damit wir den Überblick über unser in CDs gebundenes Kapital behalten. Ihr Brieffreund aus New York ist nun zu Besuch und staunt über Ihre CD-Sammlung, die mittlerweile einen beträchtlichen Wert hat. Um ihm die Preise der einzelnen CDs zu zeigen, müssten Sie nun für jede Zeile den Preis in DM durch den Dollarkurs (wenn Ihnen der aktuelle Kurs nicht bekannt ist, nehmen Sie 1,60 DM für einen Dollar) dividieren.

Das ist bei Ihrer großen Sammlung ein ziemlicher Aufwand, also lassen Sie doch einfach DB2 die Arbeit machen:

```
SELECT CD_Titel, Preis / 1,60 FROM CD ;
```

Operator **+**	Addition zweier Zahlen		
DB2	**ORACLE**	**MySQL**	**MS-SQL**
Ja	**Ja**	**Ja**	**Ja** Verknüpft auch Strings
Operator **–**	Subtraktion zweier Zahlen		
DB2	**ORACLE**	**MySQL**	**MS-SQL**
Ja	**Ja**	**Ja**	**Ja**
Operator *****	Multiplikation zweier Zahlen		
DB2	**ORACLE**	**MySQL**	**MS-SQL**
Ja	**Ja**	**Ja**	**Ja**
Operator **/**	Division zweier Zahlen		
DB2	**ORACLE**	**MySQL**	**MS-SQL**
Ja	**Ja**	**Ja**	**Ja**
Operator **%**	Modulo-Division zweier Zahlen (ganzzahliges Ergebnis)		
DB2	**ORACLE**	**MySQL**	**MS-SQL**
Nein Funktion mod()	**Nein** Funktion mod()	**Nein** Funktion mod()	**Ja** Auch mod()
Operator **CONCAT**			
DB2	**ORACLE**	**MySQL**	**MS-SQL**
Ja	**Nein**	**Nein** ist hier eine Funktion	**Nein** stattdessen „+“
Operator **||**			
DB2	**ORACLE**	**MySQL**	**MS-SQL**
Ja unter MVS als „!!“	**Ja**	**Nein**	**Nein**

SQL stellt Ihnen verschiedene Operationen zur Verfügung (Siehe Tabelle 3). Diese Operationen variieren jedoch von System zu System zum Teil erheblich – und das trotz verbindlicher ISO-Normen. Die Stringverknüpfungen können bei DB2 auf allen Plattformen mit dem Operator CONCAT erfolgen:

```
SELECT CD_Titel CONCAT Verlag
from CD;
```

IBM ermöglicht auch die Verknüpfung über die Zeichenfolge ||:

```
SELECT CD_Titel || Verlag
from CD;
```

Unter MVS mit nicht-US Zeichensatz stehen diese Zeichen jedoch oft nicht zur Verfügung. Alternativ können bei DB2 auf allen Plattformen daher auch zwei Ausrufungszeichen benutzt werden:

```
SELECT CD_Titel !! Verlag
from CD;
```

Unter MS-SQL wird die Lösung einiger gängiger Programmiersprachen (BASIC, TurboPascal) genutzt und die Verkettung mit dem „+“ der Addition durchgeführt:

```
SELECT CD_Titel + Verlag
from CD;
```

ORACLE wiederum besteht auf der „||“-Variante und kennt den Operator CONCAT nicht, wogegen mySql die Stringverknüpfung nicht als Operator, sondern als Funktion durchführt:

```
SELECT CONCAT(CD_Titel, Verlag)
from CD;
```

Selektieren Sie aus der Tabelle INTERPRET eine Ergebnismenge, in der die Spalten I_NAME und I_FANCLUB aus der Adresse des Fanclubs zusammen in einer Spalte, getrennt durch die Zeichenkette ‘c/o‘, erscheinen.

Sie sind nun in der Lage, mit den Tabellenspalten Berechnungen und ähnliche Operationen durchzuführen, die in die Ergebnismenge einfließen.

8 Funktionen

Eine weitere Methode, Daten in Abfragen zu modifizieren, bieten die Spaltenfunktionen, die SQL anbietet. Hier müssen wir zwischen zwei Kategorien unterscheiden:

- 'einfache' oder 'skalare' Funktionen
- Spalten- bzw. Mengenfunktionen

Einfache oder skalare Funktionen manipulieren die Spalten der Ergebnismenge direkt. Sie dienen beispielsweise zur Stringmanipulation, zur Typumwandlung, zum Herausrechnen von Bestandteilen aus Datums- und Uhrzeitwerten und zum Ersetzen von NULL-VALUES durch Daten.

8.1 Skalare Funktionen

Einige der gebräuchlichsten Funktionen finden Sie in der folgenden Aufstellung. Leider sind die Funktionen von den unterschiedlichen Systemphilosophien abhängig. Daher sollten Sie für den täglichen Einsatz ein entsprechendes Handbuch besitzen (meist in digitaler Form beim DB-System dabei), um die richtigen Funktionen und ihre Syntax nachlesen zu können.

8.1.1 CHR

DB2	ORACLE	mySQL	MS-SQL
Ja	Ja	Nein	Nein

Syntax: CHR(Argument)
Beispiel: CHR(32)

Diese Funktion gibt das Zeichen zurück, das im aktuellen Zeichensatz den Wert der als Parameter übergebenen Zahl besitzt.

CHR(32) wäre beispielsweise im ASCII-Zeichensatz ein Leerzeichen.

Vorsicht: In mySQL und MS-SQL wird diese Funktionalität von der Funktion CHAR erreicht.

8.1.2 CHAR()

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Ja	Ja

DB2

Syntax: CHAR(Argument)
Beispiel: CHAR(PersNr)

Diese Funktion wandelt eine DATE- oder DECIMAL-Spalte in die Zeichenkette um, in der sie üblicherweise dargestellt würde (mit Vornullen). Auf diese Weise können Sie in Ihren Programmen auf einem Umweg auch numerische Spalten als Textketten (STRING) behandeln. In Interaktivem SQL (über SPUFI oder QMF beispielsweise) können Sie so umgeformte Zahlen auch mit Stringoperationen (CONCAT) oder -funktionen (siehe später z.B. unter SUBSTR) bearbeiten.

Syntax: CHAR(Datum, Datumsformat)
Beispiel: CHAR(GEBURTSTAG, JIS)

Dieser Aufruf dient zum Festlegen eines vom eingestellten DEFAULT abweichenden Datumsformats. In den DB2-DEFAULTS kann aus einer Liste von Datumsformaten (ISO, USA, EUR, JIS) eines als Standardwert eingestellt werden. Da Datums- und Zeitspalten mit den Anwendungsprogrammen als Zeichenketten ausgetauscht werden, müssen Programme entweder auf den jeweiligen DEFAULT abgestimmt sein oder aber über die CHAR-Funktion ein Format erzwingen. Das erste übergebene Argument muss keine einzelne Spalte sein, sondern kann auch aus einem gültigen Ausdruck bestehen.

mySQL, MS-SQL

Diese Funktion gibt das Zeichen zurück, das im aktuellen Zeichensatz den Wert der als Parameter übergebenen Zahl besitzt. Sie entspricht daher der Funktion CHR() der anderen beiden Datenbanksysteme

8.1.3 CONCAT

DB2	ORACLE	MySQL	MS-SQL
Nein	Nein	Ja	Nein

Syntax: CONCAT(Argument, Argument...)
Beispiel: CONCAT(Vorname, ' ',Nachname)

Verknüpft zwei oder mehr Zeichenketten. Nur bei mySQL vorhanden, dort anstelle des || oder CONCAT-Operators.

8.1.4 DATE()

DB2	ORACLE	MySQL	MS-SQL
Ja	**Nein** Siehe to_date()	**Nein**	**Nein**

Syntax: DATE(Argument)
Beispiel: DATE('1965-11-30')

DB2

DATE wandelt das Argument in einen gültigen Datumswert um. Dies ist dann interessant, wenn sich in einer Spalte mit CHARACTER-Typ korrekte Datumswerte befinden, die für eine Abfrage in ein 'echtes' Datumsformat gewandelt werden sollen.

Insbesondere, wenn das Datumsformat einer solchen CHARACTER-Spalte von dem als Ausgabe-Default eingestellten Format abweicht (siehe unter CHAR), kann man die Funktion zur Anpassung nutzen. Das oben angeführte Beispiel würde in unserer Installation als Ergebnis die korrekte ISO- Darstellung '30.11.1965' liefern.

Sie können als Argument jedes DB2 bekannte Datumsformat benutzen.

8.1.5 DATEPART()

DB2	ORACLE	MySQL	MS-SQL
Nein Siehe andere Funktionen	**Nein** Siehe to_char()	**Nein** Siehe andere Funktionen	**Ja**

Syntax: DATEPART(datepart, date)
Beispiel: DATEPART(yyyy, Geburtsdatum)

Datepart ermittelt einen Teilwert aus dem Datum. Der Teilwert wird durch den ersten Parameter bestimmt. Folgende Werte können dort stehen:

Datumsteil	Kürzel
Jahr	yy, yyyy
Quartal	qq, q
Monat	mm, m
Tag im Jahr	dy, y
Tag	dd, d
Woche	wk, ww
Wochentag	dw
Stunde	hh
Minute	mi, n
Sekunde	ss, s
Millisekunde	ms

8.1.6 DAY

DB2	ORACLE	MySQL	MS-SQL
Ja	**Nein** Siehe to_char()	**Nein** Siehe dayofmonth()	**Nein** Siehe datepart()

Syntax: DAY(Argument)
Beispiel: DAY(GEBURTSTAG)

Diese Funktion liefert als Ergebnis den Tag aus dem als Argument übergebenen Datum. DAY(DATE('30.11.1965')) würde 30 als Ergebnis liefern.

8.1.7 DAYOFMONTH

DB2	ORACLE	MySQL	MS-SQL
Nein siehe day()	**Nein** Siehe to_char()	**Ja**	**Nein** Siehe datepart()

Syntax: DAYOFMONTH(Argument)
Beispiel: DAYOFMONTH(GEBURTSTAG)

Diese Funktion liefert als Ergebnis den Tag aus dem als Argument übergebenen Datum. DAYOFMONTH(DATE ('30.11.1965')) würde 30 als Ergebnis liefern.

8.1.8 DAYOFYEAR

DB2	ORACLE	MySQL	MS-SQL
Nein	**Nein**	**Ja**	**Nein**

Syntax: DAYOFYEAR(Argument)

Beispiel: DAYOFYEAR(GEBURTSTAG)

Diese Funktion liefert als Ergebnis die laufende Nummer des Tages im Jahr zurück.

8.1.9 DAYS

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: DAYS(Argument)
Beispiel: DAYS(CURRENT DATE) - DAYS(GEBURTSTAG)

DAYS liefert als Ergebnis die Zahl der Tage, die seit dem 1.1. des Jahres 1 unserer Zeitrechnung bis zum angegebenen Datum vergangen sind. Das Beispiel ergibt die Anzahl der Tage an Lebensalter, welche die Person, deren Geburtstag in der gleichnamigen Spalte gespeichert ist, zurückgelegt hat.

8.1.10 DECIMAL

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: DECIMAL(Wert, Länge, Dezimalstellen)
Beispiel: DECIMAL(GEHALT/12,8,2)

Diese Funktion dient zum Erzwingen eines bestimmten Dezimalformats von numerischen Ausdrücken. Der umzuwandelnde Wert kann einen beliebigen numerischen Datentyp besitzen. Als Länge wird die Gesamtzahl von Ziffern in der Rückgabe der Funktion angegeben, wovon eine bestimmte Zahl von Dezimalstellen hinter dem Komma bzw. Dezimalpunkt stehen. Auch diese Funktion kann dazu benutzt werden, unbekannte Datentypen zu ersetzen oder INTEGER-Spalten, die in COBOL als COMPUTATIONAL definiert werden müssten, in PACKED- Darstellung umzuformen.

Bitte beachten Sie, dass bei der Umwandlung von Fließkommawerten einfacher Genauigkeit diese zunächst auf sechs Stellen gekürzt und entsprechend gerundet werden, bevor die Umwandlung in DECIMAL erfolgt. Da die einfache Genauigkeit in DB2 normalerweise acht Stellen umfasst, wird das Ergebnis falsch gerundet. Bei doppelt genauen Fließkommazahlen passiert dies nicht.

Daher muss eine Umwandlung von Zahlen einfacher Genauigkeit auf dem Umweg über die Funktion REAL vorgenommen werden:

DECIMAL(REAL(Argument),Länge, Dezimalstellen)

8.1.11 DIGITS

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: DIGITS(Spalten)
Beispiel: DIGITS(PREIS)

DIGITS gibt als Ergebnis die Stringdarstellung eines dezimalen Wertes zurück (mit Vornullen, je nach Definition des PACKED-Wertes). DIGITS ist nur auf DECIMAL-Werte anzuwenden. Alle andern numerischen Datentypen müssen vorher mit der DECIMAL-Funktion umgewandelt werden. DIGITS entspricht insofern der Funktion CHAR, wie DB2 sie zur Verfügung stellt.

8.1.12 FLOAT

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: FLOAT(Argument)
Beispiel: FLOAT(PREIS)

FLOAT wandelt den übergebenen numerischen Wert in Fließkommazahlen doppelter Genauigkeit um (Exponent-Mantisse-Notation). FLOAT kann z.B. benutzt werden, um die in C unbekannten DECIMAL-Werte (werden als PACKED DECIMAL übergeben) in REAL-Werte umzuformen.

8.1.13 HEX

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: HEX(Argument)
Beispiel: HEX(PREIS)

Bei HEX handelt es sich um die einzige Funktion, die eine Schnittstelle zur physischen Speicherungsmethode darstellt, die DB2 benutzt. HEX wandelt jeden Datentyp in die hexadezimale Darstellung der Bytes dar, die die jeweilige Spalte im Speicher repräsentieren.

8.1.14 HOUR

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Ja	Nein

Syntax: HOUR(Zeitwert)
Beispiel: HOUR(LUPTIME)

HOUR liefert als Ergebnis den Stundenteil des übergebenen TIME- oder TIMESTAMP-Wertes. Die Funktion ist analog zu DAY.

8.1.15 INTEGER

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: INTEGER(Ausdruck)

Beispiel: INTEGER(GEHALT / 2)

Das Ergebnis dieser Funktion ist der Vorkommateil des übergebenen numerischen Wertes. Es wird nicht gerundet.

8.1.16 LENGTH

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Ja	Nein

Syntax: LENGTH(Ausdruck)
Beispiel: LENGTH(TITEL)

LENGTH Stellt die Länge des Arguments fest. Dabei gibt es Unterschiede zwischen den verschiedenen Serversystemen:

DB2 liefert die Anzahl der Bytes zurück, die der Datenwert im Speicher belegen würde. Bei CHAR-Spalten ist das immer die maximale Länge, bei Es handelt sich hierbei nicht um die Länge der Stringdarstellung bzw. die Anzahl von Zeichen in einer Stringvariablen, sondern um die Anzahl von Bytes, die die Spalte im Speicher belegt.

MySQL liefert hingegen die tatsächliche Länge eines STRINGS zurück.

8.1.17 MICROSECOND

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: MICROSECOND(Timestamp-Wert)

Beispiel: MICROSECOND(Zeitpunkt)

Rückgabe des Mikrosekunden-Anteils eines TIMESTAMP-Wertes.

8.1.18 MINUTE

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Ja	Nein

Syntax: MINUTE(Zeit)
Beispiel: MINUTE(CURRENT TIME)

Rückgabe des Minutenteils eines TIME- oder TIMESTAMP-Wertes.

8.1.19 MONTH

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Ja	Nein

Syntax: MONTH(Datumswert)
Beispiel: MONTH(Geburtstag)

MONTH gibt den Monatsanteil eines DATE- oder TI-MESTAMP- Wertes zurück.

8.1.20 RAND

DB2	ORACLE	MySQL	MS-SQL
Nein	Nein	Ja	Nein

Syntax: RAND(Argument)

RAND ist ein Zufallsgenerator. Die Funktion erzeugt eine Fließkommazahl zwischen 0.0 und 1.0. Sie kann mit oder ohne Argument aufgerufen werden; wird sie mit Argument aufgerufen, so wird das Argument als „Saat" (seed) für den Zufallsgenerator genommen. Aufrufe mit derselben Saat ergeben auf demselben System dieselben Reihen von Zufallswerten.
RAND ist eine Funktion, die in der klassischen Datenverarbeitung und insbesondere in Datenbanksystemen nutzlos war.

8.1.21 SECOND

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Ja	Nein

Syntax: SECOND(Zeitwert)

Beispiel: SECOND(Zeitpunkt)

SECOND ermittelt den Sekundenteil eines TIME- oder TIMESTAMP- Wertes.

8.1.22 SUBSTR

DB2	ORACLE	MySQL	MS-SQL
Ja	Ja	Ja	Ja

Syntax: SUBSTR(Textkette, Anfang, Länge)
Beispiel: SUBSTR(CHAR(DATE(Datum),1,2)

SUBSTR gibt den Teil einer Textkette zurück, der an der mit Anfang bezeichneten Stelle beginnt und der Länge lang ist.

8.1.23 TIME

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: TIME(Timestamp-Wert)
Beispiel: TIME(TZeitpunkt)

TIME extrahiert den Zeit-Anteil aus einem TIMESTAMP-Wert

8.1.24 TIMESTAMP

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: TIMESTAMP(Datum, Zeit)
Beispiel: TIMESTAMP(Geburtstag, Geburtszeit)

TIMESTAMP faßt einen Datums- und einen Uhrzeitwert zu einem TIMESTAMP zusammen, wobei als Microsekunden-Anteil 0 vorgegeben wird.

8.1.25 TO_CHAR (ORACLE)

DB2	ORACLE	MySQL	MS-SQL
Nein	Ja	Nein	Nein

Syntax: TO_CHAR(*value*, [*format_mask*], [*nls_language*])
Beispiel: TO_CHAR(1210.73, '9999.9')
Beispiel: TO_CHAR(GebDat, 'yyyy/mm/dd')

Wandelt nicht-Character-Werte in einen STRING um. Als Parameter wird der umzuwandelnde Wert und eine Foratierungsmaske übergeben, in manchen Fällen kann auch die

Angabe einer Landessprache erforderlich sein (z.B. bei Datumsangaben mir ausgeschriebenen Wochentagen oder Monaten).

TO_CHAR erfüllt die Funktionen, die bei DB2 durch CHAR, und DIGITS und HEX geboten werden.

8.1.26 VALUE

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: VALUE(Argument1 ,Argument2 ,Argument3...)
Beispiel: VALUE(LieferDat, '--.--.----')

Beispiel: VALUE(LieferDat, VorLDat, BestellDat, '--.--.----')

VALUE liefert von den als Argumenten übergebenen Werten den ersten zurück, der nicht NULL ist. NULL stellt eine leere Spalte dar, in der sich kein Wert befindet. Wenn die Spalte LieferDat leer ist, wird im ersten Beispiel als Ergebnis '--.--.----' geliefert, was im Programm oder einer Liste eindeutig ist.

Im zweiten Beispiel wird in der Reihenfolge der Spalten entweder das Lieferdatum, das Voraussichtliche Lieferdatum, das Bestelldatum oder, wenn alle Spalten leer sind, die Zeichenkette '--.--.----' zurückgegeben.

8.1.27 VARGRAPHIC

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Nein	Nein

Syntax: VARGRAPHIC(Textkette)
Beispiel: VARGRAPHIC(:TEXTFELD)

VARGRAPHIC ist eine Typkonvertierungsfunktion, welche als Eingabe eine Textkette erwartet und diese in das Format VARGRAPHIC umwandelt, das beispielsweise zum Speichern von japanischen Schriftzeichen (Kanji) benutzt wird.

8.1.28 YEAR

DB2	ORACLE	MySQL	MS-SQL
Ja	Nein	Ja	Nein

Syntax: YEAR(Datumswert)
Beispiel: YEAR(GebDat)

YEAR extrahiert den Jahresanteil aus einem DATE oder TIMESTAMP.

8.2 Spaltenfunktionen

Spaltenfunktionen dienen dazu, die Werte in einer Spalte innerhalb der gesamten Tabelle - meistens vor statistischem Hintergrund - zusammenzufassen. Spaltenfunktionen liefern nicht mehr pro Zeile in der Tabelle ein Ergebnis, sondern pro Abfrage bzw. Abfrageuntergruppe.

In der SELECT-Klausel eines einfachen SELECT-Statements dürfen immer nur die Ergebnisse von Spaltenfunktionen oder skalaren Funktionen bzw. Ausdrücken benutzt werden, da die Spaltenfunktionen nicht pro Element der Ergebnismenge einen Wert liefern können.

8.2.1 SUM

DB2	ORACLE	MySQL	MS-SQL
Ja	Ja	Ja	Ja

Syntax: SUM(Ausdruck)
Beispiel: SUM(Preis)

SUM addiert die Werte aus einer numerischen Spalte oder einem numerischen Ausdruck, der auf jede Zeile der Tabelle angewandt wird. Das Beispiel würde in unserer CD-Datenbank die Summe an Geld ermitteln, die zum Ankauf aller vorhandenen CDs insgesamt aufgewendet wurde.

8.2.2 MIN

DB2	ORACLE	MySQL	MS-SQL
Ja	Ja	Ja	Ja

Syntax: MIN(Ausdruck)
Beispiel: MIN(Preis)

MIN liefert als Ergebnis von allen durch den Ausdruck ermittelten Werten den niedrigsten. In unserem Beispiel wäre das die billigste CD.

8.2.3 MAX

DB2	ORACLE	MySQL	MS-SQL
Ja	Ja	Ja	Ja

Syntax: MAX(Ausdruck)
Beispiel: MAX(Preis)

MAX liefert im Gegensatz zu MIN den höchsten Wert, den der angegebene Ausdruck in der Abfrage ergeben hat - in unserem Beispiel die teuerste CD.

8.2.4 AVG

DB2	ORACLE	MySQL	MS-SQL
Ja	Ja	Ja	Ja

Syntax: AVG(Ausdruck)
Beispiel: AVG(Titelzahl)

AVG errechnet den Mittelwert aus allen bei der Abfrage ermittelten Ergebnissen des Ausdrucks.

8.2.5 COUNT

DB2	ORACLE	MySQL	MS-SQL
Ja	Ja	Ja	Ja

Syntax: COUNT(*)
Beispiel: COUNT(*)

COUNT mit dem Jokerzeichen als Argument zählt alle Zeilen, welche die Ergebnismenge enthält. Da jede Spalte pro Zeile genau einen Wert enthält (NULL-Values werden mitgezählt) ist das Ergebnis spaltenunabhängig. Daher muss als Argument das Jokerzeichen angegeben werden.

8.2.6 COUNT DISTINCT

DB2	ORACLE	MySQL	MS-SQL
Ja	Ja	Ja	Ja

Syntax: COUNT(DISTINCT Spalte)
Beispiel: COUNT(DISTINCT Verlag)

COUNT(DISTINCT...) zählt innerhalb der angegebenen Spalte die unterschiedlichen Werte. Die gezeigte Abfrage

würde die Anzahl unterschiedlicher Verlage ergeben, von denen sich CDs in der Sammlung befinden.

8.3 Abfragen mit Spaltenfunktionen

Skalare Funktionen werden in Abfragen analog zu den mathematischen und Stringoperationen benutzt, darum brauchen wir sie hier nicht noch eingehend einzuüben. Die Spaltenfunktionen hingegen sind in der Handhabung anders. Sie erwarten (mit Ausnahme von COUNT) als Argument einen der Ausdrücke, die in der bisherigen Ergebnismenge die Spalten der Ergebnistabelle gefüllt haben.

Die Ergebnisse, die diese Ausdrücke in den einzelnen Zeilen erzielen, werden dann durch die Funktion zusammengeführt. Die einfachste derartige Abfrage ist die, mit der man die Gesamtzahl aller Zeilen in einer Tabelle zählt:

```
SELECT COUNT(*) FROM CD ;
```

Das Ergebnis ist genau eine Zeile mit einer Spalte, in der die Anzahl der CDs in unserer Sammlung steht. Mit der Abfrage

```
SELECT MIN(Preis) FROM CD ;
```

ermittelt DB2 den niedrigsten Preis für eine CD, der in der Tabelle gespeichert ist. Die Anzahl der Verlage, von denen wir CDs besitzen, kann man mit

```
SELECT COUNT(DISTINCT Verlag) FROM CD ;
```

zählen. Beachten Sie den Unterschied zwischen der ersten Abfrage in diesem Kapitel und dieser hier:

```
COUNT(*) zählt alle Zeilen der Tabelle, COUNT
(DISTINCT...)
```

zählt die unterschiedlichen Inhalte einer Spalte. Den Gesamtwert der CD-Sammlung können Sie mit

```
SELECT SUM(Preis) FROM CD ;
```

errechnen lassen, die durchschnittliche Anzahl von Titeln mit

```
SELECT AVG(Titelzahl) FROM CD ;
```

Wenn Sie eine neue CD in die Tabelle einfügen wollen, müssen Sie eine neue CD-Nummer ermitteln. Sie können sich schon denken, dass die Funktion hierzu MAX heißt. Allerdings müssten Sie das Ergebnis noch um eins hochzählen, um tatsächlich eine neue Nummer zu erhalten. Auch das kann DB2 für Sie tun:

```
SELECT MAX(CD_Nr) + 1 FROM CD ;
```

Sie können die Ergebnisse mehrerer Spaltenfunktionen auch in einer einzigen Abfrage kombinieren. Wie gehabt werden lediglich die einzelnen Ausdrücke in der SELECT-Klausel durch Kommata getrennt aufgeführt. Zum Beispiel:

```
SELECT MAX(Preis), AVG(Preis), MIN(Preis) FROM
CD ;
```

Erstellen Sie nun mit einer einzigen Abfrage eine Statistik über unsere CD-Datenbank, aus der folgende Zahlen hervorgehen: m Anzahl der CDs,

- **Anzahl der Verlage, von denen CDs vorhanden sind,**
- **Höchster, niedrigster und durchschnittlicher Preis,**
- **höchste, niedrigste und durchschnittliche Titelzahl pro CD**
- **Gesamtwert der CD-Sammlung auf Basis der Kaufpreise, umgerechnet in US-Dollar**

Ermitteln Sie weiterhin

- **die Anzahl der verschiedenen Komponisten und Interpreten, denen CDs gewidmet sind**

Sie sind nun in der Lage, DB2-Tabellen anhand der in SQL implemetierten statistischen Spaltenfunktionen komplex auszuwerten.

9 Einschränken der Ergebnismenge

Bisher haben Sie immer die gesamte Tabelle bearbeitet. Ein DBMS wäre aber schon im Ansatz unbrauchbar, wenn Sie nicht in der Abfragesprache Suchbedingungen angeben könnten.

In SQL ist die Suchbedingung so realisiert, dass Sie durch logische Bedingungen die Menge, die der Ergebnistabelle zugrunde liegt, einschränken. Codiert wird die Suchbedingungen in der WHERE-Klausel.

Die WHERE-Klausel folgt immer direkt der FROM-Klausel und ist der erste nicht zwingend erforderliche Bestandteil eines SELECT-Statements. Sie beschreibt die Elemente, die der Ergebnismenge angehören sollen. Die Ergebnismenge kann, je nach Bedingung, aus keinem, einem oder vielen Elementen bestehen. Möglicherweise schränkt die WHERE-Klausel die Menge auch gar nicht ein.
Einfache Vergleiche

Innerhalb der WHERE-Klausel stehen die üblichen Vergleichsoperatoren zur Verfügung, die mit Klammerebenen und den Operatoren AND bzw. OR verknüpft werden können. Verneinen lässt sich eine Bedingungen durch den vorangestellten Operator NOT.

Operator	**Beschreibung**
=	ist gleich
<	größer als
<=, =<	größer oder gleich
>	kleiner als
>=, =>	kleiner oder gleich
^=, <>	nicht gleich
LIKE	Ähnlichkeit von Textketten
IN ...	ist Element der Menge
BETWEEN ... AND ...	liegt zwischen ... und ...

Neben den üblichen Vergleichen besitzt SQL noch Operatoren, die Textketten auf Ähnlichkeit überprüfen, und mengenorientierte Vergleiche.

Bezogen auf unsere CD-Datenbank könnte eine Abfrage mit WHERE- Klausel etwa so aussehen:

```
SELECT CD_Titel, Preis FROM CD WHERE Preis <
10.00 ;
```

Mit dieser Abfrage ermitteln Sie alle CDs mit einem Preis von weniger als 10.00 DM. Angenommen, Sie suchen nach einer CD, deren Titel Ihnen nur bruchstückhaft bekannt ist, so können Sie mit der Suchbedingung LIKE auch nach Teilen von Textketten suchen.

Den Suchbegriff können sie einem Dateinamen nicht unähnlich maskieren. Das Prozentzeichen (%) steht dabei für keines, eines oder viele Zeichen, der Unterstrich (_) für genau ein Zeichen. Auf der Suche nach der CD mit dem Wort 'Lounge' im Titel könnten Sie folgende Abfrage durchführen:

```
SELECT CD_Nr, CD_Titel FROM CD WHERE CD_Titel
LIKE '%Lounge%' ;
```

Eine mengenorientierte Vergleichsmethode steht mit dem Operator IN zur Verfügung. Sie suchen beispielsweise alle CDs, die den Kategorien Jazz und Blues angehören. Es gibt hier zwei Möglichkeiten, dies zu formulieren:

```
SELECT CD_Titel, Musikart FROM CD WHERE
Musikart = 'Jazz' OR Musikart = 'Blues' ;
```

oder

```
SELECT CD_Titel, Musikart FROM CD WHERE
Musikart IN ('Jazz', 'Blues') ;
```

Die zweite Formulierung ist die übersichtlichste, die allerdings auch von den Vergleichsmöglichkeiten in 'klassischen' Programmiersprachen abweicht. Wie später in dieser Dokumentation noch erläutert wird, ist sie aus Performancegründen in Programmen manchmal vorteilhafter. Ähnlich sieht es mit der BETWEEN-Konstruktion aus.

Um alle Musikstücke in der Sammlung zu ermitteln, die zwischen 1980 und 1989 erschienen sind, gibt es zwei mögliche Abfragen:

```
SELECT Musiktitel, Erschein_Datum FROM
MUSIKSTUECK WHERE YEAR(Erschein_Datum) > 1980
AND YEAR(Erschein_Datum) < 1989 ;
```

und

```
SELECT Musiktitel, Erschein_Datum FROM
MUSIKSTUECK WHERE YEAR(Erschein_Datum) BETWEEN
1980 AND 1989 ;
```

Auch hier ist in einem Programm der 'moderneren' Abfrage mit BETWEEN der Vorzug zu geben.

Erstellen Sie nun Listen über unsere CD-Sammlung, die folgende Informationen enthalten:

- **Alle CD-Titel der Musikart 'Klassik'**
- **Alle CD-Titel mit mehr als 10 Titeln**
- **Alle CD-Titel von dem Verlag, der 'WEA' enthält**
- **Alle Musikstücke, die vor 1980 erschienen und länger als drei Minuten sind (Vorsicht: Die Länge wurde in der Tabelle in Sekunden definiert!)**

Sie sind nun in der Lage, die Ergebnismenge anhand der Attribute ihrer Elemente einzuschränken.

9.1 Vergleiche mit Funktionen und 'virtuellen' Spalten

In der WHERE-Bedingung können Sie auch errechnete Werte und Funktionen nutzen.

9.1.1 Skalare Funktionen

Natürlich können Sie auch die besprochenen skalaren Funktionen in der WHERE-Klausel einsetzen. Beispielsweise können Sie Datumswerte 'aufteilen': Wenn alle Interpreten gesucht werden, die im Dezember Geburtstag haben, steht Ihnen die Abfrage

```
SELECT I_Name, I_GebDat FROM INTERPRET WHERE
MONTH(I_GebDat) = 12 ;
```

zur Verfügung. Ähnlich kann man über Typkonvertierungen auch numerische und alphanumerische Spalten miteinander vergleichen, mit SUBSTR einen Vergleich auf einen bestimmten Teil einer Zeichenkette durchführen etc.
Spaltenfunktionen

Spaltenfunktionen sind in Verbindung mit der WHERE-Klausel in zweierlei Hinsicht betroffen. Sie können sie - wie gehabt - in der SELECT-Klausel benutzen, um ein Ergebnis zu erzeugen, das nur aus einer Statistik über die Spalten besteht.

Allerdings liegen den Ergebnissen nicht mehr die Werte der gesamten Tabelle zugrunde, sondern nur noch die aus der Ergebnismenge, welche die WHERE-Klausel beschreibt:

```
SELECT COUNT(*), COUNT(DISTINCT Musikart) FROM
CD WHERE Preis < 20.00 ;
```

Daher kann man die Spaltenfunktionen auch in der WHERE-Klausel nicht benutzen. Eine Abfrage auf alle CDs, die weniger als der Durchschnitt kosten, würde alle überdurchschnittlich teuren CDs ausfiltern. Die Funktion AVG() errechnet ihr Ergebnis aber auf Basis der Zeilen, die von der WHERE-Klausel zugelassen werden.

Folglich würde eine Abfrage "WHERE Preis < AVG(Preis)" mit jeder nicht zugelassenen Zeile das Ergebnis von AVG (Preis) verändern. Derartige Abfragen sind daher von der Sprachlogik her nicht zulässig. Manche SQL-Server lassen sie zu und behelfen sich damit, dass für AVG(Preis) der Durchschnitt aller Zeilen benutzt wird.

Dies entspricht aber nicht der ISO-Normierung, da es zu konzeptionellen Problemen in verknüpften WHERE-Klauseln führt Es ist nicht möglich, Spaltenunktionen in der WHERE-Klausel zu benutzen.
SQL-Variable

SQL stellt Ihnen für Abfragen nicht nur Funktionen, Spalten und Operatoren zur Verfügung, die sie benutzen können, sondern auch SQL- Variable. Die bei DB2 und MySQL vorhandenen Variablen finden Sie in Tabelle 5. Diese sind nur innerhalb von SQL-Statements verfügbar, dort aber weitgehend uneingeschränkt. Sie können Variable auch innerhalb der SELECT-Klausel benutzen, um Spaltenwerte durch Formeln zu modifizieren oder zu ersetzen. Allerdings ist diese Verwendung weitaus seltener, als die Nutzung in WHERE-Klauseln.

Tabelle 5: Special Register bei DB2 und MySQL

Name	**Bedeutung**
CURRENT TIME	Aktuelle Uhrzeit
CURRENT DATE	Tagesdatum
CURRENT TIMESTAMP	Zeitmarke aus Datum und Uhrzeit
USER	Autorisierungs-ID des Users
SQLID	aktueller Default-Table-Owner

In den bisher genutzten Klauseln und Statements ist CURRENT TIME noch nicht so sehr von Nutzen. Gleiches gilt für SQLID. Mit den Variablen CURRENT DATE und CURRENT TIMESTAMP kann man aber beispielsweise Fristen überwachen.

Zahlungsfristen beispielsweise, die auf Ablauf zu kontrollieren sind, kann man mit der Bedingung ...WHERE Faelligkeit < CURRENT DATE... überprüfen. Bei einer solchen Abfrage ist es unnötig, vor dem SQL-Statement das aktuelle Systemdatum zu ermitteln und aufzubereiten.

Man kann hiermit sogar Queries in QMF oder einem anderen Utility vorbereiten, die immer vom Tagesdatum abhängige Ergebnisse liefern. Datums- und Uhrzeitwerte werden DB-intern als codierte numerische Werte abgespeichert.

Wenn Sie in einem komplexeren Verwaltungssystem einem Anwender genau die Vorgänge zur Auswahl geben wollen, die er selbst bearbeitet hat, dann können Sie mit einer Bedingung wie ...WHERE SachBearID = USER... jedem Anwender genau seine Daten anzeigen.

9.1.1.1 Kleine Scherzartikelkunde für DB2-Anwender

Wenn man bei DB2 Grundrechenarten auf Datumswerte anwendet (Formeln wie "CURRENT DATE - Geburtstag") erhält man nicht, wie zu erwarten wäre, die Anzahl der Jahre, Monate und Tage als Datum gruppiert zurück, sondern Datenmüll. DB2 wendet die Rechenarten unabhängig von der benutzten Codierung des Datums auf die Spalten an.

Datumsangaben werden im Format Packed Decimal bzw. BCD (Binary Coded Decimal) abgespeichert. Dazu werden in jedem Byte zwei Ziffern gespeichert, je vier Bit stellen eine Ziffer dar. Lässt man sich die Bytes hexadezimal anzeigen, wird das Datum „12.02.2004“ als „20 04 02 12“ dargestellt – vier Bytes, in denen Jahr, Monat und Tag gespeichert sind.

Das macht natürlich die chronologische Sortierung des Datums einfach. Andererseits hat DB2/MVS seit den ersten Versionen ein besonderes „Feature“: Die Formel „CURRENT DATE – Geburtstag“ würde mathematisch und nicht kalndarisch berechnet. Wäre heute der 12.2.2004 und ds Geburtsfdatum der 18.07.1968 würde DB2 folgende Rechnung durchführen:

20040212 – 19680718 = 359494.

Weil das Ergebnis keinesfalls als Datum gültig ist wird es als Zahlenwert ausgegeben.

Was glauben Sie, wieviele Anwenderinnen und Anwender diese Zahl für die Anzahl der Tage zwischen den Daten gehalten und weiterverwendet haben?

Die Datumsarithmetik aus dem Beispiel im oberen Absatz muss wie folgt codiert werden:

```
CURRENT DATE - DAY(Geburtstag) DAYS - MONTH
(Geburtstag) MONTH - YEAR(Geburtstag) YEARS
```

Ermitteln Sie aus den vorhandenen Daten die Anzahl der Interpreten, die heute Geburtstag haben. Sagen Sie voraus, welches Ergebnis die folgende SQL-Abfrage erzielen wird:

```
SELECT DAYS(CURRENT DATE - K_GebDat), K_Name
FROM KOMPONIST WHERE K_Nr < 24 AND K_Todestag
^= CURRENT DATE AND CURRENT TIME < '12:00:00' ;
```

Sie sind jetzt in der Lage, SQL-Variable in Abfragen einzusetzen.

10 Sortieren

Sicherlich ist Ihnen schon aufgefallen, dass in den bisherigen Ergebnismengen die Zeilen (=Daten) unsortiert angezeigt wurden; sofern sich eine (scheinbare) Sortierfolge ergeben hat, lag das an der eher zufälligen Reihenfolge, in der die Daten eingefügt wurden. DB2 liefert die Daten in der Reihenfolge, in der sie zufällig am besten im Zugriff lagen.

Dies kann - je nachdem - die Reihenfolge sein, in der sie in die Tabelle eingefügt wurden oder auch die Sortierfolge des Index', den DB2 im Zusammenhang mit einer WHERE-Klausel benutzt hat. Wenn Sie die Ergebnismenge in irgendeiner Form zwingend sortiert haben müssen, stellt SQL Ihnen die ORDER-BY-Klausel zur Verfügung, mit der Sie sehr komplexe Sortierfolgen vorgeben können.

Es steht Ihnen frei, nach Spalten der zu Grunde liegenden Tabelle zu sortieren oder nach einer in der SELECT-Klausel definierten Spalte der Ergebnistabelle - jeweils mit- und untereinander kombiniert und wahlweise absteigend oder aufsteigend. Die Syntax ist wiedereinmal furchtbar einfach.

10.1 Sortierfolge nach 'Original-Spalten'

Die Spalten der Tabelle, die dem SELECT-Statement zu Grunde liegt, sollten Sie ungeachtet der Möglichkeit, die im nächsten Abschnitt erwähnt wird, immer namentlich angeben, um den Optimizer, der den performancegünstigsten Zugriffspfad zu den gewünschten Daten ermitteln soll, nicht unnötig zu verwirren. Um alle Interpreten sortiert nach ihrem Namen als Ergebnis zu erhalten codieren Sie

```
SELECT I_Name, I_GebDat FROM INTERPRET ORDER BY
I_Name ;
```

Sie erhalten die Namen in der korrekten alphanumerischen Sortierfolge. Beachten sollte man, dass Groß- und Kleinbuchstaben unterschieden werden und die Zeilen daher nach dem ASCII- bzw. EBCDIC-Code der Zeichen sortiert sind. Auch Umlaute liegen - je nach verwendetem Zeichensatz - teilweise sehr neben der eigentlich zu erwartenden Sortierfolge. Immer korrekt werden dagegen Datums- und Uhrzeitwerte sortiert.

Obwohl auch sie einem Programm gegenüber wie STRING-Werte erscheinen werden sie numerisch codiert in der Datenbank abgelegt. Daher können sie auch mathematisch korrekt in der ORDER BY- Klausel verarbeitet werden. Sie können eine Liste der Interpreten, sortiert nach dem Geburtsdatum, auf folgende Weise erstellen lassen:

```
SELECT I_Name, I_GebDat FROM INTERPRET ORDER BY
I_GebDat ;
```

Die Liste, die Sie erhalten, wird mit dem ältesten der gespeicherten Interpreten beginnen. Um die Sortier-Richtung zu variieren dienen die Optionen ASC und DESC (für ascending = aufsteigend und descending = Absteigend). Wird keine Option angegeben, so unterstellt DB2, dass die aufsteigende Sortierung gewünscht ist. Eine Liste aller Interpreten, die mit der jüngsten gespeicherten Person beginnt, erhalten Sie mit dieser Abfrage:

```
SELECT I_Name, I_GebDat FROM INTERPRET ORDER BY
I_GebDat DESC ;
```

Selbsterständlich können Sie nicht nur nach einer, sondern auch nach mehreren Spalten sortieren lassen. Verknüpft werden diese durch Kommata. Angenommen, Sie benötigen eine Liste aller gespeicherten Musiktitel, so kann es passieren, dass einzelne Titel doppelt vorhanden sind, weil mehrere Komponisten identische Namen für ihre Stücke ausgewählt haben. Die gleichnamigen Titel können Sie mit folgender Abfrage noch nach dem Erscheinungsjahr sortieren:

```
SELECT Musiktitel, Erschein_Datum FROM
MUSIKSTUECK ORDER BY Musiktitel ASC,
Erschein_Datum ASC ;
```

Da komplexere Sortierfolgen häufig Kombinationen aus auf- und absteigenden Sortierungen enthalten, habe ich mir (wie Sie sehen können) angewöhnt, bei kombinierten ORDER BY-Klauseln auch den Default ASC ausdrücklich anzugeben.
Sortieren nach Ergebnisspalten

Häufig ist es erforderlich, die Sortierfolge nicht von einer Spalte der Originaltabelle, sondern von einer Spalte aus der Ergebnismenge abzuleiten. Ein häufiges Beispiel ist die Sortierung nach Teilen aus Datumsspalten, um Kalenderfunktionen zu ermöglichen. Wenn Sie einen Jahreskalender mit den Geburtstagen der gespeicherten Interpreten er-

stellen wollen, hilft Ihnen das Sortieren nach dem Datum nicht weiter.

Wie Sie gesehen haben wird ein Datumswert zuerst nach dem Jahr, dann erst nach Monat und Tag sortiert. Die oben erstellte Abfrage mit ORDER BY I_GebDat ist also ungeeignet. Es wäre vielmehr erforderlich, eine Sortierfolge aus Monats- und Tagesteil des Datums selbst zu definieren. Sie können aber direkt keine Variablen oder Funktionen in der ORDER BY-Klausel benutzen - ORDER BY MONTH(I_GebDat) sollte bei normgerechten SQL-Servern zu einer Fehlermeldung führen.

Allerdings können Sie sich mit einem Umweg behelfen. Üblicherweise ist es zwingend erforderlich, dass die in der ORDER BY- Klausel benutzten Spalten auch in der SELECT-Klausel enthalten sind. Hätten Sie in der oben genannten Abfrage der Interpreten, sortiert nach Geburtsdatum, die entsprechende Spalte mit der Sortierfolge nicht in der SELECT-Klausel definiert, so hätten Sie einen Fehler erhalten. Auf der anderen Seite können Sie jede Spalte der Ergebnismenge in die ORDER BY-Klausel einsetzen, indem Sie die (numerische) Spaltenposition angeben.

Der gewünschte Kalender aller Geburtstage von Interpreten im laufenden Jahr würde also wie folgt aussehen:

```
SELECT I_Name, DAY(I_GebDat), MONTH(I_GebDat)
FROM INTERPRETEN ORDER BY 3 ASC, 2 ASC ;
```

Wie Sie sehen wird die Ergebnismenge zuerst nach ihrer eigenen dritten Spalte, dann nach der zweiten sortiert. Diese Spalten können nun alle Werte (incl. den effektlosen Konstanten und SQL-Variablen) enthalten. Zunächst ist die Definition der Sortieroptionen unabhängig von der Reihenfolge der Spalten in der Ergebnistabelle.

Wenn Sie die Abfrage verändern und sich dadurch die Position einer Spalte ändert, so beziehen sich numerische ORDER BY-Referenzen durch die erfolgte Verschiebung auf andere Spalten als vorher. Sie müssten daher die Sortieranweisung nach jeder Änderung überarbeiten. Es ist empfehlenswert, wo irgend möglich in der ORDER BY-Klausel den Namen der Spalte anzugeben. Aber auch für den SQL-Server sind Statements 'pflegeleichter', die sich, wenn möglich, über den Namen auf Spalten beziehen.

11 Gruppieren

Die statistischen Funktionen, die Sie in der SELECT-Klausel benutzen können, sind Ihnen ja bereits bekannt. Jedoch können Sie diese bislang nur auf die Werte der gesamten Ergebnistabelle beziehen.

Wenn Sie die Werte bezogen auf bestimmte, anhand von Spaltenwerten definierbare Untergruppen der Datenmenge beziehen wollen, müssen Sie (bislang) für jede Untergruppe ein eigenes SELECT-Statement erstellen und mit der WHERE-Klausel jedesmal die entsprechende Bedingung definieren, die die Gruppe beschreibt.

Das macht den Verwaltungsaufwand für den Programmierer natürlich wesentlich größer, insbesondere bei Programmänderungen. Nur sehr schwer zu realisieren ist auf diesem Wege eine Abfrage über alle im Datenbestand tatsächlich vorkommenden Gruppen, wenn nicht bekannt ist, welche Gruppen es überhaupt geben kann. Hier ein Beispiel, das wir in diesem Abschnitt lösen wollen:

Sie möchten, gruppiert nach Musikrichtungen, die jeweilige Anzahl von CDs, die Gesamtzahl der Titel, den höchsten und den durchschnittlichen Preis einer CD, sowie den Wert aller CDs einer Gruppe anhand des Kaufpreises ermitteln. Mit den Ihnen bislang bekannten Sprachelementen müssten Sie zunächst in einer Abfrage die Musikarten ermitteln.

Ein simples SELECT Musikart FROM... würde nicht pro Musikart, sondern pro CD eine Zeile ergeben. Diese müssten Sie mit ORDER BY nach der Musikart sortieren, damit alle gleichnamigen Werte auch nacheinander erscheinen. Im Programm müssten Sie nun die einzelnen Zeilen lesen und durch Vergleiche der Werte feststellen, wann eine neue Musikrichtung in der Ergebnismenge erscheint. Für jede Musikrichtung müssten Sie daraufhin eine Abfrage durchführen, bei der Sie in der SELECT-Klausel die entsprechenden Funktionen angeben und in der WHERE-Klausel die Datenmenge auf die gerade ermittelte Musikrichtung einschränken.

Ein relativ mühsames und fehleranfälliges Unterfangen.

11.1 DISTINCT

Wesentlich verkürzen können Sie die Verarbeitung mit dem Schlüsselwort DISTINCT. Analog zur Anwendung in der COUNT- Funktion können Sie DISTINCT in der SELECT-Klausel einem Spaltennamen voranstellen, um nicht für jede Zeile der Tabelle ein Ergebnis zu erzielen, sondern für jede unterschiedliche Ausprägung der Spalte. Die Anweisung

```
SELECT DISTINCT Musikart FROM CD ;
```

würde eine Liste aller unterschiedlichen Musikarten ausgeben. Ohne Distinct bekämen Sie pro CD eine Zeile Ergebnis. Da die Anzahl der Spalten der Ergebnismenge ja reduziert wird können die unterschiedlichen Werte in den anderen Spalten, die in der SELECT-Klausel angegeben werden können, nicht mehr eindeutigen Musikarten zugeordnet werden.

Daher bezieht sich DISTINCT immer auf alle Spalten, die danach angegeben sind, jeweils in Kombination. Eine Abfrage wie SELECT DISTINCT Musikart, Interpret FROM CD ; ergäbe eine Liste aller Interpreten und der Musikarten, die sie gespielt haben - aber jeweils nur eine Zeile je Kombination, unabhängig von der Zahl der jeweils gespeicherten CDs. DISTINCT muss auch immer dem Schlüsselwort SELECT folgen, es dürfen also keine Spalten oder Funktionen vorher angegeben werden. Nichtsdestotrotz wäre diese Anweisung schon eine erhebliche Erleichterung für unsere Abfrage, da Sie eine aufbereitete Liste aller vorhandenen Musikarten erhalten.
GROUP BY

Mittels der GROUP BY-Klausel können Sie unser Problem in einer einzigen SELECT-Klausel lösen. GROUP BY gruppiert die Zeilen der Ergebnismenge nach spezifizierten Spalten und arbeitet insofern ähnlich SELECT DISTINCT. Jedoch sind in der SELECT-Klausel wieder Spaltenfunktionen möglich, die Sie bei SELECT DISTINCT gar nicht codieren könnten.

Zwei Einschränkungen müssen, wegen der Zusammenfassung mehrerer Datensätze zu einer Zeile der Ergebnismenge, beachtet werden:

⇩ Die in GROUP BY zum Gruppieren angegebenen Spalten müssen zwingend wenigstens einmal in ihrer 'Grundform'

in der SELECT- Klausel vorkommen, damit man die Zeilen der Ergebnismenge der Gruppe, zu der sie gehören, zuordnen kann.

⇩ Außer den Spalten der GROUP BY-Klausel dürfen andere Spalten nur mit Spaltenfunktionen (SUM, COUNT, AVG etc.) benutzt werden, da sich unterschiedliche Wertausprägungen innerhalb einer Gruppe nur auf statistischer Basis darstellen lassen.

Die Lösung unseres Eingangs beschriebenen Problems sähe mit GROUP BY wie folgt aus:

```
SELECT Musikart, COUNT(*), SUM(Titelzahl), MAX
(Preis), AVG(Preis), SUM(Preis) FROM CD GROUP
BY Musikart ;
```

Die Auflösung pro Interpret und Musikart wäre das Ergebnis der folgenden Abfrage:

```
SELECT Interpret, Musikart, COUNT(*), SUM
(Titelzahl), MAX(Preis), AVG(Preis), SUM(Preis)
FROM CD GROUP BY Interpret, Musikart ;
```

Sie erhalten pro vorhandener Kombination der Ausprägungen der GROUP BY-Spalten eine Zeile Ergebnis.

11.2 Auswählen mit HAVING

Oft ist es nicht erforderlich, dass alle Gruppen in der Ergebnismenge erscheinen. In manchen Fällen will man nur die Gruppen in seinem Programm verarbeiten, deren ermittelten Daten (also in unserem Fall beispielsweise die Anzahl der CDs in der Gruppe) bestimmten Kriterien entsprechen. In diesem Fall könnte man sich damit behelfen, dass das Programm die 'ungewünschten' Zeilen der Ergebnistabelle überspringt.

Denkbar wäre beispielsweise, dass Sie nur diejenigen Gruppen sehen wollen, die aus 10 oder mehr CDs bestehen. Die o.g. programmierte Lösung wäre hier gangbar. Werfen Sie bitte einen Blick auf Abbildung 7. Die dort angegebene Reihenfolge der Klauseln im SELECT- Statement ist zwingend und macht schon von der Reihenfolge in der Codierung her klar, in welcher Reihenfolge sie in der Abfrage abgearbeitet werden. Die WHERE-Klausel erscheint vor dem GROUP BY.

Es werden also nur die Zeilen der ursprünglichen Tabelle von GROUP BY bearbeitet, die den Bedingungen der WHERE-Klausel entsprechen. Zeilen, die hier 'herausgefiltert' werden, fließen nicht in die Gruppen ein. Folglich ist das Herausfiltern aller Gruppen, die aus 10 oder mehr Elementen bestehen, mit WHERE nicht möglich; ein Vergleich auf "COUNT(*) > 10" wäre nicht nur aus prinzipiellen Gesichtspunkten ungültig, sondern auch noch mehrdeutig. Nutzbar ist aber die HAVING-Klausel.

Wie Sie sehen ist dieser Sprachbestandteil unmittelbar hinter der GROUP BY-Klausel platziert und bezieht sich folglich auf die Gruppen, die durch GROUP BY ermittelt wurden. HAVING macht es möglich, nur die Gruppen in die Ergebnistabelle aufzunehmen, die bestimmten Kriterien entsprechen. Diese Kriterien werden genauso codiert, wie die in der WHERE- Klausel. Sie können Vergleiche mit Spalten anstellen, aber in diesem Fall nur mit Spalten, die auch in der GROUP BY-Klausel enthalten sind.

Weiterhin können Sie - im Gegensatz zu WHERE - in HAVING auf Mengenfunktionen über Spalten abfragen. Die Mengenfunktionen werden bei Gruppierung nicht pro Tabelle, sondern pro Gruppe ausgeführt. Also ließe sich mit HAVING COUNT(*) > 9 die Ergebnismenge auf alle Spalten von 10 oder mehr Zeilen einschränken. Eine Sortieranweisung mittels ORDER BY bezieht sich schließlich auf die Zeilen, die GROUP BY erstellt und HAVING nicht ausschließt. Die erwähnte Fragestellung ließe sich wie folgt codieren:

```
SELECT Verlag, COUNT(*) FROM CD GROUP BY Verlag
HAVING COUNT(*) > 9 ;
```

Ermitteln Sie mittels SQL folgende Daten:

⇩ **Die Namen der Verlage inklusive der Anzahl der CDs sowie der billigsten und teuersten CD pro Verlag. m Die Instrumente, die von Interpreten gespielt werden, inklusive der Anzahl der Interpreten pro Instrument und den Geburtsdaten des ältesten und des jüngsten Interpreten für alle Instrumente außer dem Pseudoinstrument 'GESANG'.**

⇩ **Die Verlage inklusive der Anzahl aller CDs vom jeweiligen Verlag, von denen wenigstens 2 CDs gespeichert sind.**

Sie Sind nun in der Lage, komplexe Abfragen über isolierte Tabellen durchzuführen und beherrschen alle hierfür notwendigen Sprachelemente.

12 Verbundene Abfragen

Bislang können sie nur einzelne, vom Rest der Datenbank isolierte Tabellen abfragen, wie es auch in ISAM-Dateien möglich ist. Insofern hat SQL noch nichts 'Relationales' an sich; es ist lediglich eine äußerst mächtige Abfragesprache, die dem Programm die in anderen Sprachen notwendige Aufbereitung und weitergehende Auswahl der Daten abnimmt.
SQL wäre aber keine relationale Abfragesprache, wenn es nicht auch eine Möglichkeit gäbe, mehrere Tabellen miteinander zu verknüpfen. Hierfür gibt es mehrere Methoden.

12.1 Tabellen mit Join verknüpfen

Die einfachste und häufigste Methode, eine Abfrage auf mehrere Tabellen zu beziehen, ist der Join. Join ist kein Schlüsselwort, sondern die aus dem Englischen übernommene Bezeichnung für eine bestimmte Abfrage zweier zusammengeführter Tabellen.

Mittels Join können Sie zwei Tabellen zu einer neuen (Ergebnis-) Tabelle vereinen. Sie sind dabei nicht auf die im Daten(bank)design definierten Beziehungen zwischen den Tabellen angewiesen, sondern können beliebige Tabellen selbst aus völlig unterschiedlichen und voneinander unabhängigen Programmen miteinander verbinden. Die einzige Bedingung hierfür ist, daß sich alle Tabellen im selben DB2-Subsystem (bei DB2/MVS) bzw. in derselben DATABASE (bei DB2 UDB, ORACLE, MySQL und MS-SQL).

12.2 Mengen zusammenführen

Vielleicht kennen Sie aus der Mengenlehre verschiedene Methoden, zwei Mengen miteinander zu vereinigen. Um Datenmengen (und um die handelt es sich bei relationalen Datenbanken) unterschiedlichen Typs miteinander zu verbinden benötigen Sie die Methode, die man auch als 'Kreuzprodukt' bezeichnet.

Sie können Abbildung 8 eine schematische Darstellung entnehmen, wie man aus zwei Mengen ein Kreuzprodukt erstellt. Beide Mengen (A und B) bestehen aus einer Vielzahl von Elementen, die innerhalb einer Menge genau einen Typ haben: a oder b. Aber jedes Element ist für sich von den anderen zu unterscheiden, was durch die unterschiedlichen Nummern deutlich gemacht wird. Ebenso sind die Zeilen

einer Tabelle alle gleichartig, aber anhand der enthaltenen Daten voneinander unterscheiden. Würde man diese Elemente (bzw. Zeilen) durch Summieren einer neuen Ergebnismenge (oder -tabelle) zuordnen, enthielte diese Menge Elemente unterschiedlicher Art (a und b).

Das X als mathematisches Symbol für das Kreuzprodukt deutet schon an, daß es sich weniger um eine Art von Addition als vielmehr um eine Multiplikation handelt: Jedes Element von Menge A (a1, a2, a3, a4) wird mit jedem Element aus Menge B (b1, b2, b3, b4, b5) zu jeweils einem neuen Element an,bn verbunden. Die Anzahl der Elemente in der Ergebnismenge ist demnach so groß, wie die Zahl der Elemente in Menge A multipliziert mit der Zahl der Elemente in Menge B.

Wenden wir das einmal auf unsere CD-Datenbank an. Eine mögliche Verbindung zwischen zwei Tabellen ist die zwischen KOMPONIST und MUSIKSTUECK: zum Musikstück ist nicht der Name, sondern die Nummer des Komponisten gespeichert. Wollte man eine Liste aller Musikstücke mit dem Namen ihrer Komponisten erstellen, so müßte man mit einem Programm zu jedem Musikstück den Namen des Komponisten durch ein eigenes SELECT ermitteln.

Codieren wir zunächst einen JOIN zwischen den Tabellen und untersuchen das Ergebnis:

```
SELECT *
FROM MUSIKSTUECK,
     KOMPONIST
;
```

Wir wählen SELECT *, um alle Spalten angezeigt zu bekommen. Wie Sie feststellen können, enthält die Ergebnistabelle alle Spalten aus beiden Tabellen. Insofern ist also tatsächlich eine neue Tabelle als Menge von neuen Datenelementen entstanden. Betrachten wir nun die Inhalte der Zeilen. Aus der Tabelle MUSIKSTUECK stammt die Spalte KOMPONIST, die die Schlüsselnummer des jeweiligen Komponisten enthält. Ihr Pedant in der Tabelle KOMPONIST heißt K_NR. Vergleichen Sie nun in den Zeilen der Ergebnismenge die Werte von KOMPONIST und K_NR.

Wie Sie sehen ergeben die meisten der ausgegebenen Zeilen keinen sonderlichen Sinn, denn es wurde jede Zeile von MUSIKSTUECKE mit jeder Zeile von KOMPONISTEN verbunden - auch, wenn der jeweilige Komponist mit dem Mu-

sikstück gar nichts zu tun hat. Das sieht im ersten Moment ziemlich kompliziert und verwirrend aus, aber bei näherer Betrachtung erweist es sich als äußerst flexibles und mächtiges Konzept.

Die Ergebnismenge, die durch den JOIN entsteht, können Sie wie gewohnt mit der WHERE-Klausel einschränken. Die auszugebenden Spalten können Sie (in diesem Beispiel) auch wie gewohnt angeben. Unsere Liste aller Musikstücke mit ihren Komponisten erhalten Sie durch folgende Query:

```
SELECT Musiktitel, K_Name, K_Nr, Komponist
FROM MUSIKSTUECK,
     KOMPONIST
WHERE K_Nr = Komponist
;
```

Sicherlich wäre es einfacher, wenn DB2 sich in diesem Beispiel selbst an den (ihm durch die Datendefinition bekannten) Beziehungen zwischen den beiden Tabellen orientieren würde. Sie könnten sich dann diese WHERE-Klausel sparen. Aber dann könnten Sie nur in DB2 definierte Beziehungen zwischen Tabellen abfragen. Tabellen, die 'nichts voneinander wissen', weil sie in unterschiedlichen Verfahren benutzt werden, könnten Sie nicht in Relation setzen.

Da die Abfrage die Tabellen aber völlig 'unvoreingenommen' sieht, haben Sie die Möglichkeit, jede erdenkliche und in einer WHERE-Klausel beschreibbare Beziehung zwischen den Tabellen abzufragen. Das ist einerseits sehr flexibel und mächtig, andererseits besteht aber auch eine gewisse Gefahr, daß durch Unsicherheiten falsch formulierte Abfragen zu völlig unsinnigen Ergebnissen führen (oder die Ergebnisse korrekter Abragen mißinterpretiert werden).

Damit die erste Abfrage nicht zu schwierig ist, wurde in der Tabelle MUSIKSTUECK die Spalte mit der Schlüsselummer des Komponisten nicht K_Nr benannt, wie sie in der Tabelle KOMPONIST heißt. Üblicherweise sollte man aber Spalten mit gleichem Inhalt auch gleich benennen, damit ihre Zusammenhänge offensichtlich werden. Das bringt aber ein Problem mit sich: In der WHERE-Klausel müssen die beiden dann gleichnamigen Spalten miteinander verglichen werden. Die Codierung 'WHERE K_Nr = K_Nr' wäre äußerst zweideutig. DB2 gäbe auch eine Fehlermeldung aus, daß K_Nr mehrfach vorhanden ist und genauer bezeichnet werden muß.

Sie könnten die obige Abfrage auch wie folgt formulieren:

```
SELECT Musiktitel, K_Name, K_Nr, Komponist
FROM MUSIKSTUECK,
     KOMPONIST
WHERE KOMPONIST.K_Nr = MUSIKSTUECK.Komponist
;
```

Insbesondere, wenn Sie mit 'fremden' Tabellen, z.B. aus einem anderen Verfahren, arbeiten, wird diese Schreibweise aber unübersichtlich. Dann müßten Sie nämlich den Tabellen-Eigentümer (Owner) dem Tabellennamen voranstellen und der Name der Spalte würde dadurch aus drei Teilen bestehen. Sie können für die einzelne Abfrage auch Pseudonyme für die Tabellen definieren, indem Sie sie durch ein Leerzeichen vom Tabellennamen getrennt in der FROM-Klausel definieren. Sie können die Pseudonyme beliebig wählen, bis zu 18 alphanumerischen Zeichen, das erste muß ein Buchstabe sein. Es reicht, wenn Sie die Tabellen mit A, B, C usw. oder T1, T2, T3 etc. benennen. Die Query könnte beispielsweise so aussehen:

```
SELECT A.Musiktitel, B.K_Name, B.K_Nr,
       A.Komponist
FROM MUSIKSTUECK A,
     KOMPONIST   B
WHERE B.K_Nr = A.Komponist
;
```

Wie Sie sehen habe ich auch in der SELECT-Klausel den Pseudonamen der Tabelle vorangestellt. Das macht insgesamt die Abfrage übersichtlicher, weil Sie jeder Spalte sofort die Tabelle ansehen können, aus der sie stammt. Es ist also zu empfehlen, auch in einem Fall wie diesem, wo keine Namenskonflikte auftreten, die Spalten qualifiziert zu benennen, weil dann jedermann die Abfrage auch ohne Einblick in den Datenkatalog verstehen kann.

12.2.1 Performanceüberlegungen

Das Verfahren, das hinter der Verknüpfung mit Join steckt, scheint sehr kompliziert und umständlich, weshalb man glauben kann, daß es zu extrem langen Antwortzeiten führt. Scheinbar wird zuerst eine Art 'Zwischentabelle' erstellt, die um ein Vielfaches schneller wächst, als die Datenbestände selbst. Danach muß erst eine ziemlich komplexe Suche gestartet werden, die die inhaltlich korrekten Datensätze ermittelt, um aus diesen die gewünschten Sätze herauszufiltern.

Aber dem ist nicht so. SQL-Server haben in aller Regel einen Optimizer, der im günstigsten Fall 'kostenorientiert' arbeitet. Sofern es möglich ist, wendet der Optimizer die WHERE-Bedingungen direkt auf die Datentabellen an.

Wenn man drei Tabellen per Join miteinander verknüpft, dann muß man in der WHERE-Klausel die Schlüsselspalten aller drei Tabellen miteinander vergleichen, um die sinvollen Datensätze zu finden.

```
SELECT *
FROM Tab1 A,
        Tab2 B,
        Tab3 C
WHERE A.Key = B.Key
  AND A.Key = C.Key
  AND B.Key = C.Key
;
```

Sie werden bei kritischer Betrachtung feststellen, daß einer der beiden Vergleiche auf C.Key unnötig ist, da A.Key und B.Key von vornherein gleich sind. Man kann daher getrost auf eine der beiden Zeilen verzichten. Aber auf welche? Aus Gründen der Übersichtlichkeit würde ich vorschlagen, auf den Vergleich mit B.Key zu verzichten, da auch dem unvoreingenommenen Leser der Abfrage klar wird, daß es sich in beiden Fällen um denselben Wert handelt.

Sobald sie nur einen bestimmten Fall, also die Sätze zu einem Schlüssel nach der Spalte Key, ermitteln wollen, dann ist eine weitere Abfrage notwendig:

```
SELECT *
FROM Tab1 A,
        Tab2 B,
        Tab3 C
WHERE A.Key = B.Key
  AND A.Key = C.Key
  AND A.Key = 1000
;
```

Diese Abfrage filtert zunächst die inhaltlich korrekten Sätze aus der Ergebnistabelle aus, um unter diesen die Sätze mit dem gewünschten Schlüssel zu suchen. Soweit die Theorie!

In der Praxis versucht der Optimizer, die Anzahl der Zugriffe und Vergleiche zu minimieren. Er wird also zunächst (in dieser kleinen Abfrage jedenfalls) feststellen, daß der Vergleich auf die Konstante 1000 eine sehr große Einschränkung der Ergebnismenge zur Folge hat und diesen

Vergleich vorziehen. Wenn er dann noch einen Schritt weiter 'denkt', wird er denselben Vergleich mit der Konstante auch auf die Spalten B.Key und C.Key anwenden.

Voraussetzung dafür ist jedoch, daß die WHERE-Bedingung nicht allzu komplex ist. In der Praxis trifft das aber oft nicht zu. Daher kann man als Richtlinie heranziehen, daß in Fällen, die diesem vergleichbar sind, die WHERE-Klausel etwa so codiert wird:

```
SELECT *
FROM Tab1 A,
        Tab2 B,
        Tab3 C
WHERE A.Key = 1000
  AND B.Key = 1000
  AND C.Key = 1000
;
```

Auf diese Weise hat der Optimizer die größte Chance, die gewünschten Zusammenhänge zwischen den drei Tabellen zu ermitteln und aus allen drei wirklich nur die gewünschten Daten zu lesen.

Weitere derartige Überlegungen werden im Kapitel Performance angestellt.

12.3 UNION

Eine weitere Möglichkeit, die Daten aus unterschiedlichen Tabellen zu verknüpfen, ist das Keyword UNION. Es unterscheidet sich syntaktisch grundlegend von den bisher besprochenen Elementen.

UNION verknüpft zwei oder mehr gleichartig aufgebaute Ergebnistabellen unabhängiger SELECT-Statements zu einer gemeinsamen Ergebnistabelle. Im Gegensatz zum Joinen mehrerer Tabellen erstellt UNION nicht das Kreuzprodukt, sondern die Vereinigungsmenge. Hierzu müssen die Ergebnistabellen in den einzelnen SELECTs kompatibel sein, d.h. aus der gleichen Anzahl von Spalten bestehen, die denselben Typ haben.

UNION wird beispielsweise benutzt, um Abfragen gleichen Inhalts auf unterschiedliche Tabellen anzuwenden und für den Zeitraum der Abfrage zusammenzuführen. Aus unserer CD-Datenbank könnte man mittels UNION eine Tabelle erstellen, die alle Daten über alle erfaßten Personen (Komponisten, Texter, Interpreten) in einer gemeinsamen

Liste zusammenfaßt. Gewünscht sind jeweils Name, Geburts- und Todestag und das Instrument bzw. der Hinweis 'Komponist' bzw. 'Texter'.

Die Abfrage der Interpreten ließe sich so codieren:

```
SELECT I_Name, I_GebDat, I_Todestag,
       Instrument
FROM Interpreten
;
```

Nun müssen für die beiden anderen Tabellen entsprechende Queries codiert werden, die dieselbe Zahl an Spalten derselben Typen hat. Man sieht sofort: Es fehlt sowohl den Textern als auch den Komponisten ein Spalte, die äquivalent zu INSTRUMENT aus INTERPRET benutzt werden kann. Abhilfe schaffen hier Konstanten, die Sie in der SELECT-Klausel definieren können. Denn es soll ja (s.o.) auch ein Hinweis gegeben werden, ob es sich um einen Komponisten oder einen Texter handelt.

```
SELECT          K_Name, K_GebDat, K_Todestag,
                 'Komponist'
FROM          Komponist
;
SELECT          T_Name, T_Gebdat, T_Todestag,
                 'Texter'
FROM          Texter
;
```

Diese drei Abfragen müssen mittels UNION zu einer einzigen verknüpft werden:

```
SELECT          I_Name, I_GebDat, I_Todestag,
                  Instrument
FROM          Interpret

UNION

SELECT          K_Name, K_GebDat, K_Todestag,
                  'Komponist'
FROM          Komponist

UNION

SELECT          T_Name, T_Gebdat, T_Todestag,
                  'Texter'
FROM          Texter
;
```

Bitte beachten Sie, daß das Semikolon erst nach dem letzten SELECT-Statement auftaucht, da es sich hierbei um eine Abfrage handelt.

Ebenfalls werden Sie feststellen, daß die Ausgabe ausnahmsweise mal sortiert erfolgt. Die Zeilen sind immer nach allen Spalten aufsteigend sortiert (vgl. unbedingt auch UNION ALL im nächsten Abschnitt).

Wenn Sie eine andere Reihenfolge erzwingen wollen, so können Sie jedoch nur die gesamte Ergebnistabelle nach dem UNION sortieren, nicht die einzelnen Tabellen. In diesem Fall sollten Sie das Ergebnis nach dem Namen sortiert ausgeben, damit Ihnen doppelte Einträge auffallen.

Doch wie codiert man die ORDER BY-Bedingung? Die Spalte kann man nicht mit ihrem Namen bezeichnen, da die erste Spalte in den drei Datentabellen ja einen anderen Namen trägt. Nehmen Sie, wie im Abschnitt über die ORDER BY-Klausel beschrieben, die Nummer der Spalte. In unserer Query:

```
SELECT          I_Name, I_GebDat, I_Todestag,
                 Instrument
FROM          Interpret

UNION

SELECT          K_Name, K_GebDat, K_Todestag,
                 'Komponist'
FROM          Komponist

UNION

SELECT          T_Name, T_Gebdat, T_Todestag,
                 'Texter'
FROM          Texter

ORDER BY 1
;
```

Ein weiterer Einsatzpunkt von UNION ist die Erstellung 'mehrstufiger' Listen. Wenn Sie beispielsweise in einem Programm oder einem interaktiv erstellten Report alle CDs zusammen mit allen ihren Musiktiteln ausgeben wollen, so können Sie sich einen Join zusammenbasteln. Das Ergebnis enthielte jeweils eine Zeile pro Musiktitel, in welcher der Titel der CD und der Titel des Musikstücks enthalten sind. Ihnen reicht aber eine Titelzeile pro CD und darunter eine Zeile pro Musiktitel.

Mittels UNION können Sie diese Aufgabe mit ein paar Tricks leicht lösen:

```
SELECT CD_Nr, 0, CD_Titel
FROM CD

UNION

SELECT CD_Nr, Track, Musiktitel
FROM Musikstueck A,
     Aufnahme B
WHERE A.M_Nr = B.M_Nr

ORDER BY 1, 2
;
```

Das erste SELECT-Statement selektiert eine Zeile aus der Tabelle CD, also die 'Kopfzeilen' der Datengruppen. Das zweite SELECT erstellt eine Zeile pro Musikaufnahme. In der ersten Spalte beider Teilqueries steht die Schlüsselnummer der CD. Wenn man nach dieser Spalte sortiert, so erhält man alle Daten einer CD 'auf einem Haufen'. Um die Zeile aus der Tabelle CD an die erste Stelle zu rücken, kann man noch nach der zweiten Spalte sortieren. Diese enthält bei

'Kopfdaten' eine '0' (konstanter Wert); die Tracknummer der Musikaufnahmen beginnt aber immer mit der '1'. Auf diese Weise ist der CD-Titel immer vor den Titeln der Aufnahmen zu finden.

In einem Programm könnten Sie nun mit einer einzigen Abfrage das komplex strukturierte Ergebnis erreichen und müßten ggf. nur noch in der Verarbeitungsschleife anhand der Track_Nr (0 für CDs) die Ausgabe etwas anders formatieren.

12.4 UNION ALL

Das gerade erwähnte UNION hat allerdings performancemäßig einen entscheidenden Haken. Wie erwähnt erscheint die Ergebnismenge keinesfalls zwingend in irgendeiner Sortierfolge, sofern nicht eine bestimmte Sortierfolge hinter den SELECT-Statements angegeben wurde. UNION hat aber die Eigenschaft, die Zeilen der Ergebismenge implizit mit der "DISTINCT"-Funktion zu bearbeiten.

Hieraus folgt, daß es bei einer UNION-Abfrage keine zwei Zeilen mit gleichem Inhalt geben kann. Wenn Sie z.B. aus einer Tabelle mit Kunden- und einer Tabellen mit Mitarbeiterdaten die Namensspalten selektieren und diese mit UNION verknüpfen, dann werden doppelt auftretende Sätze entfernt. Mitarbeiter, die zugleich in der Kundenkartei enthalten sind, werden nur einmal ausgegeben.

Die zwangsläufige Folge hieraus ist, daß UNION zunächst die kompletten Zeilen der Ergebnistabelle in einen Zwischenspeicher sammelt und diese dann sortiert. Derartige Sortiervorgänge über Daten, die keinen gemeinsamen Index besitzen, dauern naturgemäß recht lange.

Leider ist dieses Vorgehen aus Gründen der Abwärtskompatibilität unumgehbar, obwohl die Ergebnismengen in den meisten Fällen schon deshalb keine unerwünschten doppelten Zeilen enthalten, weil die Herkunftstabelle durch Schlüsselfelder bzw. Konstanten (wie "Komponist" oder "Texter" in unserem Beispiel) erkennbar sind.

UNION ALL umgeht das Aussortieren doppelter Zeilen und gibt wirklich eine vollkommen unsortierte Ergebnismenge zurück, sofern Sie hinter dem letzten SELECT-Statement keine ORDER BY-Klausel einfügen. Da UNION ALL grundsätzlich eine wesentlich bessere Performance ergibt als UNION, ist es zu empfehlen, Abfragen in der Testphase zunächst mit

UNION ALL zu formulieren. Wenn unerwünschte doppelte Zeilen enthalten sind, kann zunächst durch Modifizierung der SELECT-Statements versucht werden, diese zu eliminieren. Insbesondere, wenn die identischen Zeilen aus derselben Tabelle stammen, ist diese Vorgehensweise zu empfehlen.

12.5 Subselect

Achtung: MySQL unterstützt derzeit noch kein Subselect.

Bislang haben Sie Möglichkeiten kennengelernt, die vorhandenen Zeilen von Datentabellen zusammenzufügen. Wenn Sie jedoch ermitteln wollen, zu welchen Zeilen in einer Tabelle es keine Gegenstücke in einer anderen Tabelle gibt, sind Join und UNION ungeeignet. Vielmehr müßte es eine Möglichkeit geben, in einer WHERE-Klausel mit dem Ergebnis einer anderen Abfrage zu vergleichen.

Überlegen Sie sich mal auf Basis von Join oder UNION eine Abfrage, die eine Liste aller Texter ausgibt, von denen keine Musikstücke gespeichert sind. Sie werden zu keinem Ergebnis kommen; diese Abfrage läßt sich nur mit Subselect lösen.

Eine solche Unterabfrage oder Subselect wird wie folgt codiert:

```
SELECT *
FROM TEXTER A
WHERE NOT EXISTS  (SELECT *
                   FROM MUSIKSTUECK B
                   Where A.T_Nr = B.Texter)
;
```

Beachten Sie, daß zunächst ein normales SELECT, hier über nur eine Tabelle, codiert wird. Es ist empfehlenswert, diese Tabelle mit einem Pseudonamen zu bezeichnen. Die WHERE-Klausel fragt mittels EXISTS die Ergebnismenge einer Unterabfrage ab.

Die Tabelle Texter (A) aus der "äußeren" Abfrage wird (theoretisch) Element für Element durchlaufen und für jede Zeile wird eine Unterabfrage auf die Tabelle Musikstueck (B) durchgeführt. Bei dieser Unterabfrage können Sie sich in der WHERE-Klausel auf den gerade bearbeiteten Fall der äußeren Abfrage beziehen. Die Bedingung EXISTS ist erfüllt, wenn das Subselect wenigstens eine Zeile Ergebnis liefert.

Bitte beachten Sie, daß hier sinnvollerweise - auch in Programmen! - ein SELECT * zu benutzen ist. Andere Codierungen, ein SELECT COUNT(*) z.B., würden u.U. immer eine Zeile Ergebnis liefern und die Bedingung wäre immer erfüllt. Außerdem muß man daran denken, daß man sich im inneren SELECT auf Werte aus dem Äußeren beziehen kann - und nicht umgekehrt!

Auch hier sollten Sie, was die Performance angeht, dem Optimizer auch zu erkennen geben, wie er die Abfrage am günstigsten durchführt. Wenn sie beispielsweise obige Abfrage logisch umkehren, also nur die Zeilen aus TEXTER haben wollen, zu denen in MUSIKSTUECK Referenzen vorhanden sind, sollten Sie das Subselect meiden. Ein Join ist für den Optimizer wesentlich einfacher in den Griff zu bekommen.

In unserem Fall ist das Subselect aber die einzige Möglichkeit, das gewünschte Ergebnis zu ermitteln. Sofern die beiden WHERE-Klauseln noch weitere Bedingungen enthielten, die in beiden SELECT-Statements codiert werden müßten (etwa ein GKZ), so sollten diese Bedingungen in beiden WHERE-Klauseln mit Konstanten bzw. HOST-Variablen auftauchen. Wenn im äußeren SELECT ein Vergleich auf "...AND GKZ = '111'..." vorkäme, dann sollte er im inneren SELECT genauso codiert sein und nicht beispielsweise "...AND A.GKZ = B.GKZ..."

Weitere Vergleiche mit Subselect sind:

- **IN**

  Vergleicht einen Wert links vom Schlüsselwort mit der Ergebnismenge des Subselect. Im Subselect darf nur eine Spalte angegeben sein, die vom Typ her mit dem Suchwert vereinbar ist. Der Vergleich ist ‘wahr', wenn wenigstens eine der Ergebniszeilen des Subselect dem Vergleichswert entspricht.

- **Vergleiche mit >, <, =, =>, <=, ^=**

  Auch hier darf nur eine Spalte in der Ergebnismenge des Subselect vorkommen, die mit dem links vom Vergleichsoperator stehenden Wert kompatibel sein muß. Außerdem darf das Subselect nur eine Zeile als Ergebnis erbringen. Man kann auf diese Weise auf COUNT (*), also die Anzahl der im Subselect gefundenen Zeilen, vergleichen.

Wenn die Bedingung sich auf die reine Existenz einer nicht leeren Ergebnismenge der Unterabfrage bezieht, dann ist die EXISTS-Formulierung absolut vorzuziehen. Sie ist erfüllt, wenn auch nur eine Zeile in der Ergebnismenge des Subselects enthalten ist. Folglich wird sie vom DB2 auch nach dem Auffinden der ersten Ergebniszeile beendet. Bei Abfragen wie "... WHERE 0 < (SELECT COUNT(*) FROM ..." haben zwar dasselbe Ergebnis, im Zweifelsfall muß das Subselect jedoch komplett abgearbeitet werden.

13 Verändern von Daten

SQL bietet - natürlich - nicht nur ein sehr mächtiges SELECT-Statement, sondern auch Möglichkeiten, Daten in Tabellen einzufügen und dort zu verändern und zu löschen. SQL ist "orthogonal" aufgebaut, also "rechtwinklig". Man könnte es auch als modular bezeichnen, da sich die Sprache aus immer wiederkehrenden Modulen zusammenfügt.

Daher werden Sie bei einzelnen Konstruktionen, die im Zusammenhang mit SELECT schon erklärt wurden, auf die entsprechenden Kapitel verwiesen.

13.1 INSERT

INSERT dient dem Einfügen von Zeilen in eine Tabelle. Es ist wieder ein komplettes Statement, das aus verschiedenen Klauseln aufgebaut ist.

13.1.1 Unqualifiziertes INSERT

Die simpelste Konstruktion des INSERT-Statements ist die unqualifizierte. Ähnlich einem SELECT * bezieht es sich auf die angesprochene Tabelle in ihrer vollen Breite (alle Spalten). Aus diesem Grund sollte es auch nur in interaktivem SQL genutzt werden.

```
INSERT INTO CD
VALUES (999, 'Bat out of Hell',
'123456789', 'XX-Verlag',
12, 13, 12, 'Rock', 32.95)
;
```

Sie sehen, daß für jede Spalte in der Tabelle ein Wert vom entsprechenden Datentyp angegeben ist. Es fällt auf, daß der Dezimalpunkt in diesem Statement als Komma definiert ist (wie auch in COBOL definierbar). Das führt zu einem kleinen syntaktischen Problem: Sobald SQL-Server ein Dezimalkomma erwarten, muß unbedingt darauf geachtet werden, daß nach den Kommata, die die einzelnen Werte trennen, Leerzeichen eingegeben werden. Wenn die Interpreten- und Komponisten-Nummern ohne Leerzeichen geschrieben wären, so könnte der Server anstelle zweier INTEGER- einen Fließkommawert erkennen und eine Fehlermeldung erzeugen (abhängig von den Landeseinstellungen für Dezimalpunkt oder -komma).

13.1.2 Qualifiziertes INSERT

Selbstverständlich können Sie auch INSERT-Statements formulieren, bei denen Sie die zu benutzenden Spalten und deren Reihenfolge ausdrücklich angeben. Dieser Form ist in Programmen sogar der Vorzug zu geben, da sie ein Statement innerhalb gewisser Grenzen von zukünftigen Änderungen der Datenstrukturen unabhängig macht.

Spalten in Tabellen relationaler Datenbanksystema haben üblicherweise nicht nur einen Datentyp, sondern auch noch eine NULL-Regel. Diese besagt, wie das Datenbanksystem zu verfahren hat, wenn bei einem INSERT für eine Spalte kein Wert angegeben wurde.

Eine Spalte kann mit der Option "NOT NULL" definiert werden, dann ist beim Einfügen in die Tabelle zwingend ein Wert für diese Spalte vorzugeben. Wenn keine NULL-Regel definiert wurde, kann die Spalte beim INSERT weggelassen werden. Ihr Inhalt wird auf NULL, also "Leer" gesetzt. Auch, wenn die NULL-Regel "NOT NULL WITH DEFAULT" lautet, kann die Spalte beim INSERT ausgelassen werden.

Anstelle eines vom Anwender gelieferten Wertes wird dann ein pro Datentyp definierter Standardwert eingefügt. Bei allen numerischen Spalten wird die "0" benutzt, bei Textketten aller Art werden diese mit Leerschritten aufgefüllt. Datums- und Uhrzeitfelder (incl. TIMESTAMP) werden mit dem aktuellen Datum bzw. der aktuellen Uhrzeit gefüllt.

Um ein qualifiziertes INSERT zu formulieren müssen Sie nach dem Namen der Tabelle und vor dem Schlüsselwort INSERT in Klammern und durch Kommata getrennt die Spaltennamen, auf die Sie sich beziehen wollen, eingeben.

Führen Sie das folgende Statement aus und stellen Sie fest, ob es funktionstüchtig ist. Führen Sie gegebenenfalls eine Fehleranalyse durch.

```
INSERT INTO CD
       (CD_Nr,CD_Titel, BestNr, Verlag,
        Titelzahl, Musikart, Preis)

VALUES (999, 'Bat out of Hell', '123456789',
        'WEA', 12, 'Rock', 24.95)
;
```

13.1.3 „BULK INSERT“

Eine Sonderform des INSERT ist ein Massen-Insert, auch Bulk-INSERT genannt. Durch ihn können Sie die Ergebnismenge eines SELECT-Statements in einem Schlag in eine DB2-Tabelle einfügen. Man benutzt ihn beispielsweise zum "Umbenennen" von Tabellen. In relationalen Systemen sind die Tabellennamen nicht wie Dateinamen einfach umbenennbar. Vielmehr müssen Sie eine neue Tabelle mit derselben Struktur anlegen, wie die alte Tabelle sie hat, und auf eine geeignete Weise die Daten übernehmen.

Dies kann mit folgender Formulierung erreicht werden:

```
INSERT INTO Tabelle_Neu
SELECT *
FROM Tabelle_Alt
;
```

- Bitte beachten Sie unbedingt, daß diese Methode nur für extrem kleine Testdatenbestände geeignet ist. Bei den Datenvolumina, mit denen Datenbanken üblicherweise gefüllt werden, müssen Sie andere Methoden benutzen, die der zuständige Systemadministrator für Sie vorbereiten und/oder durchführen kann.

13.2 DELETE

Das Löschen von Datensätzen ist durch die Modularität von SQL denkbar einfach (wenn man das bisher Angesprochene beherrscht).

```
DELETE
FROM          INTERPRET
WHERE         INSTRUMENT = 'Tromsaune'
;
```

Dieses Statement entfernt alle Interpreten aus der entsprechenden Tabelle, die das falsch eingegebene Instrument "Tromsaune" eingetragen haben. Wie Sie erkennen können besteht das Statement aus zwei bekannten Klauseln: der FROM-Klausel und der WHERE-Klausel.

Theoretisch können Sie alle erdenklichen und im SELECT-Statement erlaubten WHERE-Klauseln auch hier verwenden, aber diese Bedingung ist, was DB2 angeht, noch nicht vollkommen erfüllt. Allerdings nähert man sich von Version zu Version den 100% an. Aus diesem Grund wird hier auch nicht auf die Einschränkungen im Einzelnen eingegangen.

Bitte beachten Sie, daß auch das DELETE-Statement mengenorientiert arbeitet. Wenn Sie die WHERE-Bedingung weglassen, wird der gesamte Tabelleninhalt gelöscht. Eine versehentlich unsauber formulierte WHERE-Bedingung kann durchaus denselben Effekt haben.

Probieren Sie bei interaktivem Löschen, z.B. per SPUFI oder QMF, immer zuerst ein SELECT * mit der entsprechenden WHERE-Bedingung aus, um die Ergebnismenge zu überprüfen. Wenn sie dem entspricht, was Sie löschen wollen, können Sie im zweiten Schritt aus dem SELECT * ein DELETE machen.

13.3 UPDATE

Fast so einfach wie das DELETE-Statement ist das UPDATE-Statement, mit dem Sie bestehende Tabellenspalten modifizieren können.

```
UPDATE CD
SET Preis = Preis * 1.6 ,
    Musikart = 'Klassik'
WHERE Interpret = 12
;
```

Dieses Statement würde für alle Zeilen der Tabelle CD, die in der Spalte Interpret den Wert 12 enthalten, den Preis auf US-Dollar umrechnen und die Musikart auf 'Klassik' ändern. Sie sehen, daß man beim UPDATE beliebig viele Spalten der angesprochenen Tabelle auf einen Schlag ändern kann. Be-

achten Sie, daß nach dem numerischen Wert "1,6" und vor dem Komma, das die Änderungen voneinander trennt, ein Leerschritt steht. Da in deutschsprachigen Installationen als Dezimaltrenner üblicherweise das Komma definiert wird, kann ein SQL-Interpreter bei durch Komma getrennten Zahlen den Anfang und das Ende von Dezimalzahlen mit Nachkommastellen nicht mehr sauber erkennen.

So sollten daher die Kommata, die einzelne Werte voneinander trennen, von Leerschritten umschlossen sein, damit der Interpreter sie nicht versehentlich als Dezimaltrenner mißversteht.

Beachten Sie, daß auch hier eine unscharfe oder fehlende WHERE-Bedingung versehentlich die gesamte Tabelle verändern kann.

14 SQL in der Programmierung

Bisher haben Sie "einfache Abfragen" durchgeführt, die Sie direkt in SPUFI eingetippt und an DB2 übergeben haben. Man nennt diese Art, SQL-Statements zu verarbeiten, auch "interaktives SQL".

In Programmen können Sie natürlich derartige Abläufe nicht nutzen. Sie können jedoch auf zweierlei Art SQL-Statements in gängige Hochsprachen implementieren. Man spricht auch von Eingebettetem ("embedded") SQL. In dieser Schulung wird recht ausführlich die Einbettung in statischer Form eingegangen, da diese in Sachen Performance die beste Lösung darstellt und auch am einfachsten zu realisieren ist.

Es wird aber auch kurz auf die Einbettung dynamischer, also erst zur Laufzeit des Programmes erstellter, Statements eingegangen.

14.1 Prinzip des Embedded SQL

Höhere Programmiersprachen wie COBOL, C oder FORTRAN verstehen üblicherweise kein SQL. Einer der Vorteile von SQL ist jedoch, daß Sie die sehr mächtige Syntax, die Sie in den vorangegangenen Kapiteln erlernt haben, nahezu unverändert auch in der Programmiersprache Ihrer Wahl nutzen können.

Hier wird auf die Vorgehensweise bei DB2 eingegangen. ORACLE beherrscht Embedded SQL ebenfalls, MySQL und MS-SQL beherrschen es hingegen nicht.

14.1.1 Programmumwandlung

Um die Einbettung des Parasiten SQL in die Wirtssprache zu realisieren ist die Prozedur zum Umwandeln eines Programms mit SQL etwas anders, als bei Programmen, die ohne SQL auskommen. Normalerweise wandeln Sie das Quellprogramm zuerst mit einem Compiler um und linken danach die benötigten Module zusammen. Der Compiler würde jedoch schon beim ersten SQL-Statement, das im Source auftaucht, mit einer Fehlermeldung abbrechen.

Aus diesem Grund wird dem Compilerlauf ein Precompiler vorgeschaltet. Der Precompiler ist ein Programm, das zum Lieferumfang des benutzten SQL-Servers gehört und die Sourcen der unterschiedlichen Programmiersprachen für die Umwandlung mit dem normalen Compiler vorbereitet. Die ge-

naue Arbeitsweise des Precompilers hängt vom verwendeten SQL-Server ab. Wie gesagt: Hier wird nur auf die Funktionsweise bei DB2 eingegangen.

Der Precompiler extrahiert die SQL-Statements aus dem Quelltext und ersetzt diese durch einen Aufruf, der vom Compiler verstanden wird. Die SQL-Statements werden in Database Request Modules (DBRM) verpackt. Die DBRM werden als Member mit demselben Namen, wie das Source-Member in der als DBRMLIB angegebenen Datei gespeichert. Bei DB2 UDB heissen die Pakete übrigens „BINDFILES“ und haben die Endung .bnd

Im Source werden die SQL-Statements als Kommentare gekennzeichnet und durch Aufrufe von DB2-Modulen ersetzt, denen die Bezeichnung von DBRM und Statement-Nummer sowie weitere Daten übergeben werden, die vom jeweiligen Statement abhängen. Solche Schnittstellen nennt man API (Application Programming Interface).

Der so entstandene Quelltext sollte nun vom Compiler verstanden werden. Das daraus erstellte Linkmodul kann - wie üblich - vom Linkage Editor bearbeitet werden. Schließlich muß nur noch das DB2-eigene Modul hinzugelinkt werden, daß für die jeweilige Programmumgebung (TSO, IMS, MVS/BATCH, AIX, OS/2, Windows) die Funktionen der API zur Verfügung stellen.

14.1.2 Binden des Planes

Im Anschluß daran muß analog zum Linken des Programms ein DB2-Plan gebunden werden. In diesem Plan werden die anhand der Programmsourcen erstellten DBRMs zusammengefaßt und mit Berechtigungen versehen. Üblicherweise muß der Plan denselben Namen tragen, wie das Hauptprogramm (unter IMS ist das der Name des PSB zur Transaktion).

Diese Form des Bindens entspricht in etwa dem altertümlichen statischen Linken eines Programms. Wenn ein Unterprogramm ersetzt wird, muß nicht nur sein DBRM, sondern der komplette Plan neu gebunden werden. Das ist kritisch, wenn eine Vielzahl von Programmen ein bestimmtes, dynamisch gelinktes Unterprogramm benutzen.

Im DBRM und im erzeugten, SQL-freien Source speichert der Precompiler Datum und Uhrzeit des Precompilerlaufs ("Timestamp") ab. Bei Aufruf des Moduls prüft eine vom Pre-

compiler erzeugte Anweisung, ob das aktuell im Plan vorhandene DBRM und das benutzte Linkmodul denselben Timestamp tragen, um Inkonsistenzen zwischen Programm und DBRM zu verhindern.

Wenn ein neu umgewandeltes Programm dynamisch hinzugelinkt wird, im Plan des Hauptprogramms aber noch ein altes DBRM gebunden ist, wird noch vor Ausführung der ersten Anweisung ein SQL-Code -818 bzw. -805 erzeugt. Dieser weist auf die unterschiedlichen Timestamps von Programm und DBRM hin.

Seit DB2/MVS Version 2.3 können Pläne aber auch "dynamisch" gebunden werden. Hierzu werden nicht mehr DBRMs zu Plänen gebunden, sondern Packages. Ein Package entsteht aus einem bereits gebundenen DBRM. Wenn ein Package mit einer neuen Version eines DBRM gebunden wird, so wird diese neue Version sofort in allen Plänen aktiv, die das Package enthalten.

Die Common Server-Plattform von DB2 (z.B. DB2 for AIX) bindet nur noch Packages, die nicht mehr zu Plänen zusammengefaßt werden müssen. Dies gilt auch bei der Verwendung solcher Server als Gateway zu DB2 for MVS.
Aus Gründen des Handlings können Packages noch in Collections zusammengefaßt werden.

14.1.3 Form der Einbettung

Die Einbettung von SQL-Staments in höhere Programmiersprachen setzt voraus, daß der Precompiler sie auf Anhieb erkennen und überprüfen kann. hierzu werden sie - abhängig von der benutzten Programmiersprache - in bestimmte Schlüsselbegriffe eingebunden.

In COBOL hat die Einbindung die Form

```
EXEC SQL
        DELETE FROM ....
        WHERE ....
END-EXEC.
```

Die Programmiersprache C, die besonders auf UNIX-Rechnern (AIX, Linux) zum Einsatz kommt, bindet SQL-Statements wie folgt ein:

```
EXEC SQL
        DELETE FROM ....
        WHERE ....;
EXEC SQL
```

```
          INSERT INTO ....;
```

Die Unterschiede begründen sich darin, daß die für den Programmierer gewohnten Stukturelemente der jeweiligen Programmiersprache in der Einbettungssyntax nachempfunden wurden.

14.1.4 DELETE

Die Kommunikation zwischen dem Hostprogramm und den SQL-Statements geschieht über Hostvariable. Hierbei handelt es sich um gewöhnliche Variable der Hochsprache, deren Defintionen für den Precompiler besonders gekennzeichnet wurden. Sie werden innerhalb der SQL-Statements benutzt, um Werte an die Datenbank zu übergeben (bei INSERT und UPDATE bzw. in einer WHERE-Klausel) und werden bei SELECT-Abfragen von der Datenbank gefüllt zurückgegeben.

Die Hostvariablen müssen vom Datentyp her kompatibel zu den jeweils benutzten Spalten sein. Weiterhin müssen es statische, globale Variablen sein.

Weitere Kommunikationsbereiche heißen SQLCA und SQLDA. Die SQLDA wird von den API-Funktionen genutzt, um Daten mit DB2 auszutauschen. In der SQLCA (SQL-Communication Area) werden nach jedem Statement der SQL-Returncode und weitere Angaben zu Fehler, Verarbeitungskosten, Anzahl der angesprochenen Zeilen etc. zurückgegeben. Die Datenstrukturen der SQLCA sollten, die der SQLDA können auf eine Programmanweisung hin vom Precompiler automatisch in den Source eingefügt werden.

Wenn keine SQLCA vorhanden ist, so kann das Progamm nur den SQLCODE der Abfragen auswerten.

In einem C-Programm sähen die Anweisungen an den Precompiler so aus:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
EXEC SQL BEGIN DECLARE SECTION;
        int zahl;
        char vorname[40];
        [...weitere Variablendeklarationen...]
EXEC SQL END DECLARE SECTION;
```

Die Variablendefinitionen innerhalb der DECLARE-SECTION müssen in allen Programmiersprachen statisch und global sein. Der Precompiler ersetzt in den API-Aufrufen die Variablen durch Verweise auf deren Speicheradressen, daher können keine lokalen oder dynamischen Variablen in als Hostvariablen benutzt werden.

DB2/MVS kann über das Utility "DCLGEN" anhand der Tabellendefinition aus dem Online-Katalog die für eine Tabelle notwendige Declare-Section für die Programmiersprachen COBOL, COBOL2, FORTRAN, PL/1, ASSEMBLER und C erzeugen. Hierbei wird pro Tabellenspalte eine Variable angelegt, deren Name vom Namen der Tabellenspalte abgeleitet wird und die einen kompatiblen Datentyp enthält.

In WHERE-Bedingungen ist eine völlige Kompatibilität zwischen Hostvariablen und Spalten nicht notwendig, sie steigert aber möglicherweise die Performance der Abfragen.

LIKE-Abfragen auf Textketten müssen, sofern sie den Suchbegriff in einer Hostvariablen enthalten, sorgfältig behandelt werden. Technisch gesehen kann die Suchvariable in jeder Art von Textkette abgelegt werden, in 95% aller Fälle ist es jedoch erforderlich, diese Textkette variabel lang zu definieren. Beziehen Sie sich in diesem Zusammenhang bitte auf die Definition einer Hostvariable für eine VARCHAR-Spalte, wie Sie in der von Ihnen benutzten Programmiersprache üblich ist.

14.1.4.1 Einbettung der Statements

Die Einbettung von SQL-Statements mit Hostvariablen wird anhand eines UPDATE-Statements beschrieben, da sich hierbei keine Syntaxerweiterungen im Vergleich zu der bisher kennengelernten Form ergeben. Lediglich Werte, die vom Programm ermittelt werden, sind als Hostvariablen eingebunden:

```
EXEC SQL
        UPDATE Personen
        SET         Gehalt = :PERSONEN-GEHALT,
                   Stufe = Stufe + 1,
                Vorlaeufig = 'Ja',
                LUPDATE = CURRENT DATE,
                LUPTIME = CURRENT TIME,
                LUPUSER = :SYS-USER-ID
        WHERE P_Nr = :PERSONEN-P-NR
END-EXEC.
```

In diesem (COBOL-) Beispiel können Sie sehen, daß Ihnen auch in Programmen alle Datenquellen des interaktiven SQL zur Verfügung stehen. Die Spalte "Stufe" wird aus einer Formel gefüllt, die Spalten "LUPDATE" und "LUPTIME" aus besonderen SQL-Variablen. "Gehalt" und "LUPUSER" werden aus Hostvariablen gefüllt, deren Name zur Kennzeichnung ein Doppelpunkt vorangestellt wird.

Die Personennummer "P_Nr", auf die sich die Änderungen beziehen, wird auch mit einer Hostvariablen verglichen. An dieser Stelle könnten Sie ebenso Vergleiche mit Konstanten, anderen Spalten und speziellen SQL-Variablen anstellen.

Bitte bedenken Sie, daß DB2 ein Datenbanksystem und kein Taschenrechner ist. Die Formulierung einer WHERE-Bedingung mit Hostvariable kann formell gesehen auch so gestaltet sein:

```
... WHERE Alter > :SUCH-ALTER - 30 ...
```

Hier wird aber DB2 gezwungen, sich den gesuchten Wert selbst zu berechnen. Dies kann zu extrem langen Laufzeiten dieser Abfrage führen. Es ist absolut empfehlenswert, in solchen Fällen vor der Abfrage in der benutzten Hochsprache eine Berechnung durchzuführen und dem SQL-Statement die fertig gefüllte Hostvariable zu übergeben.

14.2 Erweiterungen der Syntax

SQL ist eine mengenorientierte Abfragesprache. Mit dem SELECT-Statement beschreiben Sie eine Ergebnismenge. Diese müssen Sie zur Weiterverarbeitung in herkömmlichen Programmiersprachen zunächst in ihre einzelnen Elemente aufteilen. Das kann das SELECT-Statement nicht erreichen. Doch selbst, wenn durch Nutzung von Skalarfunktionen nur eine einzelne Zeile vom SELECT-Statement zurückgegeben wird, muß seine Syntax noch erweitert werden. Beispielsweise muß man eine Möglichkeit schaffen, die Hostvariablen

zu definieren in welchen die Ergebnisse abgelegt werden sollen.

14.2.1 INTO

Bei der Abfrage einzelner Zeilen, z. B. bei Verwendung skalarer Funktionen, kann in das SELECT-Satement eine Hostvariable als Ziel des Abfrageergebnisses eingebettet werden. Hierzu ist die Erweiterung des SELECT-Statements um eine INTO-Klausel erforderlich. Die Anzahl von CD-Titeln in unserer Datenbank ermitteln Sie z. B. mit der folgenden Abfrage in einem COBOL-Programm:

```
EXEC SQL
        SELECT COUNT(*)
        INTO   :ZAHL
        FROM   CD
END-EXEC.
```

Nach Ausführung der Abfrage enthält die Programmvariable ZAHL, die einen numerischen Typ haben muß, die Anzahl der Zeilen in der Tabelle CD. Natürlich muß ZAHL - wie alle anderen Hostvariablen auch - in der SQL-DECLARE-SECTION am Programmanfang statisch vereinbart werden.

Wenn mehr als eine Spalte in der Ergebnismenge enthalten ist, so müssen Sie selbstverständlich für jede dieser Spalten eine Hostvariable des entsprechenden Typs vorhalten. Wenn die Abfrage mehr als genau eine Zeile als Ergebis liefert, erhalten Sie zur Laufzeit des Programms einen SQL-Fehler.

14.2.2 CURSOR-Verarbeitung

Um SELECT-Statements mit mehr als einer Zeile Ergebnis abarbeiten zu können, müssen Sie mit DB2 einen CURSOR vereinbaren, einen Zeiger auf jeweils ein Element der Ergebnismenge. Ein CURSOR entspricht in seiner Verarbeitung in etwa einer Datei: Er wird, im SELECT-Statement deklariert, vor der Verarbeitung geöffnet. Die einzelnen Zeilen werden wie Datensätze gelesen und am Ende schließt man den Cursor wieder.

14.2.2.1 DECLARE CURSOR

Zur Deklaration des Cursors wird wiedereinmal das SELECT-Statement erweitert. Um eine Ergebnismenge abzuarbeiten, wird das SELECT-Statement wie folgt aussehen:

```
EXEC SQL
        DECLARE Cu1 CURSOR FOR
        SELECT CD_Nr, CD_Titel
```

```
         FROM CD
END-EXEC.
```

In dieser Abfrage wird ein Cursor mit der Bezeichnung Cu1 als Zeiger auf die Ergebnismenge eines bestimmten SELECT vereinbart. Der Name des Cursor kann unter den üblichen Namenskonventionen frei gewählt werden, darf im jeweiligen Sourcemodul aber nur ein einziges Mal vereinbart werden. Auch, wenn Sie innerhalb des Programms dafür Sorge tragen, daß zwei bestimmte Cursor nie zugleich offen sind, müssen diese verschiedene Namen tragen.

Gleichzeitig hat der Name eines Cursors nur innerhalb eines Quellmoduls Gültigkeit. Wenn Sie den Code aus mehreren Quelltexten erstellen, deren Objectfiles Sie später zusammenlinken, müssen Sie darauf achten, daß die gesamte Verarbeitung eines Cursors innerhalb desselben Stücks Quelltext abläuft. Da die verschiedenen Quelltext-Module unabhängig voneinander duch den Precompiler laufen und eigene DBRM ergeben, kann die Konsistenz zwischen SELECT-Anweisung und den Hostvariablen beim späteren FETCH weder beim Precompile noch beim Bind geprüft werden.

14.2.2.2 OPEN und CLOSE

Denkbar einfach ist das Öffnen und Schließen eines Cursors. Vor Eintritt in die Verarbeitungsschleife codieren Sie

```
EXEC SQL
         OPEN Cu1
END-EXEC.
```

und nach Beendigung der Schleife

```
EXEC SQL
         CLOSE Cu1
END-EXEC.
```

Sie können den Cursur zu jeder Zeit wieder schließen (auch vor Erreichen des letzten Satzes) und neu öffnen, um die Verarbeitung der Ergebnismenge von vorn zu beginnen.

Bei statischem SQL, wie es in den bisherigen Beispielen codiert wurde, müssen Sie ein paar Besonderheiten beachten:

- Der erste Zugriff auf Daten erfolgt nicht zu dem Zeitpunkt, zu dem das SELECT-Statement im Programmfluß erreicht wird, sondern erst beim OPEN. Das SELECT-Statement wird aus diesem Grund vom Precompiler als

Kommentar markiert und das entsprechende DBRM erst beim OPEN angesprochen.

- Nach jedem Statement sollte unbedingt der SQLCODE abgefragt werden, um fehlerhafte Operationen zu erkennen.
- Die Statements müssen zwingend in dieser Reihenfolge im Source auftauchen: DECLARE ... FOR SELECT ... vor OPEN, danach das Einlesen der Zeilen mittels FETCH, am Ende CLOSE.

14.2.2.3 FETCH

7.2.2.3 FETCH

Zum Einlesen der einzelnen Zeilen in ihr Programm benutzen Sie das FETCH-Statement. Es besteht lediglich aus zwei Teilen: Der FETCH-Klausel, in der Sie den Cursor bezeichnen, auf den Sie sich beziehen, und der INTO-Klausel, die sytaktisch exakt so aussieht, wie die im SELECT-Statement.

Nach dem FETCH müssen Sie unbedingt den SQLCODE abfragen, denn über ihn bekommen Sie den einzigen Hinweis darauf, ob Sie das Ende der Abfragemenge erreicht haben, oder nicht. Die überwiegende Mehrzahl der SQL-Datenbanksysteme (auch DB2 auf allen Plattformen) liefern einen SQL-CODE von +100, wenn bei einem FETCH das Ende der Abfragemenge überschritten wurde.

Das bedeutet, daß die beim FETCH eingelesenen Daten schon nicht mehr gültig sind. Folglich muß die Verarbeitungsschleife nach dem Auftreten von SQLCODE +100 verlassen werden, ohne den gerade eingelesenen Datensatz noch zu verarbeiten.

```
EXEC SQL
        FETCH Cur1 INTO :Variable1, :Variable2
END-EXEC.
```

14.2.2.4 DECLARE CURSOR

Eine Erweiterung der Syntax bringt die Cursorverarbeitung auch in der WHERE-Klausel mit sich. Unter der Bedingung, daß in einer SELECT-Klausel mit Cursor nur eine Tabelle angesprochen wird - und dies ohne skalare Funktionen -, kann man sich in UPDATE und DELETE-Statements auch auf den letzten mit FETCH eingelesenen Datensatz beziehen:

```
EXEC SQL
        UPDATE CD
        SET Preis = :CD-PREIS
        WHERE CURRENT OF Cu1
```

```
END-EXEC.
```

Die WHERE-Klausel darf in diesem Fall nur aus diesem einen Element bestehen. Alle weiteren Vergleiche sind sinnlos und werden daher nicht zugelassen.

14.2.3 FOR UPDATE OF

Ein möglicher Grund für Laufzeitfehler während einer Anwendung sind Lock- und Deadlock-Situationen. DB2 (und alle anderen Datenbankträger auch) reservieren Datensätze für die angeschlossenen Anwender je nach der vermutlich gewünschten Verarbeitung. Wenn Sätze gelesen wurden (mit einer CURSOR-Verarbeitung beispielsweise), dann werden sie normalerweiseweise bis zum Ende der Verarbeitung für Updates durch andere Benutzer gesperrt. Lesender Zugriff ist weiterhin möglich.

Wenn Sie in einer CURSOR-Verarbeitung eine Tabelle Zeile für Zeile abarbeiten und bestimmte Zeilen mit WHERE CURRENT OF löschen oder verandern möchten, so kann es passieren, daß die entsprechenden Zeilen auch noch von anderen Anwendern gerade gelesen wurden. Solange, bis diese Anwender ihre aktuellen Verarbeitungen abgeschlossen haben, ist ein Update auf diese Zeilen nicht möglich.

Sofern Updates auf Zeilen der gerade gelesenen Tabelle mittels WHERE CURRENT OF vorgesehen sind, empfiehlt es sich, im SELECT-Statement bereits auf die UPDATE-Absicht hinzuweisen. hierdurch werden die aktuell bearbeiteten Zeilen exclusiv für die eigene Anwendung reserviert. Hierbei kann man auch die Update-Absicht auf bestimmte Spalten der Tabelle beschränken, falls der Programmentwickler nicht alle Spalten updaten darf.

```
EXEC SQL
        DECLARE Cur1 CURSOR FOR
        SELECT          CR_Nr, CD_Titel
        FROM          CD
        FOR UPDATE OF CD_Titel
END-EXEC.
```

14.2.4 FOR FETCH ONLY

Im umgekehrten Fall kann es aber auch passieren, daß DB2 anhand eines SELECT-Statements ohne FOR UPDATE OF mit Updates auf selektierte Daten rechnet und diese von vornherein exklusiv reserviert. Um dies zu vermeiden können Sie anstelle von FOR UPDATE OF die Formulierung FOR FETCH ONLY verwenden. So können Sie sichergehen, daß die Datensätz nur "shared" ausgelesen und zum frühestmöglichen Zeitpunkt wieder freigegeben werden.

14.2.5 OPTIMIZE FOR

In vielen Fällen, z.B. bei IMS-Transaktionen, wird dem Anwender grundsätzlich nur ein Teil der Ergebnismenge angezeigt. Eine IMS-Transaktion, die ein Blättern in einer Liste von CDs erlaubt, müßte zB. für jeden Blättervorgang ein SELECT mit OPEN, FETCH und CLOSE durchführen, um von der ersten Zeile der auf der neuen Seite anzuzeigenden Tabelle an die entsprechende Menge von Zeilen zu ermitteln. Die Ergebnismenge der einzelnen SELECT-Statements ist dabei u.U. wesentlich größer als der angezeigte Teil.

Der DB2-Optimizer kann das beim Optimieren der Statements aber nicht feststellen und muß davon ausgehen, daß im Zweifelsfall die gesamte mit WHERE beschriebene Teilmenge der Tabelle angezeigt werden soll. Das kann möglicherweise dazu führen, daß extrem lange Antwortzeiten entstehen, weil DB2 jedesmal erst viel zu große Teile der Datenbank einliest und sortiert. Man kann derartige Effekte vermeiden, indem man DB2 im SELECT die Anzahl der Sätz vorgibt, die man zwischen OPEN und CLOSE jeweils verarbeiten will.

```
EXEC SQL
        DECLARE Cur1 CURSOR FOR
        SELECT CD_Nr, CD_Titel, CD_Verlag
        FROM CD
        WHERE CD_Nr >= :LETZTE-NUMMER
        ORDER BY CD_Nr
        OPTIMIZE FOR 15 ROWS
END-EXEC.
```

Hier würde DB2 zunächst entscheiden, ob es schneller geht, die ersten 15 Zeilen des Ergebnisses direkt aus den Dateien herauszusuchen, oder die gesamte Datenbank in den Zwischenspeicher (BUFFERPOOL) zu laden.

Bei IMS-Transaktion mit Blätterfunktion ist grundsätzlich die Angabe von OPTIMIZE FOR zu empfehlen. OPTIMIZE kann auch positive Wirkungen haben, wenn man mit einem geöffneten Cursor in einer TSO-Anwendung (oder RS/6000-Anwendung) durch eine Tabelle blättert, was allerdings im Einzelfall ausprobiert werden muß.

14.2.6 WITH HOLD

Ein CURSOR kann auch mit der Option WITH HOLD vereinbart werden, wenn er über die Grenze einer Transaktion hinaus (Siehe auch 7.3, Transaktionsgrenzen) Gültigkeit haben soll. Die Syntax lautet

```
EXEC SQL
        DECLARE Cur1 CURSOR WITH HOLD FOR
        SELECT CD_Nr
        FROM CD
END-EXEC.
```

Unter Transaktion ist in diesem Zusammenhang eine Zusammenfassung sachlich eine Einheit bildender Datenbankoperationen zu verstehen. Es ist nicht nur eine IMS-Transaktion gemeint. Prinzipiell gilt aber, daß eine IMS-Transaktion eine solche Einheit bildet.

14.3 Transaktionsgrenzen

Ähnlich der Verarbeitung in transaktionsorientierten Systemen wie IMS kann auch DB2 die zu einer Verarbeitung (Transaktion) gehörenden Datenbankoperationen zu einer Einheit zusammenfassen. Dies vereinfacht u.a. die Programmierung von Fehlerroutinen und verhindert inkosistente Datenbestände nach Programmabbrüchen.

14.3.1 COMMIT

In relationalen Datenbanken werden die "Fälle" meist über eine Vielzahl von Tabellen verteilt abgespeichert. Das Erfassen einer neuen CD z.B. wird wenigstens in den Tabellen CD und AUFNAHME zu neuen Zeilen führen. Diese INSERT-Statements gehören inhaltlich zusammen. Die Zeilen in der Tabelle AUFNAHME ergeben ohne die Zeile in der Tabelle CD keinen Sinn - und umgekehrt.

Bei Systemabbrüchen vor dem Abschluß der Transaktion "Erfassen einer neuen CD" würde DB2 ohne entsprechende Automatismen die Tabellen in einem inkonsistenten Zustand hinterlassen. Dem Datenbanksystem kann nicht klar sein, welche Veränderungen zusammengehören und wie unvollständige Transaktionen nach einem Abbruch zurückzusetzen sind.

Durch das Statement COMMIT können Sie DB2 (und anderen DB-Trägern) mitteilen, daß die aktuelle Transaktion abgeschlossen ist. Durch COMMIT werden alle seit dem letzten COMMIT bzw. dem Programmstart durchgeführten Änderungen an Daten festgeschrieben. Bei Abbrüchen durch Programmfehler oder Systemabbrüche werden bei nächster Gelegenheit alle Transaktionen, die noch nicht committiert sind, vollständig rückgängig gemacht. Man spricht hier von einer RECOVERY; den Zeitraum zwischen zwei COMMIT-Punkten bezeichnet man als UNIT OF RECOVERY, UNI OF WORK oder UNIT OF CONFIDENCE.

COMMIT hebt zugleich alle Reservierungen von Datensätzen auf und schließt alle offenen CURSOR, die nicht mit der Option "...WITH HOLD" vereinbart wurden (Siehe 7.2.6). Die Syntax lautet

```
EXEC SQL
       COMMIT
END-EXEC.
```

Automatisch werden COMMIT-Punkte beim Ende des Hauptprogrammes gesetzt, nach dem der Plan benannt wurde.
Das Setzen ausdrücklicher COMMITs sowohl in Online- als auch in Batchprogrammen kann nur dringend empfohlen werden. Sofern ein Anwender mehrere Stunden in einer Anwendung arbeitet, die keine COMMIT-Punkte nach Updates setzt, sperrt er bis zum Beenden des Programmes alle geänderten Datensätze. Weiterhin wird bei einem Fehlerabbruch des Programmes automatisch ein ROLLBACK ausge-

führt und möglicherweise die gesamte Tagesarbeit rückgängig gemacht.

14.3.2 ROLLBACK

Sie können die Transaktionsgrenzen und das Rückgängigmachen einer kompletten Transaktion auch in Ihren Programmen gezielt als Fehlerroutine einsetzen.

Beispielsweise besteht die Möglichkeit, daß eine Transaktion mit Veränderung von Daten fehlschlägt, weil bestimmte Datensätze durch andere Anwender blockiert sind und nicht geändert werden können. Andererseits können auch während des Updates Laufzeitfehler oder ähnliches auftreten.

In diesen Fällen müßten Sie - je nach Phase der Verarbeitung, an welcher der Fehler auftrat - gezielt alle Änderungen rückgängig machen. Bei Updates müßten Sie hierzu vor dem Update den alten Inhalt der Tabellenzeilen auslesen und zwischenspeichern.

Einfacher geht es mit ROLLBACK. Dieses Statement macht alle Änderungen an Datenbeständen seit dem letzten COMMIT bzw. dem Programmstart rückgängig. Es ist sinnvoll, nach jedem UPDATE den SQL-Code abzufragen und bei Fehlercodes unverzüglich in eine Fehlerroutine zu verzweigen, die den SQL-Fehler auswertet und für die Programmierer bzw. die Systemadministration sichtbar macht und einen ROLLBACK in der folgenden Syntax durchführt:

```
EXEC SQL
        ROLLBACK
END-EXEC.
```

14.4 Dynamisches SQL

Dymamisches SQL kommt immer dann zum Einsatz, wenn die genaue Abfrage gegen die Datenbank während der Programmierphase noch nicht feststand. Dies kann beispielsweise der Fall sein, wenn Anwender die komplette WHERE-Bedingung selbst zur Laufzeit des Programms erstellt, beispielsweise als Ad Hoc-Abfrage. Dynamisches SQL muß auch dann benutzt werden, wenn die Tabelle zum Zeitpunkt der Programmerstellung noch nicht erstellt war.

Manche Software erstellt Tabellen für Anwenderdaten sogar erst "vor Ort". Deren Namen und Strukturen sind daher während der Programmierung nicht bekannt, was ebenfalls ein dynamisches Erstellen der SQL-Statements vorausetzt.

14.4.1 DECLARE CURSOR

Statisches SQL wird von DB2 zunächst Precompiliert, die entstandenen DBRMs werden später gebunden. Beim Precompilieren führt DB2 eine Syntaxüberprüfung der im Source enthaltenen Statements durch und wandelt diese in eine verkürzte ("tokenisierte") Form um.

Beim Binden wird überprüft, ob Tabellen und Statements zueinander passen und ob der User, der den Plan bindet, die Berechtigungen zum Zugriff auf die Tabellen besitzt. Weiterhin stellt der Optimizer fest, wie bei den codierten Statements der performancegünstigste Zugrff auf die Daten durchgeführt werden kann.

Diese Tätigkeiten müssen bei dynamischem SQL zur Laufzeit des Programmes auf eine Anweisung hin durchgeführt werden. Diese Anweisung heißt PREPARE.

Zunächst muß das fragliche Statement in eine String-Variable gestellt werden. Innerhalb des Strings können wieder alle Statements so codiert werden, wie sie auch interaktiv abgesetzt werden könnten. Selbstverständlich können auch Hostvariable benutzt werden; an ihrer Stelle wird aber noch nicht der Name der Variable, sondern ein Fragezeichen codiert. Ein derartiges Statement könnte so aussehen:

```
INSERT INTO CD (CD_Nr, CD_Titel) VALUES(?,?)
WHERE CD_NR = ?
```

Die Vorbereitung des Statements durch PREPARE erzeugt eine Art "temporäres DBRM", das mit einem im Programm eindeutigen Namen versehen sein muß:

```
EXEC SQL
       PREPARE Stmnt1 FROM :STATEMENT
END-EXEC.
```

Bei erfolgreicher Ausführung existiert nun ein fertig vorbereitetes Statement, das den Bezeichner Stmnt1 besitzt.

14.4.2 EXECUTE

Der einfachste Weg, dieses Statement auszuführen, ist EXECUTE bzw. EXECUTE USING. Die simpelste Form des EXECUTE wird wie folgt codiert:

```
EXEC SQL
       EXECUTE Stmnt
```

```
END-EXEC.
```

Dies funktioniert natürlich nicht, wenn im vorbereiteten Statement Platzhalter für Hostvariable ("?") vorhanden sind. Diese müssen nun noch durch Verweise auf die tatsächlichen Variablen ersetzt werden.

```
EXEC SQL
        EXECUTE Stmnt
        USING :CR_Nr, :CD_Titel, :CD_Such
END-EXEC.
```

Hierbei werden die angegebenen Hostvariablen in der Reihenfolge, in der sie im EXECUTE-Statement auftauchen, für die Platzhalter eingesetzt.

Wie Sie erkennen, ist auch für dynamisch erzeugte Statements ein wenig statisches SQL erforderlich, denn auch die PREPARE- und EXECUTE-Statements müssen vor der Umwandlung des Programms precompiliert werden, wodurch ein DBRM erzeugt wird, das zu einem Plan gebunden werden muß.

14.4.3 EXECUTE IMMEDIATE

Die gerade gezeigte Form der Ausführung dynamischer Statements hat unverzichtbare Vorteile, wenn man das Statement in einer Programmschleife mehrmals, jedesmal aber mit anderen Hostvariablen ausführen möchte. In diesem Fall müssen Sie den PREPARE-Schritt genau einmal pro Programmlauf ausführen und können das Statement bis zum Programmende beliebig oft benutzen.

Wenn Sie ein dynamisches Statement erstellen, das Sie genau einmal ausführen müssen, dann können Sie DB2 veranlassen, beide Schritte zusammengezogen auszuführen. Allerdings dürfen auf diese Weise mit EXECUTE IMMEDIATE ausgeführten Statements keine Hostvariablen beinhalten. Das macht auch Sinn, denn wenn Sie das Statement nur ein einziges Mal in dieser Form ausführen, können Sie die Inhalte der Hostvariablen auch direkt als Konstante in das Statement einbetten.

```
EXEC SQL
        EXECUTE IMMEDIATE :STATEMENT
END-EXEC.
```

15 Performance

An verschiedenen Stellen haben wir uns bereits Gedanken zur Performanceverbesserung gemacht. Diese Überlegungen sollen an dieser Stelle nochmals zentral wiederholt und ergänzt werden.

15.1 Datenmodellierung

Bereits in der Datenmodellierung können Sie nicht unerheblich Einfluss auf die Performance von DB-Abfragen nehmen. Hierbei ist weniger die logische (abstrakte) Struktur der Daten, sondern deren physische Implementierung ausschlaggebend.

15.1.1 Datentypen

Die Auswahl von Datentypen - insbesondere der Schlüsselspalten - kann die Performance beeinflussen.

15.1.1.1 Numerische Datentypen

Generell kann gesagt werden, dass Spalten möglichst kurz sein sollen, da sich hierdurch die Anzahl der pro I/O-Operation gelesenen Zeilen erhöhen lässt. Wenn ganzzahlige Werte gespeichert werden müssen, ist ein Abspeichern in INTEGER oder SMALLINT-Spalten meist günstiger, als in DECIMAL-Spalten.

Auch bei Vergleichsoperatoren in WHERE-Klauseln erweisen sich die COMPUTATIONAL gespeicherten Integer-Werte meist als performancegünstiger als die PACKED gespeicherten DECIMAL-Werte.

15.1.1.2 Character-Spalten fester Länge

CHAR-Felder sollten auf die notwendige Länge reduziert werden. Umfangreiche Texte, die zu speichern sind, sollten in sequentiellen Dateien und nicht in der Datenbank abgelegt werden; DB2 ist kein Textverarbeitungsprogramm.

Die Common Server-Plattform bietet ihrerseits zum Abspeichern der sequentiellen Dateien in der Datenbank die Datentypen CLOB und BLOB (Character Large Object und Binary Large Object) an. In diesen Spalten können binäre bzw. Characterdateien abgelegt werden, die mit den Sicherungen der Datenbank synchron gehalten werden.

Sofern die Datenbank aus technischen oder anderen Gründen auf einen früheren Stand zurückgedreht werden muss, werden auch die damals aktuellen BLOB und CLOB-Inhalte restauriert. Diese Spaltentypen dürfen nur in den Fällen benutzt werden, in denen diese Synchronisie rung nicht anders realisiert werden kann. Ansonsten führen sie derzeit zu erheblichem Overhead im Speicherplatzbedarf.

Für CLOB und BLOB muss nämlich beim Anlegen der Tabelle eine Maximalgröße der zu speichernden Dateien angegeben werden. Dieser Speicherplatz wird dann unabhängig von der tatsächlichen Länge der Dateien für jeden Datensatz verbraucht.

BLOB und CLOB-Spalten werden abseits der normalen Tabellenspalten in eigenen Dateien abgespeichert. Sie dürfen nicht in WHERE- Bedingungen benutzt werden. Dafür beeinträchtigen sie die Performance der Abfragen nur, wenn sie auch tatsächlich angesprochen werden.

15.1.1.3 VARCHAR-Felder

Variabel lange Spalten sind eingeführt worden, um bei stark variieren der Länge des tatsächlichen Feldinhalts erheblich Speicherplatz zu sparen. Es ist jedoch zu bedenken, dass die Performanceeinbußen durch derartige Spalten z.T. erheblich sein können.

Beim Abspeichern von Daten benutzt DB2 einen Mechanismus, der auf Zeilen mit fester Länge optimiert wurde. Bei Zeilen mit VARCHAR- Spalten ist die Länge aber vom tatsächlichen Inhalt abhängig. Daher muss zunächst die Länge der Zeile berechnet werden. Diese muss dann mit dem freien Speicherplatz in allen Pages verglichen werden, um eine Page zu finden, in der ausreichend Platz ist.

Bei Zeilen mit fester Länge und den üblichen Definitionen der Table spaces (Datendateien) kann DB2 die Suche nach freiem Raum im Vergleich zu variablen Zeilenlängen erheblich abkürzen.

Das gleiche Berechnungsproblem ergibt sich, wenn WHERE-Bedingun gen sich auf Spalten beziehen, die hinter einer VARCHAR-Spalte in der Tabelle stehen. Für jede einzelne Zeile muss zunächst die Position der Spalte berechnet werden. Aus diesem Grund ist dringend zu empfehlen, VAR-

CHAR-Spalten ganz am Ende der Tabelle zu platzieren und nicht in WHERE-Bedingungen aufzunehmen.

15.1.1.4 Schlüsselspalten (Primary Key)

Schlüsselspalten werden regelmäßig für logische Verknüpfungen der Tabellen benutzt. Ebenso wird am häufigsten über Schlüsselspalten nach bestimmten Zeilen oder Zeilenbereichen gesucht werden.

Es empfiehlt sich daher, die Schlüsselspalten so kurz wie möglich zu halten. Man kann sagen, dass CHAR-Spalten (mit mehr als 5 Zeichen Länge) “langsamer” sind, als DECIMAL-Spalten, die wiederum (ab 5 Ziffern) “langsamer” sind, als INTEGER oder SMALLINT-Spalten.

15.2 Nicht-relationale Strukturen

DB2 greift auf die Daten in PAGES (4 Kilobytes große Blöcke) zu. In jeder Page können eine oder mehrere Zeilen einer Tabelle gespeichert sein. Daher kann die Anzahl der notwendigen IO-Operationen erheblich reduziert werden, wenn eine Tabelle entgegen dem relationalen Modell auf die Daten, nach denen regelmäßig gesucht werden muss, reduziert wird.

Innerhalb einer Entität kann es z.B. eine Reihe Felder geben, die immer wieder in Auswahllisten etc. auftauchen sollen. Andere Spalten sind hingegen vorhanden, die womöglich nicht immer gefüllt sein müssen und daher NULL-Values enthalten können. Diese würden trotzdem die Länge der Tabelle erhöhen und dadurch die Zahl der pro IO-Operation gelesenen Zeilen verringern.

Bei extrem performancekritischen Operationen kann es daher an gebracht sein, eine Tabelle entgegen der relationalen Konzeption “vertikal” zu teilen und alle immer gefüllten und regelmäßig in Listen auftauchenden Spalten in einer eigenen Tabelle zu platzieren. Die regelmäßig erstellten Auswahllisten würden somit schneller, weil mit weniger IO-Operationen erstellt. Die dazugehörigen Daten aus der anderen Tabelle müssten dann allerdings in einer zweiten Abfrage gelesen werden.

Diese Überlegungen sollten allerdings erst dann angestellt werden, wenn im konkreten Fall unter Produktionslast Probleme auftreten oder die Erfahrungen in vergleichbaren Fällen zu akzeptablen Ergebnisse geführt haben.

15.3 Index-Definitionen

Neben den Datentypen und gegebenenfalls gegen relationale Konzepte verstoßenden Designmöglichkeiten kann auch die Index-Definition erhebliche Einflüsse auf die Performance haben.

Jede Tabelle soll wenigstens einen Index haben, den man auch als Primary Key bezeichnet. Dieser, als "eindeutig" (unique) definierte Index dient zur Identifizierung der einzelnen Zeilen. Schon durch seine Definition kann sehr viel Performance verlorengehen.

Die meisten SQL-Systeme beurteilen die Nutzbarkeit eines Index' für Suchvorgänge vorwiegend anhand der "first key cardinality". Das ist die Anzahl der unter unterschiedlichen Indexeinträge bezogen auf die ersten Bytes – bei DB2 fünf Bytes - des Index.

Gegeben sei folgende Tabelle:

Spalte	*Datentyp*	*Länge (Bytes)*	*Postion im Index*
MANDANT	INTEGER	4	1
AKTENZEICHEN	INTEGER	4	2
LFDNR	SMALLINT	2	3
...	...	...	...

Die ersten vier Bytes jedes Indexeintrages sind mit dem Mandantenkennzeichen gefüllt, danach folgt das höherwertigste Byte des Aktenzeichens. Das Mandantenkennzeichen unterscheidet die Daten verschiedener Firmen, wenn sich mehrere Firmen im gleichen Rechenzentrum einmieten. In einem Rechenzentrum, das für nur eine Firma arbeitet, werden die ersten fünf Bytes mit hoher Wahrscheinlichkeit immer dieselben Werte haben. Die first key cardinality wäre (unabhängig von der tatsächlichen Zahl an Zeilen in der Tabelle) vermutlich 1.

Selbst in Rechenzentren mit mehreren Firmen als Kunden wird dieser vom Optimizer benutzte Wert nicht viel höher sein als die Anzahl der Firmen, die mit dem Verfahren arbeiten. Er wird also weit unter 10 liegen.

Vorteile kann es hier bringen, den Index unabhängig von der Reihenfolge der Spalten innerhalb der Tabelle zu definieren:

Spalte	*Datentyp*	*Länge (Bytes)*	*Postion im Index*
MANDANT	INTEGER	4	3
AKTENZEICHEN	INTEGER	4	1
LFDNR	SMALLINT	2	2
...	...	...	...

Der Index würde sich technisch nicht von der ersten Definition unterscheiden; dieselben Spalten sind als eindeutiger Primary Key definiert. Jedoch beinhalten die ersten vier Bytes schon das Aktenzeichen, das erfahrungsgemäß eine weitaus höhere Kardinalität aufweisen wird, als das Mandantenkennzeichen (in diesem Fall vermutlich Gesamtzahl Zeilen / Anzahl Mandanten).

Sofern die angeschlossenen Mandanten nicht auf eigenen Schlüsselreihen bestehen und einverstanden sind, dass die Schlüsselwerte über alle Mandanten verteilt werden, ist auch folgende Definition denkbar:

Spalte	*Datentyp*	*Länge (Bytes)*	*Postion im Index*
GKZ	INTEGER	4	-
AKTENZEICHEN	INTEGER	4	1
LFDNR	SMALLINT	2	2
...	...	...	...

In diesem Fall wäre die first key cardinality noch um einiges höher. Die notwendige WHERE-Bedingung auf das Mandantenkenzeichen ist in diesem Beispiel ebenso wie in den vorangegangenen Index-Versionen übrigens fast nicht zu optimieren (siehe auch unten). Lediglich in Ausnahmefällen kann der Optimizer überlistet werden.

15.3.1 Optimieren von Suchbedingungen durch Indizes

Neben dem Primärschlüssel können nahezu beliebige weitere Indizes definiert werden. Diese können wahlweise eindeutig (unique) sein oder auch Einträge mehrfach enthalten. Auch hier spielt die first key cardinality eine entscheidende Rolle bei der Nutzung des Index durch den Datenbankserver.

Insbesondere WHERE-Klauseln lassen sich optimieren, wenn ein Index definiert wurde. Allerdings sind auch hier ein paar Regeln zu beachten. Die WHERE-Bedingung wird beispielsweise von DB2 generell nur über den Index bzw. die Indizes aufgelöst, wenn diese eine ausreichende Kardinalität besitzen und in - Verbindung mit dem gesuchten Wertebereich - nur ein ausreichend kleiner Teil der Tabelle tatsächlich gesucht wird.

Ein Beispiel für einen unbrauchbaren Index liefert die bereits erwähnte Spalte Mandantenkennzeichen. In einem Rechenzentrum mit 3 Kunden wird ein Index über diese Spalte vermutlich nur 4 unterschiedliche Werte enthalten (die drei Kunden und ggf. eine Kennung für Schulungen).

Wenn jeder Kunde ca. 10.000 Zeilen in einer Tabelle hat, kommen wir auf rund 30.000 Zeilen und vier verschiedene Werte der Spalte MANDANT.

Unterstellt, dass DB2 in jeder Datenpage (die nur komplett gelesen werden können) 5 Zeilen der Tabelle abspeichern kann, muss man feststellen, dass beispielsweise DB2 und ORACLE statistisch gesehen in jeder Datenpage eine Zeile zu jedem Mandanten finden wird. Es müssten also sowieso alle Datenpages gelesen werden, wenn nach einem bestimmten Mandanten gesucht wird.

Ein gezielter Zugriff über den Index würde daher voraussichtlich nur zu zusätzlichen I/O-Operationen führen. Der SQL-Interpreter würde einen Index über die Spalte MANDANT daher ignorieren.

15.3.2 Indizes und Sortierfolgen

Auch zur Beschleunigung von ORDER-BY-Klauseln können Indizes benutzt werden. SQL-Systeme speicherndie Daten prinzipiell unsortiert ab, viele versuchen aber, sich beim Abspeichern annhernd an die Sortierfolge des CLUSTERING INDEX zu halten.

Der CLUSTERING INDEX ist entweder der erste Index, der zur Tabelle definiert wurde (also regelmäßig der Primary Key) oder der letzte Index. Soll ein anderer Index die CLUSTER-Folge angeben, kann das meist mit einer Option wie „CLUSTERING“ beim Anlegen des Index' erreicht werden.

Bei Reorganisationen wird immer die Reihenfolge des CLUSTERING INDEX wiederhergestellt. Bei Updates und Inserts in die Tabelle wird (wie gesagt) versucht, der Sortierfolge möglichst nahe zu kommen. Der Erfolg dieses Versuches wird über die CLUSTERRATIO (so heisst es bei DB2 jedenfalls) ausgedrückt. Diese steht für den prozentualen Anteil der Zeilen einer Tabelle, die in der Reihenfolge des CLUSTERING INDEX abgespeichert sind.

Wenn Sie über diesen Index sortieren wollen, so wird dieser (zum sog. LIST PREFETCH) nur dann benutzt, wenn die Clusterratio größer als 90% ist. Ansonsten muss die Tabelle nacher sowieso mittels SQLSORT sortiert werden, wodurch die I/O-Operationen auf den Index zum Sortieren überflüssig sind.

15.4 Überlisten des Optimizers

Das genannte Paradebeispiel nicht optimierbarer Spalten, der Mandantenschlüssel, kann unter bestimmten Voraussetzungen sehr wohl optimiert werden. Wenn über den Schlüssel der CLUSTERING INDEX definiert wurde, liegen die Zeilen überwiegend nach dem Mandanten sortiert vor.

Wenn nach dem Mandanten gesucht werden soll, kann der Zugriff über den Index die Zahl der zu lesenden Datenpages in obigem Beispiel auf ca. 1/3 der vorhandenen Daten reduzieren. Diesen Zusammenhang kann der Optimizer bislang nicht erkennen. Er wird anhand der geringen Kardinalität des Mandanten-Index’ annehmen, dass seine Nutzung unergiebig ist.

Diese statistischen Angaben über Tabelleninhalte werden nicht automatisch, sondern nur beim Lauf spezieller Utilities ermittelt. Abgespeichert werden sie im Datenkatalog. Dieser besteht aus Tabellen, die in der Regel nicht verändert werden dürfen.

Bestimmte Spalten, die sich auf den Optimizer beziehen, können jedoch in vielen Fällen sehr wohl modifiziert werden. Es besteht daher die Möglichkeit, nach dem Statistiklauf über die Tabelle die Spalten, die sich auf die Indexkardi-

nalität beziehen, mit der Kardinalität der gesamten Tabelle zu füllen. Der Optimizer wird daraufhin annehmen, dass der Index für jede Zeile der Tabelle einen eigenen Wert enthält. Der Index würde daraufhin in nahezu jeder WHERE-Bedingung benutzt, in der seine Spalten relevant sind.

VORSICHT:
Derartige Änderungen sind zwar in vielen Bereichen denkbar, um einzelne Abfragen erheblich zu beschleunigen, sie können jedoch unter dem Strich auf ein Gesamtverfahren bezogen zu erheblich längeren Antwortzeiten führen.

Derartige Modifikationen können wegen der notwendigen Produktionslast nur im Produktionssystem vorgenommen werden, weshalb sie sorgfältig zu bedenken und zu planen sind. Sie sollten erst dann in Betracht gezogen werden, wenn sich erweist, dass bestimmte Abfragen durch eine normalerweise nicht indizierbare Spalte unakzeptable Antwortzeiten erhalten.

Wenn die Modifikation kein spürbares Ergebnis bringt, ist sie wieder rückgängig zu machen.

15.5 Programmdesign

Auch das Design der Anwendungen kann zu Unterschieden in der Performance führen.

Es gibt mehrere Möglichkeiten, auf SQL-Datenbanken zuzugreifen.

Die verschiedenen Codierungsarten werden nun diskutiert.

15.5.1 Statisches, eingebettetes SQL

Bei dieser Variante sind regelmäßig die besten Zugriffszeiten erzielbar. Alle in der Applikationssprache komplett eingebetteten Statements zählen hierzu. Da der Precompiler schon die wichtigsten Prüfungen des Statements durchführt (Syntax) spart das im echten Betrieb Laufzeit. Bei DB2 kommt hinzu, dass die vorkompilierten Statements in Paketen (Database Request Modules oder auch Bindfiles) gespeichert werden, die erst in die Datenbank gebunden werden müssen. Zu diesem Zeitpunkt erledigt der Optimizer bereits den größten Teil seiner Arbeit. Man sagt, dass ein gutes SQL-Statement 90% seiner Laufzeit im Optimizer zubringt.

Dafür ist die Programmierung eben statisch. Das Statement steht fest codiert im Code und die Anwendung kann

höchstens Suchwerte oder Werte zum Einfügen in die Datenbank an das Statement übergeben. Andererseits hat diese Statik den Vorteil, dass die Statements von der Datenbank wiedererkennbar sind. In der Regel werden die precompilierten Statements heutzutage in einem Statement- oder Packagecache zwischengespeichert. Die Datenbank erkennt bereits benutzte Statements und kann den fertig definierten Zugriffsplan wiederbenutzen.

15.5.2 Dynamisches SQL

Dynamisches SQL kann auf mehrere Weisen codiert werden. Einerseits können dynamische Statements über die EXECUTE- und PREPARE-Funktionalität in jegliche Hochsprache eingebettet werden, für die ein Precompiler verfügbar ist. Andererseits gibt es einige Programmierumgebungen, in denen eine statische Codierung von SQL gar nicht möglich ist.

Zu letzteren zählen unter MVS beispielsweise CDB/REXX und vergleichbare Programme sowie SAS. Auf der Client/Server-Ebene gilt diese Einschränkung für Programme, die ODBC-Treiber zum Datenbankzugriff benutzen – also faktisch alle Windows-Programme und für die CLI-Programmierung. Auch PERL und PHP sowie Desktopdatenbanke wie Paradox oder Access arbeiten bei der Kommunikation mit SQL-Servern ausschließlich mit dynamischem SQL.

Ebenfalls müssen alle Programme, die interaktive Abfragen ermöglichen (SPUFI, QMF, SAS, Paradox, Visualizer, Seagate Info, InfoCube...) zwingend die eingetippten Abfragen in dynamisches SQL umwandeln, damit sie ausgeführt werden können.

Da während der PREPARE-Phase das Precompilieren und “Binden” inclusive Lauf des Optimizers zur Laufzeit ausgeführt wird, ist dynamisches SQL (neben dem höheren Codierungsaufwand in Sprachen, für die Precompiler verfügbar sind) vom Grunde her langsamer, als statisches SQL.

Außerdem neigt man zur Schlamperei: Die Statements werden mit den üblicherweise recht komplexen Stringoperationen moderner Programmierumgebungen zur Laufzeit zusammen gebaut; Suchbegriffe etc. finden sich im Text. In der Regel könnte auch hier mit Platzhaltern gearbeitet werden: Dieselbe Suchoperation wird im Laufe eines Tages sicherlich mehr als einmal ausgeführt. Werden die Suchbegriffe jedesmal in den Klartext des Statements eingebaut,

hätte der Optimizer viel zu tun, um sich wiederholende Statements zu erkennen und den fertig optimierten Zugriffsplan aus dem Cache zu holen.

Allerdings kann es notwendig sein, in einer WHERE-Bedingung eine Vielzahl von Suchbegriffen vorzusehen. Bei statischem SQL hätte der Optimizer eine nahezu unlösbare Aufgabe, da der tatsächliche Zugriffs Zugriffspfad von der Füllung der übergebenen Hostvariablen zur Laufzeit des Programmes abhinge. Zum Zeitpunkt des Codierens noch nicht abgesehen werden, welche Suchbedingungen tatsächlich genutzt werden. Es würde typischerweise für jede möglicherweise relevante Spalte ein Suchwert vorgegeben – bei Verwendung von Platzhaltern oder gar statischem SQL müsste ein sehr komplexes Statement codiert werden. Der Optimizer hätte kaum eine Chance, die aktuell gar nicht gewünschten Suchbedingungen zu erkennen und zu ignorieren.

Bei dynamischem SQL ist es möglich, die Suchbegriffe zur Laufzeit als Strings in die WHERE-Bedingung einzubinden. Beim PREPARE kann der Optimizer auf Basis der tatsächlichen vorhandenen Suchbedingung den Zugriffspfad bestimmen.

Insbesondere, wenn der Anwender eine zentrale Maske zur Abwicklung vieler unterschiedlicher Suchfunktionen benötigt, die alle in einer identisch strukturierten Liste münden, ist eine Lösung über dynamisches SQL performancegünstiger, als eine statisch codierte Lösung.

15.6 Codierung der Statements

Wie üblich kann auch bei SQL die genaue Formulierung der Statements über gute oder schlechte Performance entscheiden.

Generell gilt, dass bei WHERE-Klauseln eine bestimmte Reihenfolge eingehalten werden sollte. Weiterhin sollten bestimmte Codierungsmöglichkeiten anderen gegenüber vorgezogen werden.

- Die Bedingungen, welche die geringste Ergebnismenge liefern, sollten zuvorderst stehen
- Formulierungen mit BETWEEN sind gleichwertigen AND-Verknüpfungen vorzuziehen.
- Formulierungen mit IN (...) sind gleichwertigen OR-Verknüpfungen vorzuziehen.

Ein Verstoß gegen diese Richtlinie führt nicht unbedingt zu anderen Laufzeiten der Abfragen, da die modernen Optimizer nicht mehr zwingend auf solche Hilfestellungen angewiesen sind. Sie erkennen z.B. meistens, dass eine Spalte mit OR-Verknüpfungen auf mehrere unter unterschiedliche Werte abgefragt wird. Die IN (...)-Formulierung vermeidet jedoch, dass der Optimizer dies "übersieht". Auch einem menschlichen Leser des Programmcodes wird der Zusammenhang so auf den ersten Blick erkennbar.

Garantieren kann man Performancevorteile folgender Richtlinien für WHERE-Bedingungen:

- Suchbedingungen sollen auf einer Seite des Vergleichsoperators eine Tabellenspalte, auf der anderen Seite eine andere Spalte, eine Host-Variable oder eine Konstante enthalten. Rechenoperationen in Suchbedingungen führen zu erheblich längeren Laufzeiten.
- Bei Vergleichen mit numerischen Spalten ist unbedingt darauf zu achten, dass die Vergleichswerte bzw. Hostvariablen denselben Datentyp haben, wie die Spalte selbst. SQL sieht in vielen Fällen Typkonvertierungen zwischen numerischen Werten vor, die sich genauso nachteilig wie Rechenoperationen auswirken.
- Auf LIKE-Abfragen mit Hostvariablen ist in statischem SQL soweit wie möglich zu verzichten, da diese sich nicht optimieren lassen. Alternativ sollte das Statement dynamisch codiert werden und den Suchbegriff als Stringkonstante enthalten.
- Wenn LIKE-Abfragen notwendig sind, lassen diese sich nur dann optimieren, wenn der Suchstring als Konstante enthalten ist, ein nutzbarer Index über der betroffene Spalte liegt und die ersten drei bis fünf Bytes nicht maskiert sind.

15.6.1 Bewertung von Statements mittels EXPLAIN

Wichtige Hinweise auf die Ursachen von Performanceproblemen bietet bei DB2 das EXPLAIN-Utility. Es gibt mehrere Möglichkeiten, dieses Utility zu nutzen.

Die Auswertung der EXPLAIN-Ausgaben ist wegen der komplexen Zusammenhänge und der Verschlüsselung nur möglich, wenn ein entsprechender Lehrgang besucht wurde. Das notwendige Wissen ist in der DB2-Administration vorhanden.

15.6.1.1 Explain beim Binden

Wird beim Binden eines Plans oder Packages EXPLAIN(YES) angegeben, so erzeugt der BIND-Lauf zu jedem Statement eine Analyse der “Handgriffe”, die der Optimizer für die Ausführung des Statements als beste Lösung ermittelt hat. Diese Ausgaben sind sehr umfangreich und in den meisten Fällen nicht notwendig.

Es ist jedoch nicht schädlich, bei einer Produktionsübernahme die Pläne und Packages einmal mit EXPLAIN(YES) zu binden, da bei auftretenden Problemen anhand der schon vorhandenen Ausgaben des EXPLAIN Aussagen zur Ursache gemacht werden können.

15.6.1.2 EXPLAIN über interaktives SQL

Wenn nur einzelne Statements bewertet werden sollen, kann alternativ auch das SQL-Statement EXPLAIN zur Analyse benutzt werden. EXPLAIN wird ein beliebiges SQL-Statement übergeben, woraufhin der Optimizer auf Basis der aktuellen RUNSTATS-Werte im Katalog den Zugriffspfad ermittelt. Das Ergebnis wird nicht ausgedruckt sondern in eine spezielle Tabelle gestellt (PLAN_TABLE unter der Prefix des Anwenders, der das EXPLAIN ausgeführt hat).

Diese Tabelle muss erst explizit angelegt werden und variiert von Version zu Version des DB2-Systems. Wenden Sie sich daher bitte an zu mehreren Statements abgespeichert und verglichen werden. Auch hier ist wegen der Codierung der einzelnen Zugriffe und der Bedeutung der Zugriffsarten prinzipiell ein Lehrgang notwendig.

1 Anhang A: Tabellendefinitionen

Die Definitionen der Test-Datenbank finden Sie in diesem Anhang abgedruckt. Die Syntax bezieht sich auf DB2/MVS

```
-----------------------------------------------------
-----------
-- CREATEs und ALTERS für CD-Datenbank SQL-Kurs
18/43
--
-----------------------------------------------------
-----------

   CREATE TABLE AUFNAHME
    (CD_NR                  SMALLINT
                                        NOT NULL
,
     M_NR                   SMALLINT
                                        NOT NULL
,
     I_NR                   SMALLINT
                                        NOT NULL
,
     TRACK                  SMALLINT
                                        NOT NULL
,
     JAHR                   SMALLINT
                                        NOT NULL
,
     LAENGE                 SMALLINT
                                        NOT NULL
)
         IN DI43KURS.SI43KURS
;
    CREATE        UNIQUE                INDEX
        AUFNAHME_I                   ON AUFNAHME
                    (CD_NR                 ASC ,
                     M_NR                  ASC ,
                     I_NR                  ASC )
               USING  STOGROUP GTDIB501
               PRIQTY      40  SECQTY      40
ERASE NO
               FREEPAGE   9  PCTFREE  10
               SUBPAGES   4  BUFFERPOOL BP0
               CLOSE NO
;
   CREATE TABLE CD
    (CD_NR                  SMALLINT
                                        NOT NULL
,
     CD_TITEL               CHAR    (60)
                                        NOT NULL
,
     BESTNR                 CHAR    (60)
                                        NOT NULL
,
     VERLAG                 CHAR    (60)
```

```
                                        NOT NULL
,
     INTERPRET                SMALLINT
,
     KOMPONIST                SMALLINT
,
     TITELZAHL                SMALLINT
                                        NOT NULL
,
     MUSIKART                 CHAR    (60)
                                        NOT NULL
,
     PREIS                    DECIMAL (5, 2)
                                        NOT NULL
)
         IN DI43KURS.SI43KURS
;
    CREATE       UNIQUE                INDEX
        CD_I                      ON CD
                    (CD_NR                    ASC )
               USING  STOGROUP GTDIB501
               PRIQTY      40  SECQTY      40
ERASE NO
               FREEPAGE   9  PCTFREE  10
               SUBPAGES   4  BUFFERPOOL BP0
               CLOSE NO
;
   CREATE TABLE INTERPRET
    (I_NR                     SMALLINT
                                        NOT NULL
,
     I_NAME                   CHAR    (60)
                                        NOT NULL
,
     I_GEBDAT                 DATE
,
     I_TODESTAG               DATE
,
     I_ADRESSE                CHAR    (60)
,
     I_FANCLUB                CHAR    (60)
,
     INSTRUMENT               CHAR    (60)
                                        NOT NULL
)
         IN DI43KURS.SI43KURS
;
    CREATE       UNIQUE                INDEX
        INTERPRET_I               ON
INTERPRET
                    (I_NR                     ASC )
               USING  STOGROUP GTDIB501
               PRIQTY      40  SECQTY      40
ERASE NO
               FREEPAGE   9  PCTFREE  10
               SUBPAGES   4  BUFFERPOOL BP0
               CLOSE NO
;
   CREATE TABLE KOMPONIST
    (K_NR                     SMALLINT
                                        NOT NULL
,
```

```
      K_NAME                    CHAR     (60)
                                            NOT NULL
,
      K_GEBDAT                  DATE
,
      K_TODESTAG                DATE
)
          IN DI43KURS.SI43KURS
;
    CREATE        UNIQUE                INDEX
        KOMONIST_I                    ON
KOMPONIST
                    (K_NR                 ASC )
               USING  STOGROUP GTDIB501
               PRIQTY       40  SECQTY      40
ERASE NO
               FREEPAGE    9  PCTFREE  10
               SUBPAGES    4  BUFFERPOOL BP0
               CLOSE NO
;
   CREATE TABLE MUSIKSTUECK
    (M_NR                       SMALLINT
                                            NOT NULL
,
      MUSIKTITEL                CHAR     (60)
                                            NOT NULL
,
      KOMPONIST                 SMALLINT
                                            NOT NULL
,
      TEXTER                    SMALLINT
,
      LAENGE                    SMALLINT
                                            NOT NULL
,
      MUSIKART                  CHAR     (60)
                                            NOT NULL
,
      ERSCHEIN_DATUM            DATE
                                            NOT NULL
)
          IN DI43KURS.SI43KURS
;
    CREATE        UNIQUE                INDEX
        MUSIKSTUECK_I                 ON
MUSIKSTUECK
                    (M_NR                 ASC )
               USING  STOGROUP GTDIB501
               PRIQTY       40  SECQTY      40
ERASE NO
               FREEPAGE    9  PCTFREE  10
               SUBPAGES    4  BUFFERPOOL BP0
               CLOSE NO
;
   CREATE TABLE TEXTER
    (T_NR                       SMALLINT
                                            NOT NULL
,
      T_NAME                    CHAR     (60)
                                            NOT NULL
,
```

```
     T_GEBDAT               DATE
,
     T_TODESTAG             DATE
,
     SCHRIFTSTELLER_MM      CHAR    (1)
                                        NOT NULL
WITH DEFAULT)
         IN DI43KURS.SI43KURS
;
    CREATE         UNIQUE                 INDEX
        TEXTER_I                    ON TEXTER
                    (T_NR                 ASC )
               USING  STOGROUP GTDIB501
               PRIQTY       40  SECQTY      40
ERASE NO
               FREEPAGE   9  PCTFREE  10
               SUBPAGES   4  BUFFERPOOL BP0
               CLOSE NO
;
    ALTER TABLE AUFNAHME
      PRIMARY KEY (CD_NR,M_NR,I_NR)
;
    ALTER TABLE CD
      PRIMARY KEY (CD_NR)
;
    ALTER TABLE INTERPRET
      PRIMARY KEY (I_NR)
;
    ALTER TABLE KOMPONIST
      PRIMARY KEY (K_NR)
;
    ALTER TABLE MUSIKSTUECK
      PRIMARY KEY (M_NR)
;
    ALTER TABLE TEXTER
      PRIMARY KEY (T_NR)
;
    ALTER TABLE AUFNAHME
      FOREIGN KEY CD        (CD_NR)
    REFERENCES CD          ON DELETE RESTRICT
;
    ALTER TABLE AUFNAHME
      FOREIGN KEY INTERPRE  (I_NR)
    REFERENCES INTERPRET          ON DELETE
RESTRICT
;
    ALTER TABLE AUFNAHME
      FOREIGN KEY MUSIKSTU  (M_NR)
    REFERENCES MUSIKSTUECK          ON DELETE
RESTRICT
;
    ALTER TABLE MUSIKSTUECK
      FOREIGN KEY KOMPONIS  (KOMPONIST)
    REFERENCES KOMPONIST          ON DELETE
RESTRICT
;
```