Prof. Dr. Günter Brackly
FB Informatik/Mikrosystemtechnik
Fachhochschule Kaiserslautern

Software-Engineering und Datenbanken

Skript zur Vorlesung

Günter Brackly

Inhalt: Seite

Einleitung

Dieses Skript ist entstanden aus einer Vorlesung ‚Software-Engineering und Datenbanken, deren Inhalt also Theorie und Praxis der Entwicklung relationaler Datenbank-Applikationen umfaßt.
Die Vorlesung befaßt sich also mit mehreren verschiedenen **praktischen** Aspekten der Applikationsentwicklung:

- Die Erfassung und Modellierung einer vorgegebenen Ausgangssituation in einem ersten Modell

- Die Überführung dieser ersten Modellierung in ein relationales Datenbankdesign

- Die Implementierung des Designs in ein konkretes Datenbanksystem

- Die Konzeption und Implementierung der Applikation, die auf dieser konkreten relationalen Datenbank aufsetzt.

Neben diesen praktischen Aspekten müssen natürlich auch die dazu notwendigen **theoretischen** Hintergründe bereitgestellt werden:

- Das Instrumentarium eines semantischen Datenmodells am Beispiel Entity-Relationship-Modell

- Theorie relationaler Datenbanken

- Anfragesprachen relationaler Datenbanksysteme am Beispiel SQL inklusive prozeduraler Erweiterungen zur Implementierung von stored procedures und Triggern

- Möglichkeiten der Einbettung von SQL-statements in 3GL-Code am Beispiel von JDBC

Zusätzlich zu diesen unbedingt notwendigen Informationen und praktischen Handhabungen im Zusammenhang mit der Erstellung relationaler Datenbankapplikationen ist es unbedingt auch erforderlich, grunsätzliches Wissen über die **interne Organisation** eines modernen Datenbanksystems bereitzustellen. Dies umfaßt vor allem die Bereiche Transaktionskonzept und die Kontrolle konkurrierender Zugriffe in einer multi-user-Umgebung, aber auch die Frage nach dem Management einzelner Komponenten eines Datenbanksystems, Backup- und Recovery-Strategien und Tuning-Möglichkeiten.

Diese Themenbereiche werden in der Vorlesung und damit auch in diesem Skript angesprochen und an vielen Beispielen demonstriert und erklärt.

Ausgangspunkt ist also die Notwendigkeit, Applikationen zu entwickeln, deren anfallende Daten die Beendigung der Applikation überleben müssen, also auch dauerhaft zur Verfügung stehen müssen. Dies definiert ein Einsatzgebiet moderner Datenbanksysteme.

Zielsetzungen für den Einsatz eines Datenbanksystems sind allgemein:

- Persistente Speicherung von Daten
- Speichermöglichkeiten unabhängig vom Datentyp (Zahlen, Zeichen, Zeichenketten, Fließtext, Bild, Audio, Video, Raster)
- Beliebige Kombinierbarkeit von Daten in verschiedenen digitalen Medienarten
- Speicherung mit Garantie des Erhalts der Semantik (Konsistenz)
- Sicheres und verlustfreies Speichern selbst bzgl. Systemabstürze oder Mediafehler
- Paralleler Zugriff beliebig vieler Nutzer auf gleiche Datenbestände
- Sicherheit der Daten vor unberechtigtem Zugriff
- Performantes Arbeiten mit gespeicherten Daten
- Exaktes Retrieval gespeicherter Information auch in neuen Sinnzusammenhängen

Diese Liste möglicher Zielsetzungen ließe sich bestimmt noch weiterführen, aber hier wird schon deutlich, daß diese Zielsetzungen nur umgesetzt werden können, wenn solche Datenbanksysteme über ausgeklügelte Mechanismen und Konzepte verfügen, die man in anderen Programmsystemn so nicht wiederfindet!

Die Vorlesung (das Skript) sind wie folgt aufgebaut:

Kapitel I (Theorie relationaler Datenbanksysteme) beschreibt zunächst die Konstrukte und Konzepte des Entity-Relationship-Modells (ERM) und anschließend des relationalen Datenbankmodells (RDM). Anschließend wird eine Transformationsvorschrift angegeben, um ein ERM in ein RDM zu überführen. Nach einem ausführlichen Beispiel (KAPV) geht es darum, das erreichte Datenbankdesign auch für den laufenden Betrieb sicher und konsistenzbewahrend zu gestalten. Dazu sind verschiedene Normalisierungsschritte notwendig, die ausführlich besprochen werden. Den Abschluß dieses Kapitels bildet die Beschreibung der Relationenalgebra, also der möglichen Operationen, die auf den Relationen ausgeführt werden können.

Kapitel II ist eine geraffte Beschreibung des derzeit gültigen SQL-Standards bzgl. klassischer Datenbankoperationen (also z.B. ohne die zur Zeit diskutierten Multimedia-Erweiterungen von SQL hinsichtlich Retrieval-Fähigkeiten). Wir unterscheiden hier data definition language statements zur Erzeugung, Veränderung und Löschung von Datenbankobjekten (Relationen, etc...) und data manipulation language statements zur Erzeugung, Veränderung, Löschung und Anzeige von Datensätzen. Hier wird immer auch konkret Bezug genommen auf das am Fachbereich eingesetzte datenbanksystem der Firma ORACLE.

Kapitel III befaßt sich mit der Datenbankorganisation und beschreibt die Themenbereiche Komponenten eines Datenbanksystems (die spätestens bei den Tuning-Überlegungen wieder eine wichtige Rolle spielen), Transaktionskonzept mit den ACID-Bedingungen, die Probleme

in multiuser-Umgebungen an den klassischen Beispielen *lost update*, *dirty read* und *unrepeatable read*, die Lösung durch den Einsatz von Sperren, das dadurch neu entstehende Problem der deadlocks und schließlich die allgemeine Lösung durch die Serialisierbarkeit der Transaktionen durch das 2-Phasen-Sperrprotokoll. Danach werden noch kurz Backup-Strategien hinsichtlich der recovery-Möglichkeiten, Tuning-Möglichkeiten und die theoretischen und praktischen Konsequenzen beim Einsatz moderner Datenbanksysteme in Client-Server-Umgebungen diskutiert. Den Abschluß dieses Kapitels bildet ein Exkurs über views also Möglichkeiten in relationalen Datenbanksystemen, logische datenunabhängigkeit und Zugriffssicherheit zu schaffen.

Kapitel IV schließlich befaßt sich wieder mit der praktischen Umsetzung von Datenbankapplikationen. Zunächst wird die prozedurale Erweiterung von SQL beschrieben, wie sie der Standard vorsieht und wie sie bei OARCLE umgesetzt ist in PL/SQL. Damit sind wir dann in der Lage, stored procedures und Datenbanktrigger zu schreiben, um Anwendungsfunktionalität unter der Verantwortung des Datenbanksystems zu realisieren oder zusätzliche konsistenzsichernde Funktionalitäten zu implementieren. Im zweiten Teil wird die Einbettung von SQL-statements in moderne Programmiersprachen via Call Level Interface-API's besprochen, hier am Beispiel von JDBC.

Dieses Skript ist gedacht als Leitfaden zur Vorlesung und Hilfe zu den praktischen Übungen, es mangelt unter Umständen an einigen Stellen an ausführlicheren Komentaren!

Zweibrücken im Juli 2000

Günter Brackly

Kapitel I Theorie relationaler Datenbanken

I.1 Semantische Modellierung: Das Entity-Relationship-Modell ERM

Ziel des ER-Modell ist:
eine vorgegebene Ausgangssituation zu strukturieren und die wesentlichen Elemente abstrakt zu beschreiben

Als Strukturkonzept und Bausteine stehen dabei im wesentlichen zur Verfügung:
Entities und Relationships

I.1.1 Entities

In der realen Welt sind wohlunterscheidbare Objekte gegeben: bestimmte Personen, Kunden, Autos, Städte, usw. Diese sind die Entities im ER-Modell.

Entities besitzen Eigenschaften (Farbe, Länge, Name, usw.).
Die Eigenschaften haben bestimmte Wertebereiche (Domänen), die die Menge der zugelassenen Werte für die Eigenschaften definieren.
Entities werden zu Mengen von Entities eines bestimmten Typs zusammengefaßt, den Entity-Sets!

Beispiel:
Entity-Set: Alle Kunden eines Unternehmens

Name: KUNDE

Eigenschaften	**Wertebereich**
Name	Charstring der Länge 40
Vorname	Charstring der Länge 40
Geburtsdatum	Datum
bish. Auftragsvol.	10-stellige Zahl mit zwei Nachkommastellen
Anschrift	?
Hobbies	Charstring der Länge 50

Ein konkretes Entity erhält man, indem man jedem Attribut des E-Sets einen Wert zuordnet.

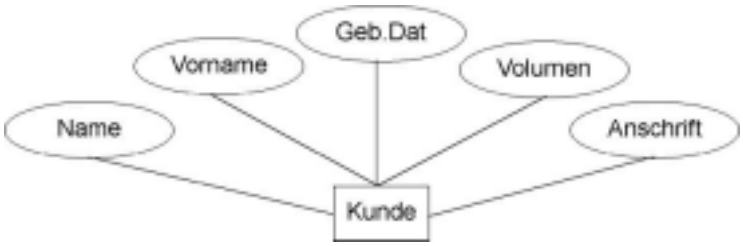
$e_{1=}$ (Müller; Horst; 11.07.65; 1.500,42; Weg2;12355 Teststadt; {Schwimmen, Reiten, Fußball})

Definition:
Die Menge X aller Attribute eines E-Sets heißt
Entity-Format

Ein Entity-Set mit seinem Format kann man graphisch darstellen:

Set: Rechteck
Attribut: Kreis, mit dem Rechteck verbunden

Beispiel:



Problem:

1) Anschrift besteht selbst aus :

Straße	Charakterstring 40
PLZ	Charakterstring 05
Ort	Charakterstring 40

Zusammengesetzte Attribute werden dargestellt, indem die Komponenten extra mit Kreisen ausgewiesen und mit den Attributen verbunden werden:

2) Hobbies ist ein Attribut, das eigentlich mehrwertig ist; ein Kunde kann viele verschiedene Hobbies haben.
Solche Attribute werden durch Doppelkreis gekennzeichnet:

Also erhalten wir für das E-Set Kunde die Darstellung:

Wir hatten alle Entities als wohlunterscheidbar vorausgesetzt, d. h. alle Attribute zusammen identifizieren eindeutig ein Entity.
Meist kann man jedoch spezielle Teilmengen als sogenannte Schlüssel auszeichnen:

Definition:

Ein Schlüssel K ist eine Teilmenge der Menge aller Attribute eines Entity-Sets, die jedes Entity eindeutig identifiziert und die minimal ist, d. h. es gibt kein $K' \subseteq K$ mit der gleichen Eigenschaft!
Gibt es mehrere Schlüssel für ein Entity-Set, muß einer ausgezeichnet werden. Diesen nennt man den Primärschlüssel.

Die Attribute eines Schlüssels müssen einwertig sein (sonst identifizieren sie nicht eindeutig!).

Formalisierungen:

Definition:

Ein Entity-Typ hat die Form E = (X, K);
er besteht aus dem Namen E, dem Format X und dem Primärschlüssel K.

Ein Entity-Format beschreibt also die Struktur eines Entity-Sets.

Notationen für die Elemente des Formats X:

- Einwertige Attribute: A
- Mehrwertige Attribute: {A}
- Zusammengesetzte Attribute: A (B_1, ... , B_k)

Beispiel:

Nehmen wir als Primärschlüssel des Entity-Sets Kunde die Kundennummer, so erhalten wir die Beschreibung als Entity-Typ:

KUNDE: = ({Kdnr, Name, Vorname, Gebdat, Volumen, {Anschrift (Straße, PLZ, Ort)}, {Hobbies}}, {Kdnr})

Für die Wertebereiche der einzelnen Attribute führen wir folgende Notation ein:

Definition:

Sei E = (X,K) ein Entity-Typ; $X = \{A_1, ... , A_n\}$
Ist $A \in X$ ein einfaches, nicht zusammengesetztes Attribut, so bezeichnet W(A) die Wertemenge von A.
Dann ist für jedes $A \in X$:

$$\text{dom}(A) := \begin{cases} W(A) & \text{falls A einwertig,} \\ \text{Pot}(W(A)) & \text{falls A mehrwertig,} \\ W(B_1) \times ... \times W(B_k) & \text{falls A aus dem einwertigen } B_1, ..., B_k \text{ zusammengesetzt ist} \end{cases}$$

Beispiel:

dom (Name)= {Charakterstring der Länge ≤ 40}
dom (Hobbies)= Pot ({Charakterstring der Länge ≤ 50})
dom (Anschrift)= Pot ({Char ≤ 40} $\times$ {Char ≤ 5} $\times$ {Char ≤ 40})

Für die formale Darstellung des Formats X eines Entity-Typs sind zwei Arten denkbar:
Als Menge oder als Folge:

Als Menge: $\Rightarrow$ Reihenfolge der Attribute ist egal, spielt keine Rolle
Als Folge: $\Rightarrow$ Reihenfolge ist genau definiert!

Wir machen es mit Mengendarstellung (im Hinblick auf das Relationale Modell, zu dem Kompatibilität bestehen soll!).

Also: $X = \{A_1, ..., A_n\}$, A_i Attribute

Entities kann man auch formal definieren:
Entities sind konkrete Ausprägungen der Werte eines Formats, d. h.:

Definition:

Ist $X = \{A_1, ..., A_n\}$ eine Menge von Attributen, so ist ein Entity e eine injektive Abbildung.

$$e = \{A_1, ..., A_n\} \rightarrow \bigcup_{i=1}^{n} dom(A_i)$$

für die gilt:
$(\forall\ 1 \leq i \leq n)\ (e\ (A_i) \in dom\ (A_i))$

Erläuterung:

Wieso diese Darstellung als Abbildung?
eine Abbildung ist bestimmt durch die Bilder des Definitionsbereiches, also durch die Wertemenge, d. h. eben hier ein Entity!
injektiv: Gleichheit von Entities nur bei Gleichheit aller Werte:
$(f(x) = f(y) \Rightarrow x = y)$

Nach der Definition von Entities (als injektive Abbildung) und Entity-Typen (mit Format X und Schlüssel K) können wir jetzt auch formal ein Entity-Set definieren:

Definition:

Ein Entity-Set E^s ist eine beliebige Teilmenge der Menge aller möglichen Entities über einem bestimmten Format, die den Primärschlüssel erfüllen:
Erfüllen heißt:

$$(\forall\ e_1, e_2 \in E^s)\ (e_1\ [K] = e_2\ [K] \Rightarrow e_1 = e_2)$$

dabei bedeutet die Notation e [K] die Einschränkung des Entities e auf die Attributmenge K, d. h. alle Attribute aus X \ K werden vernachlässigt.

im Beispiel:

ist Kunde = (X, K) der Entity-Typ Kunde mit dem
Format X = {Kdnr, Name, Vorname, Gebdat, {Anschrift (Straße, PLZ, Ort)}, {Hobbies}}
und e ein bestimmter Kunde:
e = (12345, Maier, Horst, 1.1.1967, {Wiesenweg 7, 55555, Teststadt}, {Schwimmen, Tennis})

Ist Y = {kdnr},
so ist e[Y] = 12345
Ist Y = {Name, Vorname}
so ist e [K] = e [{Name, Vorname}] = {Meier, Horst}

Zusammenfassung:

Die realen Objekte der Umwelt werden im Modell beschrieben durch die Konzepte:

Entities	als injektive Abbildung in die Wertebereiche der Attribute
Entity-Typ	Struktur der Entities in Form von Attributen und Schlüsselangabe
Entity-Set	Zusammenfassung von Entities über einem gemeinsamen Format und mit einem gemeinsamen Schlüssel

Graphisch dargestellt werden die realen Objekte im Modell durch Entity-Sets bzw. Entity-Typen als Rechtecke, die Attribute als Kreise, verbunden mit dem Rechteck über ungerichtete Kanten; Schlüsselattribute werden unterstrichen.

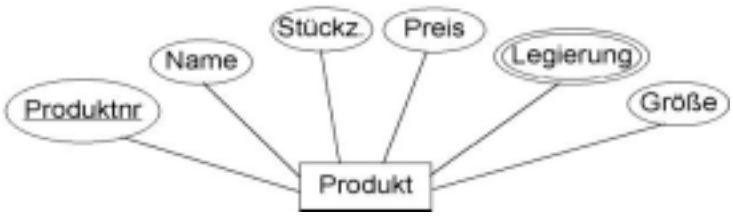
Beispiel:

Reale Objekte vom Typ Produkte haben folgende relevanten Attribute:
Produktnr, Name, Stückzahl, Preis, Legierung, Größe

Entity-Typ:

PRODUKT= ({Produktnr, Name Stückzahl, Preis, {Legierung}, Größe}, {Produktnr})

graphische Darstellung:



Wir wissen, daß reale Objekte zueinander in Beziehung treten: Leser entleihen Bücher, Studenten besuchen Vorlesungen, Professoren prüfen Studenten bzgl. einer Vorlesung.
Dies definiert die zweite zentrale Kategorie von Modellierungskonstrukten im Entity-Relationship-Modell:

I.1.2 Relationships:

Relationships setzen bestimmte Entity-Sets zueinander in Beziehung:

entleihen: die Entity-Sets LESER, BUCH
bestellen: die Entity-Sets KUNDE, PRODUKT
prüfen: die Entities STUDENT, PROFESSOR, VORLESUNG

zu beobachten ist:

1)Relationships können beliebig viele Entity-Sets in Beziehung setzen (meist nur zwei, aber eben auch mehr, wie z. B. prüfen)

2)Relationships können auch Attribute haben:
entleihen: z.B. Rückgabedatum

Wie bei den Entities können die Attribute einwertig, mehrwertig oder zusammengesetzt sein.

Wie bei den Entities definieren wir:
Relationship-Typ
Relationship
Relationship-Set

Definition:

1)Ein Relationship-Typ hat die Form
R = (Ent, Y).
R ist der Name des Typs
Ent ist die Menge der beteiligten Entity-Typen
Y eine (eventuell leere) Menge von Attributen

2)Sei Ent = $\{E_1, ..., E_K\}$, $E_i = (X_i, K_i)$ Entity-Typ
E_i^s zugehöriges Entity-Set
Sei Y $\{B_1, ..., B_n\}$, B_i Attribute

Eine Relationship r ist eine konkrete Ausprägung von Ent und Y (wie bei den Entities):

$r \in E_1^s \times ... \times E_k^s \times \text{dom}(B_1) \times .... \times \text{dom}(B_n)$
d.h.
$r = (e_1, ..., e_k, b_1, ..., b_n)$ e_i Entities,
b_j Werte aus der (B_j)

3)Ein Relationship-Set R^s ist eine Menge von Relationships:
$R_s \subseteq E_1^s \times ... \times E_k^s \times \text{dom}(B_1) \times ... \times \text{dom}(B_n)$

Beispiel:

Sei E_1 STUDENT = (X_1, K_1)
E_2 PROFESSOR = (X_3, K_2)
E_3 VORLESUNG = $(X_3, K3)$
Y {Datum, PrüfungsNr}

⇒

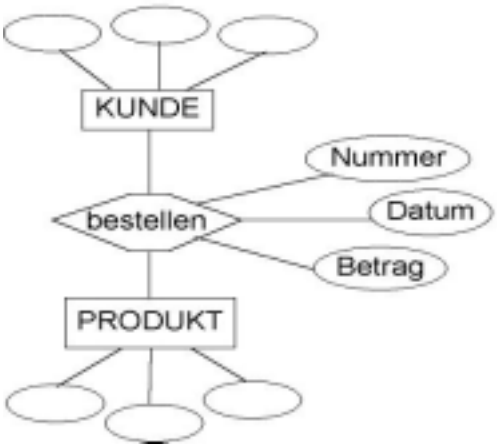
PRÜFUNG = ({STUDENT, PROFESSOR, VORLESUNG}, {Datum, PrüfungsNr})

ist der 3-stellige Relationship-Typ Prüfung!

Auch Relationship-Typen werden graphisch dargestellt:

- Relationship-Typ als Raute mit dem Namen des Typs
- Die Raute wird durch Kanten mit den beteiligten Entity-Typen verbunden
- Attribute wie bei den Entity-Typen durch Kreise, verbunden mit der Raute

Die graphische Darstellung der Entity-Typen mit dem Relationship-Typen heißt Entity-Relationship-Diagramm.
Beispiel:



Zusätzlich ist aus der Realität bekannt, welches Entity mit wieviel Entities des beteiligten Relationship-Typs in Beziehung stehen kann und umgekehrt:

Dies definiert die Kardinalität der Relationship:

ein Kunde-Typ kann viele Produkte bestellen zu einem Zeitpunkt,
ein Produkt-Typ kann zu einem bestimmten Zeitpunkt von vielen Kunden bestellt werden
man sagt: der Relationship-Typ hat die Kardinalität n : m ;
und man schreibt die Komplexität ins Diagram.

Es gibt z. Beispiel die Kardinalitäten

1 : n oder 0 : n,
n : m
1 : 1 oder 0 : 1

Zu erwähnen ist noch eine spezielle Relationship:
die sogenannte IS-A-Beziehung (is a kind of)

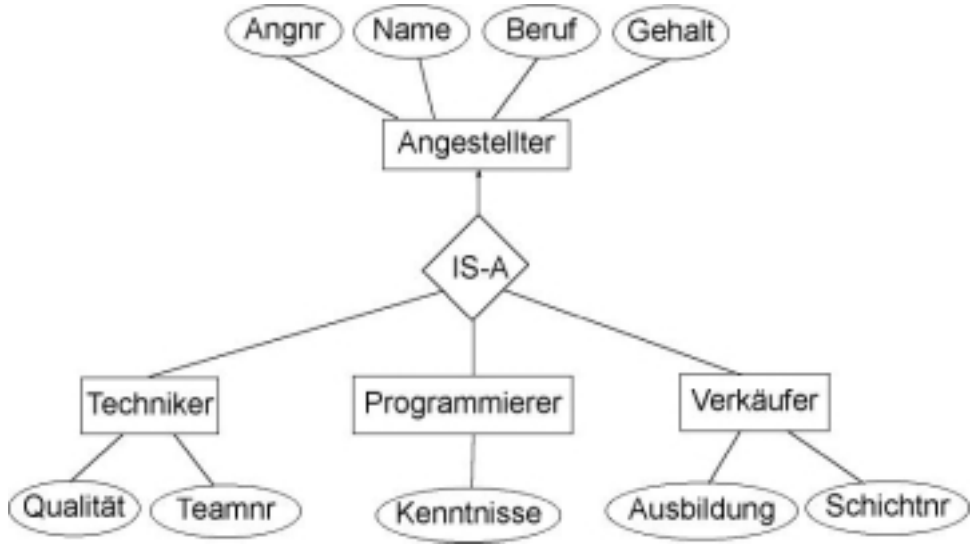
Beispiel:

Eine Firma benutzt den Entity-Typ Angestellter:
Angestellter = ({Angnr, Name, Beruf, Gehalt}, {Angnr})
Angestellte werden aber weiter spezifiziert und man möchte unterscheiden, ob es Techniker, Programmierer oder Verkäufer sind.

Techniker haben spezielle Qualifikationen, die erfasst werden sollen, gehören zu bestimmten Teams;
Programmierer haben spezielle Kenntnisse, die erfasst werden
Verkäufer haben Fachausbildung, Schichtdienste, usw.
also es existieren auch die E-Typen
Techniker = ({Angnr, Qualifikation, Teamnr}, {Angnr})
Programmierer = ({Angnr, Kenntnisse}, {Angnr})
Verkäufer = ({Angr, Ausbildung, Schichtnr}, {Angnr})

Alle drei Spezialisierungen haben die Eigenschaft, daß sie insbesondere Angestellte sind, also auch alle Attribute dieses Entity-Typen haben müssen, zusätzlich zu ihren eigenen. Bzgl. der Entity-Sets und konkreter Entities muß es so sein, daß das Auftreten eines Entities Verkäufer explizit an das Vorhandensein des "entsprechenden" Entities Angestellter gebunden sein muß!
Stichwort: Vererbung

graphische Darstellung:



Formal:

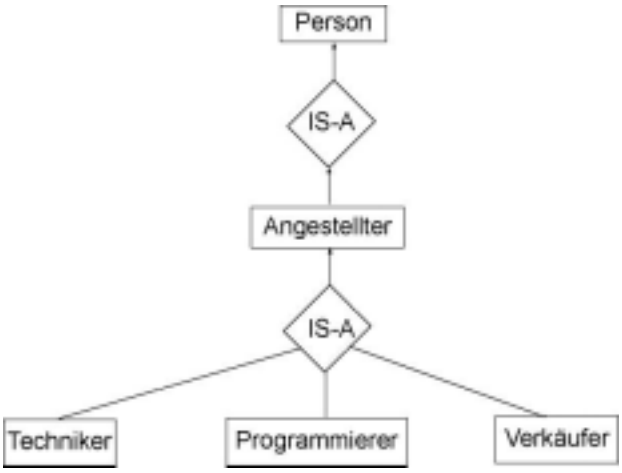
Seien $E_1 = (X_1, K_1)$, $E_2 = (X_2, K_2)$ zwei Entity-Typen.
Dann besteht zwischen E_1 und E_2 eine IS-A-Beziehung der Form E_1 IS-A E_2
$\Leftrightarrow$

i. $(\forall A \in X_2)\ (A \in X_1)$
ii. $(\forall e_1 \in E_1^s)\ (\exists e_2 \in E_2^s)$ mit $e_1(X_2) = e_2(X_2)$

Teil i) regelt die Teilmengenbeziehung der Formate der beteiligten Entity-Typen, Teil ii) regelt die konkrete Ausprägungen in den Entity-Sets

Mit diesen IS-A-Beziehungen lassen sich komplette Vererbungs-Hierarchien aufbauen:
Es kann ja z. B. sein, daß die Angestellten Spezialisierungen eines Entity-Typs Personen sind.

Das führt zur folgenden graphischen Darstellung:



Damit ist das Entity-Relationship-Modell vollständig beschrieben.
Es umfaßt die folgenden Modellierungsinstrumente:
1)Strukturkonstrukte:

i)Entity-Typ mit
- Namen
- Einwertigen und/oder mehrwertigen und/oder zusammengesetzten Attributen und deren Wertebereichen

ii)Relationship-Typ mit
- Namen
- Beteiligten Entity-Typen
- eventuell eigenen Attributen
- Kardinalitäten

iii)Spezialisierungen von Entity-Typen in Form von IS-A-Beziehungen

2) Konkrete Ausprägungen (Bewertungen)
Entities, Entity-Sets
Relationships, Relationship-Sets

Mit diesen Konstrukten kann aus einer realen Situation heraus ein erstes sogenanntes semantisches Datenmodell erstellt werden.
Ursächlich und das Modell bestimmend sind die Festlegungen der E-Typen und R-Typen. Insbesondere wird durch den E-Typ ein Schlüssel festgelegt, dem alle E-Sets und Entities, die irgendwann einmal als konkrete Ausprägung bestimmt werden, genügen müssen!
D. h. durch die E-Typen wird für die Dauer der Gültigkeit des Modells festgelegt, welche Entities sinnvoll sind und vom Modell berücksichtigt werden!! (eben die, die der Schlüsselbedingung genügen)
D. h. durch die E-Typen und die zwischen ihnen bestehenden R-Typen wird ein Regelwerk aufgebaut, das dem Informationsgehalt der realen Situation entspricht und dem also alle E-Sets und Entites genügen müssen, um als sinnvolle Daten zu gelten.
Dieser Gedanke wird sich im Relationalen DB-Modell fortsetzen!

Beispiel:

Gegeben sei die Situationsbeschreibung der Kunden-Auftrag-Produkt-Verwaltung (KAPV).

Wie kommt man jetzt an ein ERM?
Die gängige Methode:
1) Herausschreiben aller Substantive aus der Situationsbeschreibung:

Unternehmen
Produkte
Kunden
Kundenverwaltung
Aufträge
Kundennr
Name
Ort
.....

2) Herausstreichen aller Substantive, die für die Applikation offensichtlich als Objekttyp nicht relevant sind (das sind z. B. eventuell die, zu denen es im Text keine Eigenschaft gibt!) (z. B. Unternehmen, Kundenverwaltung, usw.)

3) Überführen aller Substantive, die offensichtlich Eigenschaften anderer Substantive sind, zu diesen Substantiven:

Produkte (Produktnr, Bezeichnung, usw.)
Kunde (Kdnr, Name, Vorname, usw.)
Auftrag (Autragsnr, usw.)

4) Entfernen aller unsinnigen Substantive:

ERGEBNIS: Eine Liste potentieller Entity-Typen.

Zu diesen E-Typen werden jetzt alle Eigenschaften erhoben, die für die DB-Applikation benötigt werden!
Anschließend muß ein Primärschlüssel gefunden oder definiert werden!

ERGEBNIS:

Kunde = ({Kdnr, Name, Vorname, Gebdat, Geschlecht, {Adresse (PLZ, Ort, Str, Staat)}, Profil (avolumen, zbilanz, zverhalten, {Vorlieben}, Bestellfrequenz), Kundenkonto (abetrag, adatum, {Zahlungen (zdatum, zbetrag)})}, {Kdnr})

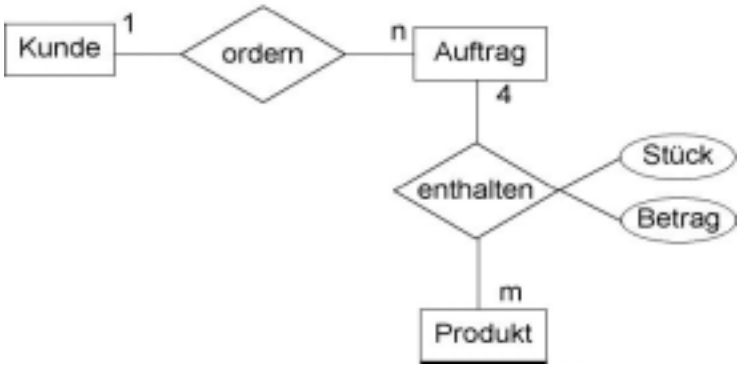
Auftrag = ({anr, adatum, abetrag, astatus, anz.mahnungen, Rechnung (rdatum, rbetrag, rbemerkung)}, {anr})

Produkt = ({prodnr, bez., pdat, material, größe, stckz, preis}, {prodnr})

Zu diesen E-Typen werden dann die R-Typen bestimmt:

ordern = ({Kunde, Auftag}, ∅)
enthalten = ({Auftrag, Produkt}, {stckzahl, Betrag})

Diagramm:



meist werden ER-Diagramme aus Gründen der Übersicht ohne Attribute gezeichnet.

I.2 Das relationale Datenmodell RDM

Genau wie bei jedem Modell müssen wir die Konstrukte und Operationen zur Veränderung der Konstrukte definieren.
Das RDM wurde in den 70'er Jahren von Codd entwickelt und ist seit Mitte der 80'er definierter Standard für DBMS!

Einschub:

Der Name kommt vom mathematischen Konzept einer Relation
(Sind A, B Mengen, so ist $r \subseteq A \times B$ eine Relation zwischen der Menge A und B)
hier: Sind A_1, ... , An Attribute eines Objekts mit den Wertebereichen dom (A_1), ... , dom (A_n) , so ist
$r \subseteq$ dom (A_1) $\times$ ... $\times$ dom (A_n) eine Relation.

I.2.1 Konstrukte

Voraussetzung:

Alle Attribute sind einfach und einwertig!

Formal:

Bezeichnen dom (X) = dom (A_1) $\cup$ ... $\cup$ dom (A_n), für X = {A_1, ... , A_n)}

Definition:

1) Eine Tupel über X ist eine injektive Abbildung.
$\mu : X \rightarrow$ dom (X) mit ($\forall A \in X$) μ (A) $\in$ dom (A)
(d. h. μ (X) = {μ (A_1), ... , μ (A_n)})
Bezeichne Tup(X) die Menge aller Tupel über einer Menge X von Attributen
(injektiv $\Rightarrow$ Tup(X) ist eine Menge, d. h. ohne mehrfache Elemente!)

2) Eine Relation r über X ist eine endliche Menge von Tupel über X,
d. h. $r \subseteq$ Tup(X)
Bezeichne Rel(X) die Menge aller Relationen über X.

3) Die Menge X der Attribute einer Relation heißt Relationenformat.
Hat man die Attribute von X in eine beliebige, aber feste Reihenfolge gebracht, so kann eine
Relation über X als Tabelle dargestellt werden, wo die Spalten die Attribute und die Zeilen die Tupel sind.

Entsprechungen zum ER-Modell:

ERM	**RDM**
Entity	Tupel einer Relation, falls Attribute einwertig und einfach
Entity-Set	Relation
Entity-Format	Relationenformat

Das Ziel ist ja, z. B. über das ER-Modell ein RDM aufzubauen! Deshalb muß man Entsprechungen und ggf. Transformationen suchen und bestimmen!

Bei der Definition eines Tupels gibt es ein Problem:
$(\forall A \in X)\ \mu(A) \in \text{dom}(A)$!

In der Realität: es gibt genügend oft die Situation, daß für ein bestimmtes Attribut für bestimmte Tupel kein Wert angebbar ist: Beispiel: Attribut Telefonnummer

i. es kann Menschen geben, die kein Telefon haben
ii.)es kann Menschen geben, deren Telefonnummer zur Zeit nicht bekannt ist

Folge: diese dürfen nicht als Tupel in die Relation Person oder Kunde aufgenommen werden!
Das ist das Problem der Nullwerte!

Lösung:

Man muß den Wertebereich jedes Attributs erweitern. Will man die obigen beiden Fälle unterscheiden, muß man zwei neue Werte vereinbaren und dem Wertebereich hinzufügen.

z. B. NULL als Wert, falls kein Wert existiert
∅ oder ein anderes Sonderzeichen, z. B. '?', falls ein Wert existiert aber nicht bekannt ist!

also: $(\forall A \in X) : \mu(A) \in \text{dom}(A) \vee \mu(A) \in \{?, \text{NULL}\}$

Problematisch bleibt aber, daß Tupel bzgl. NULLwerte nicht verglichen werden können:
Was bedeutet $\gamma(A) = \mu(A) = \text{NULL}$

nicht, daß die Werte von γ und μ bei A "gleich" sind, höchstens in dem Sinne, daß eben keine Information vorhanden ist!

Bemerkung:

Oracle verwendet den NULL-Wert!
Dieser genießt genau die oben angesprochene Sonderstellung: bei Vergleichen (z. B. in einem Join) werden Tupel mit NULL-Werten nicht berücksichtigt, etwa:

Select * from Kunde where Telnr ≠ 681817

was wird selektiert:
alle Kunden, deren Telnr ≠ 681817 **außer** den Kunden mit NULL-Werten in dieser Spalte!

I.2.2 Datenabhängigkeiten, Konsistenz

Gegeben sein ein Format X, d.h. eine Menge von Attributen.
Untersuchen wir die Menge Rel (X) genauer!
Für die konkrete Situation, aus der das Format X entstanden ist, sind in der Regel nicht alle Relationen die über X gebildet werden können sinnvoll, sonder nur bestimmte! Betrachten wir das Format X zum Objekttyp Mitarbeiter (einer Firma). Da gibt es ein Attribut Gehalt und natürlich sind nur solche Relationen über X sinnvoll, wo der Wert des Attributs Gehalt > 0 ist!
Eine zweite Art (allerdings etwas anders) ist eine Datenabhängigkeit der Form:
Ein Gehalt kann wertmäßig größer werden, niemals kleiner.
Diese Regel legt den Übergang einer Relation r über X durch eine Update-Operation in eine neue Relation s über X fest und definiert, welche Relationen s durch das update erzeugt werden dürfen!

Jeder aus der Realität gewonnene Objekttyp mit seinem Format X enthält eine Reihe von solchen Regeln (Integrität oder Geschäftsregeln), denen alle Relationen über X genügen müssen, um sinnvoll zu sein.
Das sind die sogenannten intrarelationalen Abhängigkeiten.

Eine spezielle solche Abhängigkeit ist die Schlüsselabhängigkeit:

Definition:

Sei X eine Attributmenge, $K \subseteq X$

i) K heißt Schlüssel für $r \in$ Rel (X)
falls a) $(\forall \mu, \nu \in r)\ \mu[K] = \nu[K] \Rightarrow \mu = \nu$
b) für keine echte Teilmenge $K' \subseteq K$ gilt a)

ii) Eine Schlüsselabhängigkeit ist eine intrarelationale Datenabhängigkeit, die alle Relationen $r \in$ Rel (X) bestimmt , die diese Abhängigkeit erfüllen!
Notation: $K \rightarrow X$

$$(K \rightarrow X)(r) = \begin{cases} 1 \text{ falls K Schlüssel für r} \\ 0 \text{ sonst} \end{cases}$$

Also:

$K \subseteq X$ Schlüssel $\Leftrightarrow$ i) $K \rightarrow X$
ii) K minimal, d. h. es existiert kein $K' \subset K$ mit $K' \rightarrow X$

Das heißt nicht, daß es nicht doch Schlüssel K_1 und K_2 von R geben kann mit unterschiedlicher Kardinalität!

Beispiel:

	A	B	C	D	E
	1	1	1	0	0
r =	1	2	2	1	1
	3	3	1	1	1
	1	1	1	1	1

Schlüssel: {B, D}, {B, E} {A, C, D}, {A, C, E} !

Keine andere 2-er-Kombination identifiziert eindeutig,

(∗) alle übrigen identifizierenden 3-er-Kombinationen enthalten 2-er Schlüssel;

(∗∗) Alle identifizierenden 4-er-Kombinationen enthalten Schlüssel als Teilmenge!

(∗) z. B.: {B, C, D}, {B,C, E}, {A, B, D}, usw.

(∗∗) z. B.: {A, B, C, E} usw.

Ist also ein Format X vorgegeben, so legt die Definition eines Schlüssels (ein Schlüsselabhängigkeit) fest, welche Relationen über X gültig sind und welche nicht!

Definition:

Ein Relationenschema (Relationentyp) ist ein benanntes Tupel der Form
$R = (X, \Sigma_X)$ mit:
R: Name, X: Format, Σ_X : Menge der intrarelationalen Datenabhängigkeiten.

Beispiel:

Sei X = {Produktnr, Bezeichnung, Preis, Größe, Farbe, Material, Stückzahl, Produktionsdatum }

$\Sigma_X = \{\ \}$

σ_1 : {Produktnr} $\rightarrow$ X (Schlüsselabhängig)

σ_2 : Preis $> \varnothing$

σ_3 : Farbe $\in$ {rot, gelb, grün, silber}

σ_4 : Material $\in$ {Aluminium, Eisen, Kupfer, Plastik}

σ_5 : Stückzahl ≥ 0

σ_6 : Produktionsdatum $\leq$ Tagesdatum

σ_7 : wenn Stückzahl < 5 und Produktdatum $\leq$ Tagesdatum – 5 Jahre:
Produkt entfernen
bzw. anders: es muß gelten:
Stückzahl ≥ 5 oder Produktdatum $\leq$ Tagesdatum – 5 Jahre

σ_8 : wenn Material = Aluminium dann Farbe = silber

$\Rightarrow$ PRODUKT = (X, Σ_Z) definiert die Struktur aller sinnvollen Relationen (Menge von einzelnen Produkten)

analog muß für die Relationenschemata KUNDE, AUFTRAG, usw. jeweils die Menge Σ_X aller intrarelationaler Abhängigkeiten bestimmt werden!

Ein Relationenschema unterscheidet sich also von einem E-Typ dadurch, daß bei Relationenschemata nicht nur die Schlüsselabhängigkeit, sondern alle intrarelationenalen Datenabhängigkeiten berücksichtigt sind und damit genau angegeben ist, welche Objekte bzw. Objektmengen als sinnvoll erachtet werden!

Wir haben am Anfang die einschränkende Voraussetzung gemacht, daß alle Attribute eines Relationenformats atomar sind (einwertig und einfach)! Wir müssen

also gleich bei der Transformation von ERM zum RDM geeignete Schritte definieren, um dies hier zu erreichen.

Definition:

Ein Relationenschema, dessen Format aus nur elementaren Attributen besteht, heißt Relationenschema in 1 NF (Normalform).

Wir können nun relationale Datenbanken und Datenbankschemata definieren:

Definition:

Sei $R = \{R_1, \ldots, R_k\}$ eine Menge von Relationenschemata in 1NF,
$R_i = (X_i, \Sigma_X)$ $1 \leq i \leq k$ und $X_i \neq X_j$ für $i \neq j$.

i) Eine relationale Datenbank d über R ist eine Menge sogenannter Basisrelationen $d = \{r_1, \ldots, r_k\}$, mit $r_i \in Rel(X_i)$, $1 \leq i \leq k$.

   R heißt Datenbankformat.

ii) genügen alle r_i den Datenabhängigen Σ_{Xi}, heißt d punktweise konsistent!

Genau wie bei einer Relation gibt es natürlich auch Datenabhängigkeiten zwischen verschiedenen Relationen, die interrelationalen Datenabhängigkeiten. Eben z. B. die Beziehungen, die als Relationships zwischen den E-Typen definiert wurden oder die als Geschäftsregel zwischen verschiedenen Objekttypen spezifiziert werden: z. B. Fremdschlüssel-Beziehungen oder die Regel, daß die Stückzahl in der Relation Auftrag kleiner gleich der Stückzahl in der Relation Produkt sein muß.

Konkrete Beispiel in der KAPV:

- Ein Kunde muß mindestens eine Adresse haben
- Ein Auftrag muß genau einen Kunden haben, usw.

Eine spezielle interrelationale Abhängigkeit ist die Fremdschlüsselbeziehung:

Definition:

Seien $R_1 = (X_1, \Sigma_{X1})$, $R_2 = (X_2, \Sigma_{X2})$ Relationenschemata; $K_1 \subseteq X_1$ Primärschlüssel von R_1; r_1 Relationen über R_1, r_2 Relationen über R_2
$Y \subseteq X_2$ heißt Fremdschlüssel in R_2, falls für r_1, r_2 gilt:
$r_2[Y] = r_1[K_1]$.

Es muß also eine Werteentsprechung geben zwischen der Attributmenge Y in X_2 und dem Primärschlüssel K_1 in X_1!

Wir wollen jetzt 'sinnvolle' Datenbanken definieren. Das sind solche, deren Relationen alle die Integritätsregeln (Datenabhängigkeiten) erfüllen, die durch die reale Ausgangssituation vorgegeben sind!

Definition:

Sei $R = \{R_1, \ldots, R_p\}$ ein Datenbankformat; $R_i = (X_i, \Sigma_{Xi})$;
sei Σ_R Menge aller interrelationalen Datenabhängigkeiten

i) Ein relationales Datenbankschema hat die Form $D = (R, \Sigma_R)$ und dient zur Beschreibung der Struktur aller 'sinnvollen' relationalen Datenbanken d (die also alle Σ_{Xi} und Σ_R erfüllen!
ii) Eine relationale Datenbank d heißt konsistent, falls alle Relationen den Σ_{Xi} und Σ_R genügen!

Beispiel:

R_1 = Produkt = (X_p, Σ_{Xp})
R_2 = Kunde = (X_K, Σ_{XK})
R_3 = Auftrag = (X_A, Σ_{XA})
R_4 = Positionen = (X_{pos}, Σ_{XPos})
R_5 = Adresse = (X_{ADR}, Σ_{XADR})

$R = \{R_1, \ldots, R_5\}$ ein Datenbankformat

$\Sigma_R = \{s_1, \ldots, s_K\}$

{s_1: Kundennr ist Fremdschlüssel in ADRESSE
s_2: Kundennr ist Fremdschlüssel in AUFTRAG
s_3: Auftragsnr ist Fremdschlüssel in POSITIONEN
s_4: Produktnr ist Fremdschlüssel in POSITIONEN
s_5: Stückzahl in Auftrag_PROD muß ≤ Stückzahl in PRODUKT sein
.....}

(R, Σ_R) = KAPV ist das DB-Schema der Kunden-Auftrag-Produkt-Verwaltung, das eine konsistente Datenbank und ihre Struktur beschreibt!

Ein relationales Datenbankschema beschreibt also eine bestimmte Datenbank-Struktur, nämlich ein Datenbank-Format und eine Menge interrelationaler Integritätsregeln!
Konkrete relationale Datenbanken, d. h. mit konkreten Basisrelationen $r_1, \ldots, r_p$, die dieser Struktur genügen, heißen konsistent!

Diese Konsistenz einer Rel. DB muß in der Praxis von einem DB-System im laufendem Betrieb über den gesamten Lebenszyklus der DB garantiert werden! Die dazu notwendigen Mechanismen und Konzepte werden wir später kennenlernen (Transaktion, Sperren, Sperrprotokolle).

I.3 Transformation ERM → RDM:

Voraussetzung im RDM: alle Attribute sind einfach und einwertig (1NF)
⇒ zunächst muß im ERM diese Vorschrift erfüllt sein:

1. Schritt:

a) Jedes mehrwertige Attribut eines E-Typs wird überführt in einen eigenen E-Typ, dessen Format aus dem mehrwertigen Attribut und eventuell einem PK-Attribut besteht!

Beispiel:

Person = ({Persnr, Name, {Hobbies}}, {Persnr})
↓ T
Person = ({Persnr, Name}, {Persnr})
Hobby = ({Hnr, Hobbies}, {Hnr})
ausüben = ({Person, Hobby}, ∅)

zusätzlich muß also ein neuer Relationshiptyp modelliert werden, der die Beziehung beschreibt!

b) Bei zusammengesetzten Attributen gibt es drei Möglichkeiten

i) das zusammengesetzte Attribut wird durch seine Komponentenattribute ersetzt, die als 'normale' Attribute aufgefaßt werden.
ii) Komponentenattribute werden zu einem Attribut (String zusammengefaßt
iii) es wird ein eigener E-Typ erzeugt, wieder mit Schlüssel und dem Relationshiptyp, der die Beziehung modelliert!

nur wenn es wichtig ist!

Beispiel:

Person = ({Pnr, Name, Adresse (PLZ, Ort, Str)}; {Pnr}
↓
i) Person = ({Pnr, Name, PLZ, Ort, Str}, {Pnr})
Oder Ii) Person = ({Pnr, Name, Adresse}, {Pnr}) Adresse als String
Oder Iii) Person = ({Pnr, Name}, {Pnr});
Adresse = ({adrnr, plz, Ort, Str},{adrnr})
Wohnen = ({Person, Adresse}, ∅)

c) Zusammengesetzte mehrwertige Attribute werden entsprechend a) und b) in einen eigenen E-Typ + R-Typ überführt! Beispiel: Adresse

2. Schritt:

Jeder flache E-Typ wird überführt in ein Relationenschema unter Beibehaltung des Namens und des Formats. Die Menge Σ_X besteht dann zunächst nur aus der Schlüsselabhängigkeit: $K \rightarrow X$:
$E = (X, K) \rightarrow R = (X, \Sigma_X)$, $\Sigma_X = \{K \text{ ist Schlüssel}\}$

Beispiel:

E = Hobbies = ({Hnr, Hobby}, {Hnr })
$\downarrow T$
R = Hobbies = ({Hnr, Hobby}, {r_1})

r_1: Hnr ist Primärschlüssel zu Hobbies: $\{Hnr\} \rightarrow X$

Alle weiteren Datenabhängigkeiten müssen noch nachgetragen werden!

3. Schritt:

Zweistellige 1:1 oder 1:n Relationships:
es gibt die Möglichkeiten:

i) Die 2-stellige Relationship $R = (\{E_1, E_2\}, Y)$ mit der Kardinalität 1:n wird in ein eigenes Relationenschema überführt! Attribute dieses neuen Schemas sind die Primärschlüssel von E_1 und E_2 und die Attributmenge Y. Schlüssel des neuen Relationenschemas ist der Primarschlüssel von E_2 (des abhängigen E-Typs)!
Diese Möglichkeit muß gewählt werden, falls die Attributmenge Y von R mehrwertige Attribute enthält!

Beispiel:

E_1 = Leser = ({Lesenr, Name}, {Lesenr})
E_2 = Buch = ({Invnr, Titel, Autor}, {Invnr})
Entleihen = ({Leser, Buch}, {Rückgabedat}) 1:n
$\downarrow T$
Ausleihe = ({Invnr, Lesenr, Rückgabedat}, {Invnr ist Schlüssel}

ii) 2. Möglichkeit: der Primärschlüssel von E_1 und die Menge Y werden dem Format X_2 von E_2 hinzufügen. D. h. hier entsteht kein eigenes Relationenschema, die bestehenden werden verändert!

Beispiel wie eben:

{
Leser = ({Lesenr, Name}, {Lesenr})
Buch = ({Invnr, Titel, Autor}, {Invnr})
entleihen = ({Leser, Buch}, {Rückgabedat})

Leser ({Lesenr, Name}, {Lesenr ist PK})
Buch ({Invnr, Titel, Autor, Lesenr, Rückgabedat}, {Invnr ist PK})

Möglichkeit i)

Hat den Nachteil, daß ein zusätzliches Relationenschema erzeugt wird (Aufblähung des DB-Entwurfs).

Möglichkeit ii)

Hat den Nachteil von NULL-Werten bei allen Büchern, die noch nicht ausgeliehen wurden!

anderes Beispiel:

	Professor =	({Profnr, Name, Vorname, Tel}, {Profnr})
↓	Vorlesung =	({Vnr, Titel, Semester}, {Vnr})
	halten =	({Professor, Vorlesung}, ∅) 1:n
.....	Professor =	({....}, {Profnr → X})
	Vorlesung =	({Vnr, Titel, Semester, Profnr}, {{Vnr} → X})

4. Schritt:

Alle Relationships von der Kardinalität n:m oder mit mehr als zwei beteiligten E-Typen werden in ein eigenes Relationenschema überführt!
Ist $R = (\{E_1, E_2, ..., E_r\}, Y)$ so ein Relationshiptyp und ist K_i Schlüssel von E_i (PK), so gilt:

$R \rightarrow (X, \Sigma_X)$ mit : $X = K_1 \cup K_2 \cup ... \cup K_r \cup Y$;

Schlüssel für das neue Relationenschema ist $K_1 \cup ... \cup K_r$!

Beispiel:

prüfen =	({Prof, Vorlesung, Student}, {Prüferg, Prüfdat})
$\downarrow T$	
prüfen =	({Profnr, Vnr, Studnr, Prüferg, Prüfdat}, {Profnr, Vnr, Stdnr ist PK})

Beispiel:.

STUDENT =	({Matrnr, Name, ...}, {Matrnr})
VORLESUNG =	({Vorlnr, Titel, Semester}, {Vorlnr})
R = besuchen =	({STUDENT, VORLESUNG}, ∅) vom Typ n:m
$\Rightarrow R \rightarrow (X, \Sigma_X)$ =	BESUCHEN = ({Matrnr, Vorlnr}, {Matrnr, Vorlnr} → X)

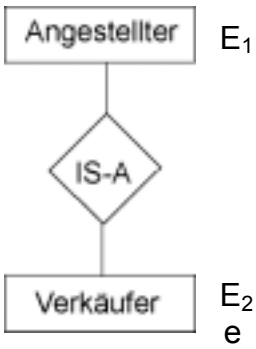
5. Schritt:

IS-A-Beziehung

Bei diesen Relationships braucht kein neues Relationenschema erzeugt zu werden, es werden nur die Relationschemata der beteiligten E-Typen erzeugt.

Dann wird i. a. der Schlüssel der Verallgemeinerung als Schlüssel für die Spezialisierung genommen, bzw. eventuell um weitere Attribute ergänzt:

Beispiel:



E_1 = ({Angnr, Name, Beruf, Gehalt, Abteilung}, {Angnr})
E_2 = ({Angnr, Ausbildung, Schichtnr}, {Angnr})

beide werden in ein Relationschema ANG bzw. VERKÄUFER überführt. Verkäufer hat als Relationenformat nur die speziellen Attribute und den PK!
Wichtig ist, daß die Gebundenheit der Existenz von Entities in der Spezialisierung an die Existenz des entsprechenden Entities in der Verallgemeinerung mit modelliert wird.
Einzige Möglichkeit: als PK-FK-Beziehung!
Im speziellen Schema ist dann der PK gleichzeitig FK zum allgemeinen Schema!
Diese letzte Bemerkung gilt für alle zusätzlichen Schemata, die erzeugt werden mußten:
bei

- Auflösung von mehrwertigen Attributen
- mehrstelligen Relationships
- n:m – Relationships
- 1:n – Relationships
- IS-A-Beziehungen

müssen die Abhängigkeiten der Relationenschemata voneinander durch PK-FK-Beziehungen modelliert werden!

Durch die Überführung der E- und R-Typen aus dem ERM in RDM haben wir also eine Reihe von Relationstypen erzeugt, die zusammen die Struktur einer Datenbank bilden. Insbesondere haben wir auch die Schlüssel aus den E-Typen als Schlüsselabhängigkeiten, d. h. als eine besondere Form von Datenabhängigkeiten modelliert und die Mengen Σ_X gebildet.
Aus der realen Situation haben wir jetzt zusätzlich alle intrarelationalen und interrelationalen Integritätsregeln formuliert und den Mengen Σ_X bzw. Σ_X hinzugefügt. Damit könnte man meinen, ist das Datenbankdesign abgeschlossen!

Als Beispiel das ERM und das resultierende RDM der KAPV:

I.4 Beispiel: Kunden-Auftrag-Produkt-Verwaltung

Ein Unternehmen handelt mit verschiedenen Produkten. Die Verwaltung der Kunden, der Aufträge und der Produkte soll als Client-Server-Datenbank-Applikation realisiert werden. Von den Kunden werden Stammdaten wie Kundennummer, Name, Vorname, Geburtsdatum, Geschlecht, Adressdaten wie PLZ, Ort, Straße und Hausnummer, Staat gehalten und müssen immer aktuell sein. Dabei kann ein Kunde mehrere Adressen haben. Zu jedem Kunden wird ein firmenspezifisches Profil unterhalten, das aus mehreren Angaben besteht: bisheriges Auftragsvolumen (aufsummiert), Zahlungsbilanz mit den Werten '+' oder '-' und Zahlungsverhalten, Vorlieben, dokumentiert in Produktkategorien, Frequenz der Bestellungen. Das Zahlungsverhalten hat die Werte: -1, falls noch Beträge ausstehen, 0, falls mehr als 10 Mahnungen pro Jahr ausgestellt wurden, 1 sonst. Produktkategorien sind: Textilien, Handwerksbedarf, Sport (kann erweitert werden). Die Frequenz der Bestellungen ist die Aufstellung Anzahl_Aufträge pro Jahr. Kunden müssen gegebenenfalls angeschrieben werden. Bestellungen von Kunden werden nur schriftlich akzeptiert.

Zu jeder Kundenbestellung wird ein Auftrag angelegt, bestehend aus einer Auftragsnummer, der Kundennummer, Auftragsdatum, Auftragsbetrag und Auftragsstatus. Der Auftragsstatus hat die Werte 'in Arbeit', 'ausgeliefert', 'bezahlt'. Zu jedem Auftrag gehört eine Liste (Positionen) der bestellten Produkte. Diese Liste beinhaltet die Informationen Auftragsnummer, Produktnummer, bestellte Stückzahl, Preis pro Stück, Betrag . Der Positionenbetrag berechnet sich aus der Stückzahl mal Preis; der Auftragsbetrag berechnet sich aus der Summe aller Positionenbeträge zu dieser Auftragsnummer. Zu jedem Kunden wird ein sogenanntes Kundenkonto gehalten. Darin sind alle Auftragsbeträge mit Bestelldatum und alle eingegangenen Zahlungen zu diesen Aufträgen mit Datum eingetragen. Weiterhin sind pro Auftrag die Anzahl von Mahnungen festgehalten. Mahnungenwerden 21 Tage nach Rechnungsdatum ausgestellt. Die Liste der bestellten Produkte wird mit dem Produktlager verglichen: ist das Produkt in ausreichender Stückzahl vorhanden, wird diese Position in die Auftrags-Bestell-Liste (Positionen) aufgenommen. Sind nicht genügend viele Produkte vorhanden, wird dies auf der Rechnung vermerkt. Ist die Bestelliste abgearbeitet und die Positionsliste komplett, wird das Lager angewiesen, die Produkte zu verpacken. Es wird eine Rechnung ausgestellt und zusammen mit dem Produktpaket verschickt. Eine Rechnung bezieht sich immer auf einen Auftrag und einen Kunden und besteht aus Rechnungsdatum, Rechnungsbetrag und Rechnungsbemerkung. Der Status des Auftrags wird geändert in 'ausgeliefert'. Nach Eingang der Zahlung ändert sich der Status in 'bezahlt'.

Das Produktlager besteht aus den folgenden Informationen: Produktnummer, Bezeichnung, Produktionsdatum, Material, Größe, Stückzahl_auf_Lager, Preis. Nach jeder Auftragsauslieferung muß die Lager-Stückzahl entsprechend verringert werden. Die Liste der Produkte wird immer wieder aktualisiert. Später soll diese KAPV erweitert werden um ein Lieferantenmodul, das die Verwaltung der Produkte mit ihren Lieferanten realisiert.

EERM der KAPV:

E-Typen:

Kunde = ({**kdnr**, name, vorname, gebdat, geschlecht, {adresse(plz, ort, str)},
profil(a_volumen, z_bilanz, z_verhalten, {vorlieben}, bestellfreq),
Kundenkonto(a_betrag, a_datum, {zahlungen(z_dat, z_betrag)}) },
{kdnr})

Auftrag = ({**a_nr**, a_datum, a_betrag, status, anz_mahnungen,
Rechnung(r_datum, r_betrag, r_bemerkung) }, {a_nr})

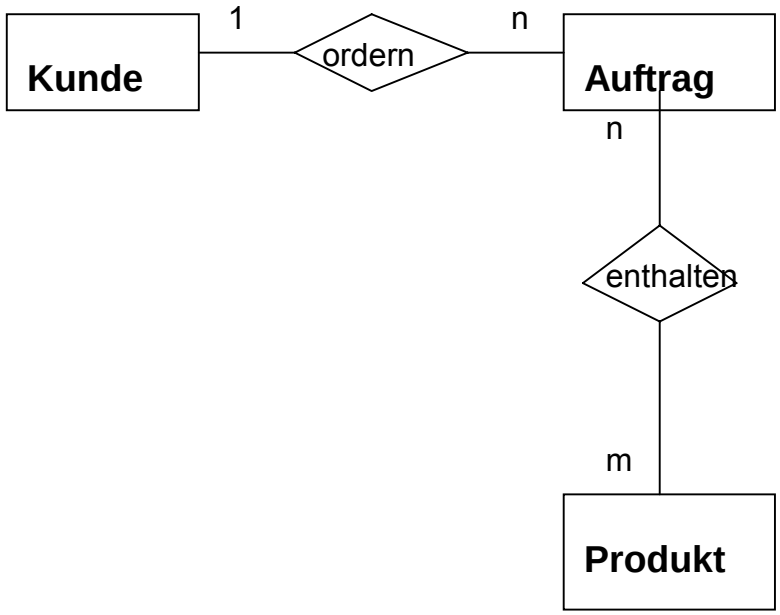
Produkt = ({**prodnr**, bez, p_datum, material, größe, stückzahl, preis}, {prodnr})

R-Typen:

ordern = ({Kunde, Auftrag}, ∅)

enthalten = ({Auftrag, Produkt}, {stückzahl, betrag})

ER-Diagramm:



ERM der KAPV (aus dem EERM):

E-Typen:

Kunde = ({**kdnr**, name, vorname, gebdat, geschlecht}, {kdnr})

Adresse = ({plz, ort, str, adrnr}, {adrnr})

Profil = ({a_volumen, z_bilanz, z_verhalten, , bestellfreq}, {profilnr}),

Vorlieben = ({vnr, Vorlieben}, {vnr})

Konto = ({a_betrag, a_datum, ktnr}, {ktnr})

Zahlungen = ({z_dat, z_betrag, znr}, {znr})

Auftrag = ({**a_nr**, a_datum, a_betrag, status, anz_mahnungen,
r_datum, r_betrag, r_bemerkung }, {a_nr})

Produkt = ({**prodnr**, bez, p_datum, material, größe, stückzahl, preis}, {prodnr})

R-Typen:

ordern = ({Kunde, Auftrag}, ∅)

enthalten = ({Auftrag, Produkt}, {stückzahl, betrag})

wohnen = ({Kunde, Adresse}, ∅)

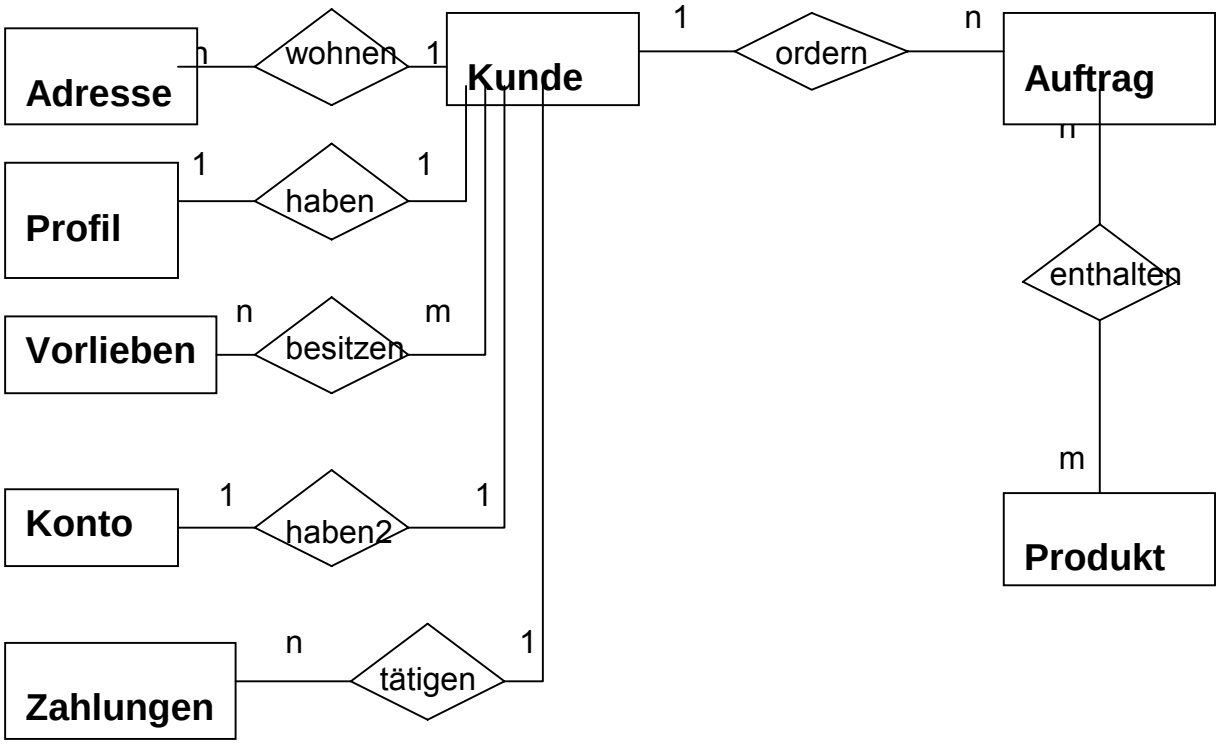
haben = ({Kunde, Profil}, ∅)

besitzen = ({Kunde, Vorlieben}, ∅)

haben2 = ({Kunde, Konto}, ∅)

tätigen = ({Kunde, Zahlungen}, ∅)

ER-Diagramm:


Adresse
n
wohnen
1
Kunde
1
ordern
n
Auftrag
n
Profil
1
haben
1
enthalten
Vorlieben
n
besitzen
m
m
Konto
1
haben2
1
Produkt
Zahlungen
n
tätigen
1

RDM der KAPV

Relationenschemata:

Kunde = ({**kdnr**, name, vorname, gebdat, geschlecht}, Σ_{Kunde})

Adresse = ({plz, ort, str, **adrnr**, kdnr}, $\Sigma_{Adresse}$)

Vorlieben = ({**vnr**, Vorlieben}, $\Sigma_{Vorlieben}$)

Zahlungen = ({z_dat, z_betrag, **znr**, kdnr}, $\Sigma_{Zahlungen}$)

Auftrag = ({**a_nr**, a_datum, a_betrag, status, anz_mahnungen,
r_datum, r_betrag, r_bemerkung , kdnr}, $\Sigma_{Auftrag}$)

Produkt = ({**prodnr**, bez, p_datum, material, größe, stückzahl, preis}, $\Sigma_{Produkt}$)

Positionen = ({**anr, prodnr**, stückzahl, betrag}, $\Sigma_{Positionen}$)

Besitzen = ({kdnr, vnr}, $\Sigma_{Besitzen}$)

Die E-Typen **Profil** und **Konto** bestehen nur aus Attributen, die entweder direkt aus anderen E-Typen übernommenwerden können oder aus solchen berechnet werden können und werden nicht als Relationenschemata modelliert. Die benötigten Werte können als **views** bereitgestellt oder **in der Applikation berechnet** werden!

Noch bestimmt werden müssen die **intrarelationalen Abhängigkeiten**, d.h. die Mengen Σ_X , für X aus Kunde, Adresse,.... und die Menge Σ_R aller **interrelationalen Abhängigkeiten**!
Beispiele für Elemente aus Σ_R :

r_1: kdnr ist Fremdschlüssel in Adresse;
r_2: kdnr ist Fremdschlüssel in Zahlungen;
r_3: kdnr ist Fremdschlüssel in Auftrag;
r_4: anr ist Fremdschlüssel in Positionen;
r_5: stückzahl in Positionen $\leq$ stückzahl in Produkt
...usw

I.5 Funktionale Abhängigkeiten, Normalisierung

Es gibt in Σ_X eines Relationenschemas R = (X, Σ_X) einen Typ von Datenabhängigkeit, dessen spezielle Ausprägungen im laufenden Betrieb der Datenbank Ärger machen können.
Es geht um Funktionale Abhängigkeiten (engl. Functional Dependencies), FD abgekürzt.

Definition:

Sei X ein Relationenformat, $U,V \subseteq X$, $r \in$ Rel (X)
V ist funktional abhängig von U, in Zeichen: $U \rightarrow V$
$\Leftrightarrow (\forall \mu,\nu \in r) (\mu [U] = \nu [U] \Rightarrow \mu [V] = \nu [V])$
d. h. die Werte aller Tupel bzgl. V sind durch die Werte bzgl. U funktional bestimmt!
Insbesondere ist also auch eine Schlüsselabhängigkeit $K \rightarrow X$ eine FD nach Definition des Schlüssels.

Beispiel:

1) ADRESSE = ({Adrnr, Straße, PLZ, Ort, Staat,...} ...)
es gibt die FD {PLZ, Strasse} $\rightarrow$ {Ort}
d. h. durch die Vorgabe von PLZ und einer Strasse ist in jeder Relation über X der Ort eindeutig bestimmt (das ist die Integritätsregel!).

2) X = {A, B, C, D, E}; dom (A) = ... = dom (E) = {0, 1, 2, 3}

	A	B	C	D	E
	1	1	1	1	1
r =	1	0	1	1	1
	2	2	0	0	1
	2	3	2	0	1

Welche funktionale Abhängigkeiten bestehen?

$A \rightarrow B$	$\{A, B\} \rightarrow \{C\}$	(A, B) sind immer verschieden
$A \rightarrow C$		$\forall \mu,\nu \in r$
$A \rightarrow D$		$\Rightarrow \{A, B\} \rightarrow Z$ besteht für alle
$A \rightarrow E$		$Z \subseteq X$
.....		

es gibt auch: $\{A, C\} \rightarrow \{D\}$, usw.....

das gilt allgemein:
Ist $U \subseteq X$ so, daß $\mu [U] \neq \nu [U]$ $\forall \mu,\nu \in r$,
so ist $U \rightarrow Z$ funktional abhängig für beliebige $Z \in$ Pot (X).
Insbesondere ist U eben ein Schlüsselkandidat!

Für diese spezielle Relation r über X = {A, ... , E} gelten also eine Reihe von FD' s!
Normalerweise ist der Weg umgekehrt:

nach Bestimmung des Formats X werden die intrarelationalen Abhängigkeiten bestimmt und in Σ_X festgeschrieben. Darin sind dann auch implizit alle FD's enthalten.
Und in einer konsistenten DB müssen alle Basisrelationen diesen genügen.

I.5.1 2NF

Betrachten wir folgendes Beispiel:

In einer Modellierung einer Hochschul-Verwaltung sei folgender Relationentyp DOZENTEN deklariert und mit einer Relation r realisiert:
R = ({Doznr, Name, Vorname, Tel, Veranstnr, Veransttyp, Veransttitel}, Σ_X)

Doznr	Name	Vorname	Tel	Veranstnr	Veransttyp	Veransttitel
001	Brackly	Günter	629	07	V	DB-Systeme
001	Brackly	Günter	629	08	Ü	DB-Systeme
001	Brackly	Günter	629	09	V	Datenschutz
002	Meier	Horst	777	09	V	Datenschutz
002	Meier	Horst	777	02	V	Netze
003	Schmidt	Karin	123	01	P	Netze

Schlüssel: {Doznr, Veranstnr} (wieso? Nachprüfen!)

Im laufenden Betrieb gibt es mit diesem Relationentyp eine Reihe von Problemen (diese Relation wäre nicht entstanden, wenn man das ERM als Vorstufe genommen hätte, aber man muß es ja nicht tun!)

A) Telnr. von Dozent Brackly ändert sich
Folge: update muß über mehrere Zeilen gemacht werden!
Fehleranfällig wegen der Redundanz.

B) Ein neuer Dozent kommt an die Hochschule, seine Veranstaltungen sind noch nicht klar!
Folge: Da {Doznr, Veranstnr} PK ist, darf Veranstnr nicht NULL sein
$\Rightarrow$ Dozent kann noch nicht eingetragen werden!

C) Praktikum der Dozentin Schmidt wird gestrichen
Folge: (Veranstnr NOT NULL)
$\Rightarrow$ Datensatz muß komplett gelöscht werden
$\Rightarrow$ Dozentin Schmidt ist mehr in der DB
$\Rightarrow$ Informationsverlust!

Worin genau liegen die Probleme?

Einziger Schlüssel des Relationenschemas ist {Doznr, Veranstnr}.

Definition:

R = (X, Σ_X) Relationenschema; A $\in$ X heißt Schlüsselattribut, falls es einen Schlüssel K $\subseteq$ X für R gibt mit A $\in$ K (andernfalls heißt A Nicht-Schlüssel-Attribut)

aber:

Die Nicht-Schlüssel-Attribute Name, Vorname, Telnr hängen nur von der Doznr funktional ab, nicht von der Veranstnr:

{Doznr} $\rightarrow$ {Name, Vorname, Telnr}.

Ebenso hängen die Attribute Veransttyp und Veransttitel nur von Veranstnr, nicht von Doznr funktional ab!:

{Veranstnr} $\rightarrow$ {Veransttyp, Veransttitel}

Verallgemeinerung:

Es gibt Nicht-Schlüssel-Attribute, die von einer echten Teilmenge des Schlüssels abhängen.

Immer, wenn in einem Relationenschema R dieser Fall gegeben ist, treten in allen durch R bestimmten Relationen die obigen Probleme beim update, insert oder delete auf (Potentieller Informationsverlust!).

Zur Wahrung der Konsistenz einer DB ist es also unumgänglich, diese update-Probleme zu lösen. Die Lösung besteht darin, zu fordern, daß kein Relationenschema der DB diese Art funktionaler Abhängigkeiten hat.

2NF:

Eine Datenbank d ist in 2. Normalform, wenn für alle Relationenschemata R = (X, Σ_X) des zugehörigen DB-Formats und für alle Schlüssel K von R gilt:

Kein Nicht-Schlüssel-Attribut (NSA) in X ist von einer echten Teilmenge eines Schlüssels K funktional abhängig.

Die Forderung 2NF an das Dozenten-Relationenschema kann wie erfüllt werden? Aufsplitten des Schemas in diesem Fall in 3 neue:

Dozenten:

Doznr	**Name**	**Vorname**	**Tel**
001	Brackly	Günter	629
002	Meier	Horst	777
003	Schmidt	Karin	123

Veranstal-tungen:

Veranstnr	**Veransttyp**	**Veransttitel**
07	V	DB-Systeme
08	Ü	DB-Systeme
09	V	Datenschutz
02	V	Netze
01	P	Netze

und

Doznr	Veranstnr
001	07
001	08
001	09
002	09
002	02
003	01

wobei wir hier 3 neue Schemata brauchten, weil die Beziehung zwischen Veranstaltungen und Dozenten vom Typ n:m ist!
Ist die Beziehung nur vom Typ 1:n, reichen 2 Schemata, wobei in das abhängige Schema der Primärschlüssel des allgemeinen Schemas als Fremdschlüssel übernommen wird!

Jetzt kann man
- Veranstaltungen löschen, ohne daß Information über Dozenten verschwindet
- Dozenten neu anlegen, ohne deren Veranstaltungen zu kennen
- Daten ändern, ohne auf Redundanz achten zu müssen

I.5.2 3NF

Nächstes Problem:
Nehmen wir an, das neue Relationenschemata Dozent wird erweitert um die Informationen, welchem Fachbereich der Dozent angehört. (Jeder Dozent gehört zu genau einem Fachbereich!)

Dozenten:

Doznr	Name	Vorname	Tel	FBnr	FB-Name
001	Brackly	Günter	629	1	IMST
002	Meier	Horst	777	1	IMST
003	Schmidt	Karin	123	2	Maschinenbau
004	Horn	Sabine	456	3	Architektur

Einziger Schlüssel: {Doznr} d. h. Dozenten ist in 2 NF! (warum? Begründung!)

Welche Probleme können hier im laufenden Betrieb auftreten?

A) Fachbereichsname ändert sich:
Folge: wegen der Redundanz Mehraufwand, Fehleranfällig

B) i) Ein neuer Dozent kommt, ist noch keinem FB zugeordnet:
kein Problem, da Doznr einziges Schlüsselattribut ist.

ii) Ein neuer FB wird gegründet: Kann nicht eingetragen werden bevor nicht mindestens ein Dozent zugeordnet ist!

C) i) Die Dozentin Horn verläßt die Hochschule, wird aus der Dozententabelle gelöscht.
Folge: Der FB Architektur wird mit gelöscht! d. h. die Information, daß an der ein FB Architektur existiert, geht verloren.
ii) Der FB Maschinenbau wird aufgelöst: nicht schlimm, dann hat eben die Prof. Karin Schmidt keine FB-Zuordnung, ist aber noch in der Tabelle.

Wodurch sind die Schwierigkeiten entstanden?
Da der Schlüssel nur aus einem Attribut besteht, ist die Relation in 2 NF, d. h. daher können die Probleme nicht kommen!

Betrachten wir die FD's der Relation Dozenten:
natürlich: $K \rightarrow X$ (K = {Doznr})
und damit $K \rightarrow A$ für alle $A \in X$
aber es gibt auch {FBnr} $\rightarrow$ {FBname}
und damit die Kette $K \rightarrow$ {FBnr} $\rightarrow$ {FBname}
Bezeichnung:
Das NSA FBname ist transitiv abhängig vom Schlüssel.

Und immer, wenn solche transitiven Abhängigkeiten bestehen, treten die oben beschriebenen Probleme auf!

Formal:
Definition:
Sie $R = (X, \Sigma_X)$ ein Relationenschema, $U \subseteq X$ Menge von Attributen. $F \subseteq \Sigma_X$ die Menge der FD's über X
$W \subseteq X$ heißt transitiv abhängig von U bzgl. F
$\Leftrightarrow \exists\ V \subseteq X$ mit $U \rightarrow V$, $V \not\rightarrow U$, $V \rightarrow W$ und $W \not\subset U \cup V$

Im Beispiel:
U = {Doznr}; V = {FBnr}; W = {FBname}
$V \not\rightarrow U$ (z. B. FBnr = 1 definiert verschiedene U-Werte!)

Genau wie eben müssen wir also, um die genannten update-Schwierigkeiten zu vermeiden (und damit die Chance zu erhöhen, daß die Datenbank konsistent bleibt), die Forderung aufstellen, daß kein Relationenschema der Datenbank so eine transitive Abhängigkeit zwischen Attributteilmenge und einem Schlüssel enthält!

3NF:
$R = (X, \Sigma_X)$ ist in 3. Normalform, falls für jedes NSA A und für jeden Schlüssel K von R gilt:
$K \rightarrow A$ ist nicht transitiv
(analog für Mengen von NSA's: W Menge von NSA's $\Rightarrow K \rightarrow W$ ist nicht transitiv).

Die Forderung 3NF kann in unserem Beispiel nur erfüllt werden, indem dieses Relationenschema Dozenten wieder aufgesplittet wird; indem also die transitive FD eliminiert wird! :

Dozenten:

Doznr	**Name**	**Vorname**	**Tel**	**FBnr**
001	Brackly	Günter	629	1
002	Meier	Horst	777	1
003	Schmidt	Karin	123	2
004	Horn	Sabine	456	3

Fach-bereiche:

FBnr	**FB-Name**
1	Eu
2	Maschinenbau
3	Architektur

Wichtig ist hier:

Sowohl 2NF als auch 3NF beziehen sich auf die Nicht-Schlüssel-Attribute eines Relationenschemas!
D. h. bevor man überprüfen kann, ob ein Relationenschema in 2NF oder 3NF ist, muß man alle Schlüssel bestimmen und hat erst dann alle NSA's!!
Es reicht also nicht, 2 NF bzw. 3 NF bzgl. eines Schlüssels zu prüfen!!
Normalformen beziehen sich nicht auf spezielle Schlüssel eines Relationenschemas, sondern auf das Relationenschema mit allen Schlüsseln!!

Ein Datenbankschema, dessen Format in 2 NF oder besser noch 3 NF ist, ist bereits ziemlich robust bzgl. update-Operationen, was die Konsistenz angeht.
Allerdings um den Preis, daß weitere Relationenschemata erzeugt werden mußten.
Je mehr Schemata erzeugt werden, desto mehr Relationen müssen eventuell in einen Join verbunden werden, um die aufgesplittete Information zurückzugewinnen!
Je mehr Relationen in einem Join, desto länger dauert die Ausführung des Statements, d. h. Normalisierung kann auf Kosten der Performance gehen. Und das muß in einer DB-Anwendung beachtet und getestet werden!
Es kann sein, daß zunächst ein ganz Normalisiertes DB-Design erstellt wurde (es gibt noch weitere Normalisierungen wie wir gleich sehen werden) und bei Einsatz festgestellt wird, daß gewisse Abfragen zu lange dauern und dann die Konsequenz gezogen wird, gewisse Normalisierung wieder zurückzunehmen!
Das ist allerdings ein schwerwiegender Schritt, da eventuell die Konsistenz der DB davon betroffen ist, und sollte also nicht leichtfertig gemacht werden!

I.5.3 BCNF

Wir hatten bei 3NF verlangt, daß keine NSA's transitiv vom Schlüssel abhängen dürfen!
Es macht Sinn, diese Forderung auf alle Attribute des Relationenformats auszuweiten, denn:

betrachten wir folgendes Relationenschema AUTO
AUTO = ({Name, Motorleistung, Typbez} , Σ_X)
und die Relation

	Name	**Motorleistung**	**Typbez**
	Sierra	55 PS	1.3 GL
r =	Sierra	75 PS	1.6 GL
	Sierra	105 PS	2.0 GL
	Sierra	175 PS	2.3 GLi
	Mondeo	105 PS	2.0 GL
	Mondeo	145 PS	2.3 GL
	Mondeo	175 PS	2.3 GLi
	.....	.....	.....

welche Schlüssel hat das Schema?

1) Es existieren keine einelementige Schlüssel!
   {Name, Motorleistung} $\rightarrow$ X, da paarweise verschieden!
   {Name, Typbez} $\rightarrow$ X, da paarweise verschieden!
   {Motorleistung, Typbez} !$\rightarrow$ X, da {175 PS , 2.3 GLi} 2 mal gleich, aber auf verschiedene Namen gehen!

$\Rightarrow \exists$ 2 Schlüssel {Name, Motorleistung} und {Name, Typbez}
$\Rightarrow$ insbesondere: es gibt kein NSA!
$\Rightarrow$ AUTO ist trivialerweise in 3NF!

Trotzdem existiert eine transitive Abhängigkeit,
nämlich: {Name, Typbez} $\rightarrow$ {Typbez} $\rightarrow$ {Motorleistung} (nachprüfen!)
d. h. man hat die gleichen update-Probleme wie zuletzt besprochen.

Forderung:
Transitive Abhängigkeiten dürfen generell nicht auftreten.

BCNF:
Ein Relationenschema R = (X, Σ_X) ist in Boyce-Codd-Normalform (BCNF), falls für jede Attributmenge W und für jeden Schlüssel K von R gilt:
K $\rightarrow$ W ist nicht transitiv!

Lösung im Beispiel:
Überführung in BCNF bedeutet die Eliminierung aller FD's aus dem Relationenschema, deren linker Operand kein Schlüssel ist. (Die verursachen die Transitivitäten!)
Unser Relationenschema AUTO hatte folgende FD's:
{Name, Motorleistung} $\rightarrow$ {Typbez}

{Name, Typbez} → {Motorleistung} } Schlüsselbeziehungen

{Motorleistung} ↔ {Typbez} verursacht die Transitivität!

Die letztere muß eliminiert werden:

⇒ AUTO = ({Name, Motorleistung, Typbez}, Σ_X)

↓

AUTO 1= ({Name, Motorleistung}, Σ_{X1})

AUTO 2= ({Motorleistung, Typbez}, Σ_{X2})

Die bisher besprochenen Normalisierungen beziehen sich auf FD's die in einem Relationenformat bestehen und die update-Anomalien verursachen.
Hat man diese eliminiert, ist das resultierende Datenbankschema schon ziemlich robust bzgl. update-Operationen. Allerdings gibt es immer noch Fälle, die zu Anomalien im laufenden Betrieb führen können.

I.5.4 Mehrwertige Abhängigkeiten und 4NF

Betrachten wir folgende Relation VORLESUNG:

Vorlesung:

Vorlesungstitel	**Dozent**	**Bücher**
Informatik	Grün	Algorithmen
Informatik	Grün	Datenbanken
Informatik	Braun	Algorithmen
Informatik	Braun	Datenbanken
Mathematik	Grün	Algorithmen
Mathematik	Grün	Vektoranalysis
Mathematik	Grün	Analyt. Geometrie

Das Beispiel muß den folgenden Integritätsregeln genügen:

i) Eine Vorlesung kann von mehreren Dozenten gelesen werden (über die Jahre verteilt abwechselnd)

ii) Jeder Vorlesung liegt eine fest definierte Menge von Büchern zugrunde, an die sich jeder Dozent halten muß (d. h. es existiert keine Abhängigkeit zwischen Dozent und Bücher), die beiden Attribute sind vollständig unabhängig voneinander!

Zu beobachten ist:

es existiert keine FD in VORLESUNG,
d. h. insbesondere VORLESUNG ist in BCNF!

Trotzdem gibt es in laufendem Betrieb update-Probleme, z. B. wenn ein neuer Mathe-Dozent eingefügt werden soll, müssen drei neue Datensätze erzeugt werden!
oder:
Soll Dozent Grün als Mathe-Dozent gelöscht werden, verschwindet auch die Information, welche Bücher der Mathe-Vorlesung zugrunde liegen!
Es ist also unbedingt erforderlich, dieses Relationenschema umzuwandeln.

Natürlich bietet es sich an, aus VORLESUNG die beiden Schemata
Vorl_Doz = ({Vorlesung, Dozent}, Σ_X) und
Vorl_Buch = ({Vorlesung, Bücher}, Σ_X)
zu erzeugen, die auch beide in BCNF sind!

Leider können wir nicht wie bei den beiden bisherigen Normalisierungen so vorgehen, daß die FD's, die die Probleme schaffen, eliminiert werden und in ein neues Schema gepackt werden, da es hier keine FD's gibt.

Es gibt allerdings die folgenden Abhängigkeiten:
- jede Vorlesung bestimmt eine Menge von Dozenten!
- jede Vorlesung bestimmt eine Menge von Büchern!

Bezeichnung:

Dozent ist mehrwertig abhängig von Vorlesung!
bzw. Bücher mehrwertig abhängig von Vorlesung!

Definition:

$R = (X, \Sigma_X)$; $U,V \subseteq X;0$ $W = X \setminus (U \cup V)$; $r \in Rel(X)$
V heißt mehrwertig abhängig von U ($U \rightarrow\!\!\!\rightarrow V$)
$\Leftrightarrow$
Die Menge der V-Werte, die zu einem gegebenen (U-Werte, W-Werte)-Paar in r gehören, hängt nur vom U-Wert ab und ist unabhängig vom W-Wert.
(d. h. V und W sind vollständig unabhängig voneinander!)

Es gilt:

Existiert in r die MVD: $U \rightarrow\!\!\!\rightarrow V$, so existiert auch $U \rightarrow\!\!\!\rightarrow W$
mit $W = X \setminus (U \cup V)$.
gilt $W = \varnothing$, so heißt die MVD trivial.

Notation: $U \rightarrow\!\!\!\rightarrow V \mid W$

MVD's sind Verallgemeinerungen von FD's:

- jede FD ist eine MVD, bei der die Menge V der abhängigen Werte, die zu einem bestimmten Wert gehören, immer einwertig ist!
- es gibt MVD's, die keine FD's sind

Definition (**4NF**):

$R = (X, \Sigma_X)$; $U,V \subseteq X$; $r \in Rel(X)$

r ist in 4. Normalform $\Leftrightarrow$ {
- i) alle MVD's sind trivial oder
- ii) existiert in r eine nicht triviale MVD $U \rightarrow\rightarrow V$, dann sind alle Attribute von R auch funktional abhängig von U (d. h. U ist ein Schlüsselkandidat).

Äquivalent:

r ist in 4NF $\Leftrightarrow$ {
- i) alle MVD's sind trivial oder
- ii) r ist in BCNF und jede nicht triviale MVD geht von einem Schlüsselkandidaten aus.

Beispiel:

r =

Vorlesung	**Dozent**	**Buch**
Physik	Grün	Mechanik
Physik	Grün	Optik
Physik	Braun	Mechanik
Physik	Braun	Optik
Mathematik	Grün	Algebra
Mathematik	Grün	Mechanik
Mathematik	Grün	Geometrie

MVD's:

m: Vorlesung $\rightarrow\rightarrow$ Dozent | Buch; Dozent und Buch sind unabhängig
m ist nicht trivial; {Vorlesung} kein Schlüsselkandidat
$\Rightarrow$ r nicht in 4NF

Auflösung:

Wieder muß das ursprüngliche Relationenschema aufgesplittet werden:

V_D: r_1 =

Vorlesung	**Dozent**
Physik	Grün
Physik	Braun
Mathematik	Grün

V_B: r_2 =

Vorlesung	**Buch**
Physik	Mechanik
Physik	Optik
Mathematik	Algebra
Mathematik	Mechanik
Mathematik	Geometrie

V_D und V_B enthalten nur die trivialen MVD's
Vorl $\rightarrow\rightarrow$ Doz, Vorl $\rightarrow\rightarrow$ Buch,
d. h. V_D und V_B sind in 4NF!

Anderes Beispiel:

Gegeben sei das Schema
Dozent = ({Doznr, Name, Vorname, Kind, Alter, Lehrgebiet}, Σ_X) und die Relation:

r =

Doznr	**Name**	**Vorname**	**Kind**	**Alter**	**Lehrgebiet**
001	Meier	Horst	Nils	6	Informatik
001	Meier	Horst	Nils	6	Mathematik
001	Meier	Horst	Svenja	2	Informatik
001	Meier	Horst	Svenja	2	Mathematik
002	Schmidt	Sabine	Peter	3	Philosophie
002	Schmidt	Sabine	Peter	3	Mathematik
002	Schmidt	Sabine	Peter	3	Physik

MVD's: {Doznr, Name, Vorname} $\rightarrow\!\!\rightarrow$ {Kind, Alter}
{Doznr, Name, Vorname} $\rightarrow\!\!\rightarrow$ {Lehrgebiet}

hier ist noch klarer, daß es keinen Zusammenhang zwischen {Kind, Alter} und {Lehrgebiet} gibt, also keine FD!

Bezeichne U = {Doznr, Name, Vorname},
V = {Kind, Alter}
so ist W = X \ (U $\cup$ V) = {Lehrgebiet} $\neq \emptyset$, d. h. U $\rightarrow\!\!\rightarrow$ V ist nicht trivial!
analog {Doznr, Name, Vorname} $\rightarrow$ {Lehrgebiet} nicht trivial,
d. h. Dozent ist nicht in 4NF.

Lösung:
Aufsplitten des Relationenschemas DOZENT in DOZENT_KIND und DOZENT_LEHRGEBIET, die dann nur noch triviale MVD's enthalten!

Ein Relationenschema mit mehrwertigen Abhängigkeiten, die nicht trivial sind, wird also in der Regel in 4NF gebracht, indem die MVD's trivialisiert werden:
R = (X, Σ_X); U, V, W $\subseteq$ X
U $\rightarrow\!\!\rightarrow$ V | W

Dann wird R ersetzt durch
R_1 = (U $\cup$ V, $\Sigma_{U \cup V}$) mit U $\rightarrow\!\!\rightarrow$ V trivial
R_2 = (U $\cup$ W, $\Sigma_{U \cup W}$) mit U $\rightarrow\!\!\rightarrow$ W trivial

bzw. besser (zur Vermeidung von Redundanzen):

ist K Schlüssel zu U, bilde

$R_1 = (U, \{K \rightarrow U\} \cup \Sigma_U)$

$R_2 = (K \cup V, \Sigma_{K \cup V})$ mit $K \rightarrow\rightarrow V$ trivial

$R_2 = (K \cup W, \Sigma_{K \cup W})$ mit $K \rightarrow\rightarrow W$ trivial

Im Beispiel:

R_1 = Dozent = ({Doznr, Name, Vorname}, {Doznr})

R_2 = Doz_Kind = ({Doznr, Kindname, Kindalter}, Σ_V)
mit {Doznr} $\rightarrow\rightarrow$ {Kindname, Kindalter} trivial

R_3 = Doz_Lehre = ({Doznr, Lehrgebiet}, Σ_W)
mit {Doznr} $\rightarrow\rightarrow$ {Lehrgebiet} trivial

Lassen sich in einem gegebenen Relationenschema R die MVD's nicht trivialisieren, und will man trotzdem eine 4 NF haben, muß versucht werden, das Relationenschema R so zu verändern, daß im neuen Schema U ein Schlüsselkandidat (eventuell durch Splitten von R).
Damit sind die Überlegungen zum relationalen DB-Design vorläufig abgeschlossen. Sie werden der Relationenalgebra wieder aufgenommen!

I.6 Relationenalgebra

Basiskonstrukt des RDM ist die Relation bzw. das Relationenschema mit Relationenformat und der Menge der intrarelationalen Abhängigkeiten.
Ziel des DB-Einsatzes ist die permanente Speicherung von Information und die Erzeugung neuer Informationen durch sinnvolle Veränderung und Verknüpfung der gespeicherten Information. D. h. notwendig für den DB-Einsatz sind nicht nur die Möglichkeiten der persistenten (und konsistenten) Speicherung, sondern auch Operationen auf den Konstrukten, die diese Konstrukte verändern können. Es muß also Operationen geben, die aus den bestehenden Basisrelationen einer Relationalen DB neue erzeugen können:
Die beiden grundlegenden Operationen in diesen Zusammenhang sind Projektion und Selektion:

Definition:

Sei $R = (X, \Sigma_X)$ ein Relationenschema, $r \in Rel(X)$ und $Y \subseteq X$

i) $\pi_Y(r) := \{ \mu[Y] \mid \nu \in r \}$ heißt Projektion von r auf die Attributmenge Y

ii) Sei $A \in X$, $a \in dom(A)$ und $\theta \in \{<, \leq, >, \geq, \equiv, <, >\}$

$\sigma_{A\theta a}(r) := \{\mu \in r \mid \mu[A] \; \theta \; a\}$ heißt Selektion von r bzgl. $A \; \theta \; a$

iii) $A, B \in X$ mit $dom(A) = dom(B)$ und $\theta \in \{<, \leq, >, \geq, \equiv, <>\}$
$\sigma_{A\theta B}(r) = \{\mu \in r \mid \mu[A] \; \theta \; \mu[B]\}$ heißt Selektion von r bzgl. $A \; \theta \; B$

Erläuterung:

Eine Projektion erzeugt also eine neue Relation r', entstanden aus r durch Reduktion des Formats X auf das neue Format $Y \subseteq X$;
Eine Selektion erzeugt eine neue Relation s durch Reduktion von r auf die Tupel, die das Kriterium $A\theta a$ erfüllen, bzw. $A\ \theta\ B$.

Beispiel:

$R = (X, \Sigma_X)$ mit X = A, B, C; dom (A) = dom(B) = dom (C) = {1, 2}

	A	B	C
	1	2	1
r =	1	1	1
	2	2	2

i) π_{AC} (r) = r':

A	C
1	1
1	2

insbesondere nur unterschiedliche Tupel!

ii) $\sigma_{C=1}$ (r) =

A	B	C
1	2	1
1	1	1

iii) $\sigma_{A\neq B}$ (r) =

A	B	C
1	2	1

Man kann bei der Selektion natürlich verschiedene Bedingungen mit $\wedge, \vee, \neg$ verknüpfen und so komplexe Ausdrücke formulieren:

Gesucht sind alle Tupel aus r die für B den Wert 2 haben und für A den Wert 1 oder für C den Wert ≥ 2 haben.

Formal:

$\sigma_{B=2\ \wedge\ (A = 1\ \vee\ C \geq 2)}$ (r) =

A	B	C
1	2	1
2	2	2

Natürlich können Projektion und Selektion kombiniert werden:

Gesucht ist die Bezeichnung aller Produkte, die aus Aluminium sind und vor dem 1.1.97 produziert werden:

$\pi_{\text{Bezeichnung}}$ ($\sigma_{\text{Material = 'Aluminium'}\ \wedge\ \text{Produktionsdatum} < \text{'01-JAN-97'}}$) (Produkt)

Es gelten die folgenden Rechenregeln:

Satz 1:

Sei X Relationenformat, $r \in \text{Rel}(X)$;

a) $U \subseteq V \subseteq X \Rightarrow \pi_U(\pi_V(r)) = \pi_U(r)$
b) $U, V \subseteq X$ beliebig $\Rightarrow \pi_U(\pi_V(r)) = \pi_{U \cap V}(r)$
   Seien C_1, C_2 komplexe Selektionsausdrücke der Form $A\ \theta\ B \wedge C\ \theta\ c_{1.....}$
c) $\sigma_{C1}(\sigma_{C2}(r)) = \sigma_{C2}(\sigma_{C1}(r))$, d. h. Selektionen können beliebig vertauscht werden!
d) $A \in U \subseteq X$
   $\Rightarrow \pi_U(\sigma_{A\theta a}(r)) = (\sigma_{A\theta a}(\pi_U(r))$
   bzw.
   $A, B \in U \subseteq X$
   $\Rightarrow \pi_U(\sigma_{A\theta B}(r)) = (\sigma_{A\theta B}(\pi_U(r))$

Relationen sind **Mengen von Tupeln**, d. h. auf Relationen kann man auch die üblichen Mengenoperationen $\cup$, $\cap$, \ anwenden.

Definition:

Seien $r, s \in \text{Rel}(X)$;
i) $r' = r \cup s := \{\mu \in \text{Tup}(X) \mid \mu \in r \vee \mu \in s\} \subseteq \text{Tup}(X)$
ii) $r' = r \setminus s := \{\mu \in \text{Tup}(X) \mid \mu \in r \wedge \mu \notin s\} \subseteq \text{Tup}(X)$

daraus läßt sich der Durchschnitt ableiten:

$r \cap s := r \setminus (r \setminus s)$

r und s müssen über dem **gleichen Format** definiert sein, da alle Mengen bzgl $\cup$, $\cap$, \ als Teilmengen von Tup (X) definiert sind!
Dies muß eventuell durch eine Umbenennung erreicht werden!

Beispiel:

$X = \{A, B, C, D\}$

	A	B	C	D
r =	1	2	3	4
	1	1	1	1
	2	1	4	1

	A	B	C	D
s =	0	0	1	2
	1	0	1	0
	1	1	0	1
	1	1	1	1

i) $\sigma_{B=C}(s) \cup r = r$

ii) $r \setminus s =$

A	B	C	D
1	2	3	4
2	1	4	1

iii) $r \cap s =$

A	B	C	D
1	1	1	1

iv) $\pi_{A \cup B}(r) \cup \pi_{A \cup B}(s)$ =

A	B
1	2
1	1
2	1
0	0
1	0

Die bisher besprochenen Operationen können also kombiniert werden. Es sind Operationen auf Relationen und liefern als Ergebnis immer wieder eine Relation!

Die nächste Operation ist zentral für das Relationale Datenmodell und fürs praktische Arbeiten mit Relationalen DB-Systemen.

Definition: (Verbund)

a) Sein X_1, X_2 Relationenformate, $r_1 \in \text{Rel}(X_1)$, $r_2 \in \text{Rel}(X_2)$
r_1 join $r_2 := \{\mu \in \text{Tup}(X_1 \cup X_2) \mid \mu[X_1] \in r_1 \wedge \mu[X_2] \in r_2\}$
heißt (natürlicher) Verbund von r_1 und r_2

b) Allgemein: Seien $X_1, \ldots, X_n$ Relationenformate, $r_i \in \text{Rel}(X_i)$
r_1 join ... join $r_n = \{\mu \in \text{Tup}(\bigcup_{i=1}^{n} X_i) \mid \mu[X_1] \in r_1 \wedge \ldots \wedge \mu[X_n] \in r_n\}$

Beispiele:

1) $X_1 = \{A, B, C\}$; $X_2 = \{B, D, E\}$

r =

A	B	C
1	2	3
0	1	1
0	0	1

s =

B	D	E
1	0	0
1	1	0
0	1	1

r join s = $\{\mu \in \text{Tup}(X_1 \cup X_2) \mid \mu[X_1] \in r \text{ und } \mu[X_2] \in s\}$

$X_1 \cap X_2 = \{B\}$; wegen $\mu[X_1] \in r$ und $\mu[X_2] \in s$ können nur solche Tupel im join vorkommen, die in r und s in {B} übereinstimmen!
also:

r join s =

A	B	0	D
0	1	1	0
0	1	1	0
0	0	1	1

} 1 kommt in s in B zweimal vor!

in r existiert noch ein Tupel mit B = 2, aber in s existiert kein Tupel mit (2, ?, ?), d. h. wegen $\mu[X_1] \in r$ und $\mu[X_2] \in s$ kann es kein Tupel im join geben mit dem Wert 2 für B!

2)

	A	B	C
r =	1	1	0
	1	0	1
	1	1	1
	0	1	0

	A	C	D
s =	1	0	0
	0	0	0
	0	1	1

r join s = ?

$X_1 = \{A, B, C\}$; $X_2 = \{A, C, D\}$

Beide Formate haben {A, C} gemeinsam.
Wegen der Bedingung $\mu [A, B, C] \in r \wedge \mu [A, C, D] \in s$ können also insbesondere nur solche Tupel im Verbund sein, die in r und in s in {A, C} übereinstimmen!
also:

	A	B	C	D
r join s =	1	1	0	0
	0	1	1	1

da $\{\mu [A, C], \mu \in r\} \cap \{\nu [A, C], \nu \in s\} = \{(1, 0), (0, 1)\}$

zwar existieren in r noch 1 Tupel mit

A	B
1	1

so ein Tupel existiert aber nicht in s!
$\Rightarrow$ es existiert kein $\mu \in s$ mit $\mu [A, C, D] = (1, 1, ?)$
$\Rightarrow$ es kann im Verbund kein Tupel geben mit den Werten 1 1 in A C!
analog für A C = 0 0 in s!

3) r wie in 2);

	D	E
s =	0	1
	1	0

$\Rightarrow$ r join s = ?
$X_1 = \{A, B, C\}$; $X_2 = \{D, E\}$ und $X_1 \cap X_2 = \varnothing$
$\Rightarrow$ für jede Kombination von Tupeln μ aus r mit Tupeln ν aus s gilt:
$(\mu \times \nu) [X_1] \in r \wedge (\mu \times \nu) [X_2] \in s$

$\Rightarrow$ r join s ist das kartesische Produkt aller Tupel aus r mit allen Tupeln aus s!

<u>Also:</u>

Die definierende Eigenschaft eines Verbundes ist:
Die Einschränkung jedes Tupels aus dem Verbund muß in der jeweils ursprünglichen Relation liegen!
$\Rightarrow$ Bei der Bildung des Verbundes muß erst überprüft werden, ob für die Formate X und Y gilt: $X \cap Y \neq \varnothing$
gilt $X \cap Y \neq \varnothing$, so existieren im Verbund nur solche Tupel, deren Einschränkung auf $X \cap Y$ in u <u>und</u> in s vorkommen!

gilt $X \cap Y = \emptyset$, so ist r join s = $r \times s$ das kartesische Produkt.

es kann sogar sein, daß der Verbund leer ist:

	A	B	C		A	B	D
r =	1	0	0	s =	0	0	1
	1	1	0		0	1	1
	1	1	1		0	1	0

$\Rightarrow$ $X = \{A, B, C\}$; $Y = \{A, B, D\}$; $X \cap Y = \{A, B\}$
r [A, B] = {(1, 0), (1, 1)}; s [A, B] = {(0, 0), (0, 1)}

$\Rightarrow$ r [A, B] $\cap$ s [A, B] = $\emptyset$,
d. h. es existieren keine Tupel von r und s, die auf der gemeinsamen Attributmenge übereinstimmen

$\Rightarrow$ r join s = $\emptyset$

<u>Regeln für den Verbund:</u>

<u>Satz 2:</u>
Sei $r_i \in$ Rel (X_i) für i = 1, 2, 3. Dann gilt:

i) r_i join $r_i = r_i$
ii) r_1 join $r_2 = r_2$ join r_1 Vertauschbarkeit wichtig für Performance
iii) (r_1 join r_2) join $r_3 = r_1$ join (r_2 join r_3) Beliebigkeit der Bearbeitung
iv) $X_1 \cap X_2 = \emptyset \Rightarrow r_1$ join $r_2 = r_1 \times r_2$
v) $X_1 = X_2 \Rightarrow r_1$ join $r_2 = r_1 \cap r_2$

Wie ist nun der Zusammenhang zwischen Verbund und Projektion?
Das sind ja eigentlich zueinander inverse Operationen:
Projektionen zerlegen eine gegebene Relation bzgl. des Formats X, Verbund fügt Relationen zu einer neuen zusammen.
Man sollte also erwarten können, daß, wenn
$X = X_1 \cup X_2$ $r \in$ Rel (X), $r_1 = \pi_{x1}$ (r) , $r_2 = \pi_{x2}$ (r)
dann gilt:
$r = \pi_{x1}$ (r) join π_{x2} (r) ?

<u>Machen wir ein Beispiel:</u>

Sei $X_1 = \{A, B\}$; $X_2 = \{B, C\}$; $X = \{A, B, C\}$
$r \in$ Rel (X) mit

	A	B	C
r =	0	0	0
	1	0	1

$\Rightarrow$ π_{X1} (r) =

A	B
0	0
1	0

π_{X1} (r) =

B	C
0	0
0	1

} π_{X1} (r) join π_{X2} (r) =

A	B	C
0	0	0
0	0	1
1	0	0
1	0	1

also insbesondere $r \neq \pi_{X1}$ (r) join π_{X2} (r)

genauer: $r \subseteq \pi_{X1}$ (r) join π_{X2} (r), d. h. bei Projektion bekommt man durch einen Verbund mindestens all die Tupel, die ursprünglich zu r gehörten, eventuell mehr!

trivial:

$X_1 \cap X_2 = \emptyset$, dann gilt $r \not\subseteq \pi_{X1}$ (r) join π_{X2} (r) = kartesisches Produkt
(falls r nicht selbst das kartesische Produkt ist!)

2. Problem:

Kann man durch eine Projektion eine Komponente eines Verbundes herausschneiden? D. h. gilt:
π_{X2} (r_1 join r_2) = r_2 ? (mit $X = X_1 \cup X_2$, $r_i \in$ Rel (X_i))

Betrachte

r_1 =

A	B
0	0
0	1

r_2 =

B	C
0	0
0	1

$\Rightarrow$ r_1 join r_2 =

A	B	C
0	0	0
0	0	1

aber: $\pi_{\{A, B\}}$ (r_1 join r_2) =

A	B
0	0

$\neq r_1$

genauer: $\pi_{\{A, B\}}$ (r_1 join r_2) $\subseteq r_1$
D. h. die Projektion eines Verbundes ist enthalten in einer Komponente!

Fassen wir diese Überlegungen zusammen:
Für Verbund und Projektion gelten die folgenden allgemeinen Zusammenhänge:

Satz 3:

Sei $X = \bigcup_{i=1}^{n} X_i$; $r \in Rel(X)$, $r_i \in Rel(X_i)$

Dann gilt:

i) $r \subseteq \pi_{X1}(r)$ join $\pi_{X2}(r)$ join .... join $\pi_{Xn}(r)$

ii) $\forall\ 1 \le j \le n$ gilt : $\pi_{Xj}(r_1$ join .... join $r_n) \subseteq r_j$

iii) $\forall\ 1 \le j \le n$ gilt : $\pi_{Xj}(\pi_{Xj}(r)$ join .... join $\pi_{Xn}(r)) = \pi_{Xj}(r)$

Anmerkung:
Die Fälle i) und ii) haben wir durch Beispiele veranschaulicht.

Aus den Überlegungen zu Projektion und Verbund ergibt sich die folgende Definition:

Definition:
Ist $X = X_1 \cup ... \cup X_n$ eine Zerlegung einer vorgegebenen Relationenformats und sind $R_1 = (X_1, \Sigma_{X1})$, ... , $R_n = (X_n, \Sigma_{Xn})$ die resultierenden Relationenschemata, $r \in Rel(X)$,
so heißt diese Zerlegung verlustfrei (verbundtreu)
$\Leftrightarrow$ wenn gilt:
$r = \pi_{X1}(r)$ join .... join $\pi_{Xn}(r)$

Neben den Normalisierungen zur Vermeidung von update-Anomalien ist es also immens wichtig zu wissen, daß das resultierende Datenbank-Design auch verlustfrei ist, d. h. alle Originalinformationen durch Verbundoperationen aus den aufgesplitteten Relationenschemata wieder zusammengefügt werden können!
Wir wollen die am Beispiel der Dozenten-Relationen nachvollziehen:

Beispiel Dozentenrelation:

r =

Doznr	**Name**	**Vorname**	**Telnr**	**Veranstnr**	**Veransttyp**	**Veransttitel**
001	Meier	Horst	123	07	V	Datenbanken
001	Meier	Horst	123	08	Ü	Datenbanken
001	Meier	Horst	123	09	V	Mathematik
002	Schmidt	Sabine	456	01	V	Rechnernetze
002	Schmidt	Sabine	456	09	V	Mathematik
003	Schulze	Karin	789	02	P	Programmieren

Also X = {doznr, name, vorname, telnr, veranstnr, veransttyp, veransttitel}

Normalisieren in 2NF durch die Projektionen:

X1 = {doznr, name, vorname, telnr} $r_1 = \pi_{X1}(r)$
X2 = {veranstnr, veransttyp, veransttitel} $r_2 = \pi_{X2}(r)$
X3 = {veranstnr, doznr} $r_3 = \pi_{X3}(r)$

Bestimmen Sie die Projektionen und beweisen Sie durch ausrechnen:

$r = \pi_{X1}(r)$ join $\pi_{X2}(r)$ join $\pi_{X3}(r)$

D. h. die hier durch Normalisierung bestimmte Zerlegung ist verlustfrei !!

π_{X1} (r) =

Doznr	**Name**	**Vorname**	**Telnr**
001	Meier	Horst	123
002	Schmidt	Sabine	456
003	Schulze	Karin	789

π_{X2} (r) =

Veranstnr	**Veransttyp**	**Veransttitel**
01	V	Rechnernetze
02	P	Programmieren
07	V	Datenbanken
08	Ü	Datenbanken
09	V	Mathematik

π_{X3} (r) =

Veranstnr	**Doznr**
01	002
02	003
07	001
08	001
09	001
09	001

π_{X1} (r) join π_{X3} (r) =

Doznr	Name	Vorname	Telnr	Veranstnr
001	Meier	Horst	123	07
001	Meier	Horst	123	08
001	Meier	Horst	123	09
002	Schmidt	Sabine	456	01
002	Schmidt	Sabine	456	09
003	Schulze	Karin	789	02

π_{X1} (r) join π_{X3} (r) join π_{X3} (r) =

Doznr	Name	Vorname	Telnr	Vnr	Vtyp	Vtitel
001	Meier	Horst	123	07	V	Datenbanken
001	Meier	Horst	123	08	Ü	Datenbanken
001	Meier	Horst	123	09	V	Mathematik
002	Schmidt	Sabine	456	01	V	Rechnernetze
002	Schmidt	Sabine	456	09	V	Mathematik
003	Schulze	Karin	789	02	P	Programmieren

= r, d. h. verlustfrei!

Zur Überprüfung auf Verbundtreue gibt es das folgende Kriterium:

Kriterium:

Sei X Relationenformat, F Menge von FD's
R = (X, F); D = (R_1,..R_n, ...); R_i = (X_i, F_i) Zerlegung
Dann gilt:
existiert ein $X^* \subseteq X$ und $X^* \subseteq X_i$ für ein $i \in \{1, ... , n\}$
mit $X^* \rightarrow X \in F^+$ (d. h. X^* ist Schlüsselkandidat)
$\Rightarrow$ D ist verbundtreue und verlustlose Zerlegung von R

Im Beispiel:

X = {Doznr, ... , Veransttyp}
X_1 = {Doznr, Name, Vorname, Tel}
X_2 = {Veranstnr, Veransttyp, Doznr}
R_1 = (X_1, F_1) mit F_1 = {{Doznr} $\rightarrow$ X_1}
R_2 = (X_2, F_2) mit F_2 = {{Veranstnr} $\rightarrow$ Veransttyp}

$X = X_1 \cup X_2$
Existiert ein $X^* \subseteq X$ mit $X^* \subseteq X_i$ für i = 1 oder i = 2

und $X^* \rightarrow X \in F^+$?

$X^* = \{Veranstnr, Doznr\} \subseteq X_2 \wedge X^* \rightarrow X$, da Schlüssel, existiert in F, also insbesondere in F^+

$\Rightarrow$ Die Zerlegung ist verbundstreu!

Auch für Verbund und Selektion gelten einige Rechenregeln.
Ist C eine komplexe Selektionsbedingung, X ein Relationenformat, $X = \{A_1, \ldots, A_n\}$, so bezeichnet attr (C) die Menge aller Attribute, die in C vorkommen!

Beispiel:

$X = \{A, B, D, E, F\}$
$C = A > 0 \wedge (B < D \vee F \neq 1)$
$\Rightarrow$ attr (C) = {A, B, D, F}

Satz 4:

Seien $r \in$ Rel (X), $s \in$ Rel (Y) Relationen,
C eine Komplexe Selektionsbedingung.
Dann gilt:

i) attr (C) $\subseteq X \Rightarrow \sigma_C$ (r join s) = σ_C (r) join s
(d. h. wirkt C nur auf die Attribute einer der beteiligten Relationen, kann man die Selektion bzgl. dieser Relationen aus dem Verbund herausziehen!)

ii) attr (C) $\subseteq X \cap Y \Rightarrow \sigma_C$ (r join s) = σ_C (r) join σ_C (s)
(d. h. enthält C nur Attribute, die in beiden Formaten vorkommen, kann man Selektion und Verbund vertauschen!

Beispiele zu i) und ii):

1) $X_1 = \{A, B, C\}$; $X_2 = \{B, C, D, E\}$

r_1 =

A	B	C
0	0	0
1	0	1
1	1	1

r_2 =

B	C	D	E
1	1	1	1
0	1	0	1
1	0	1	0

φ: (A > B); attr (φ) = {A, B} $\subseteq X_1$

$\Rightarrow \sigma_\varphi$ (r_1 join r_2) = σ_φ

A	B	C	D	E
1	0	1	0	1
1	1	1	1	1

=

A	B	C	D	E
1	0	1	0	1

$\sigma_\varphi (r_1)$ join r_2 =

A	B	C
1	0	1

join r_2 =

A	B	C	D	E
1	0	1	0	1

↑

auf r_2 läßt sich φ nicht anwenden!

d. h. $\sigma_\varphi (r_1)$ join $r_2) = \sigma_\varphi (r_1)$, <u>falls attr ($\varphi$) $\subset X_1$</u>

2) X_1, X_2, r_1, r_2, wie in 1)
φ : ($B \neq C \wedge C > 0$); attr (φ) = {B, C} $\subseteq X_1 \cap X_2$

$\Rightarrow \sigma_\varphi (r_1$ join $r_2) = \sigma_\varphi$

A	B	C	D	E
1	0	1	0	1
1	1	1	1	1

) =

A	B	C	D	E
1	0	1	0	1

$\sigma_\varphi (r_1)$ join $\sigma_\varphi (r_2)$ =

A	B	C
1	0	1

join

B	C	D	E
0	1	0	1

=

A	B	C	D	E
1	0	1	0	1

d. h. $\sigma_\varphi (r_1$ join $r_2) = \sigma_\varphi (r_1)$ join $\sigma_\varphi (r_2)$, falls attr (φ) $\subseteq X_1 \cap X_2$

I.7 Zusammenfassung

Aus bisherigen Überlegungen zum Relationalen Datenbankmodell kann man die folgenden **Gütekriterien** herleiten:

- <u>(Minimalität)</u> Das DB-Schema sollte so wenig Relationenschemata wie möglich enthalten!
- <u>(Normalisierung)</u> Zur Vermeidung von update-Anomalien sollte das DB-Design soweit wie möglich normalisiert sein!
- <u>(Informationserhaltung)</u> Das DB-Schema sollte verbundtreu sein!
- <u>(semantische Korrektheit)</u> Das DB-Schema sollte alle intra- und interrelationalen Abhängigkeiten der Ausgangssituation enthalten.

Diese Gütekriterien und die Forderung nach guter Performance sind im realen Anwendungsfall nicht immer alle gleichzeitig 100 % zu erfüllen. (Normalisierung geht auf Kosten der Minimalität, usw...)
Ist dies der Fall, muß mit dem Auftraggeber verhandelt werden, an welchem dieser Kriterien Abstriche gemacht werden können.

Anmerkung 1:

Alternativ zur Entwicklung einer RDM über ein ERM gibt es noch folgendes Verfahren zur Bestimmung eines RDM:

1. Sammle alle Attribute der gesamten Ausgangssituation in einem einzigen Format X, ohne Berücksichtigung, welchen Objekttypen diese zugeordnet sind.
2. Sammle alle funktionalen Abhängigkeiten zwischen Attributen aus dem Format X aus 1. in einer Menge F.
3. Sammle alle übrigen Datenabhängigkeiten in einer Menge G.
4. Zerlege dieses Universalschema sukzessiv durch Schritte, die Normalisierung und Verbundtreue garantieren.
5. Ergebnis: ein relationales DB-Schema $D = (R, \Sigma_R)$;
   $R = \{R_1, ... , R_p\}$; $R_i = (X_i, \Sigma_{Xi})$

Zur Realisierung von Schritt 4 gibt es in der Literatur eine Reihe von Beispielalgorithmen: DEKOMPOSITION oder SYNTHESE, die automatisch aus den Vorgaben von Schritt 1-3 ein relationales Datenbankschema in 3NF erzeugen, das verbundtreu ist!
Solche Algorithmen werden in CASE-Tools eingesetzt, die automatisch aus den Vorgaben eine relationale Datenbank (-Anwendung) erstellen!

Anmerkung 2:

Neben der Verbundtreue gibt es auch noch ein weiteres Gütekriterium für Relationale DB-Schemata, das auch Bezug nimmt auf die Zerlegung vorgegebener Relationenschemata:
das Kriterium der Unabhängigkeit der Zerlegung. Dieses bezieht sich auf die durch die Zerlegung eines Relationenschemas implizit auch mitvollzogene Zerlegung der Menge der funktionalen Abhängigkeiten des ursprünglichen Schemas:
ist $R = (X, \Sigma_X)$ ein Relationenschema und ist $F \subseteq \Sigma_X$ die Menge der funktionalen Abhängigkeiten bzgl. R, so wird durch die Zerlegung
$R = R_1, ... , R_p$ mit $R_i = (X_i, \Sigma_{Xi})$ und $X = X_1 \cup ... \cup X_p$
implizit auch die Menge F zerlegt in $F_1, ... , F_p$ mit $F_i \subseteq \Sigma_{Xi}$!

F bzw. die F_i enthalten die Semantik der Ausgangssituation und die Forderung ist also, daß nicht nur Datenverluste vermieden werden (im Fall der Verbundtreue), sondern daß durch die Aufsplittung von F keine Verluste der Semantik auftreten dürfen.
Das bedeutet, daß die Vereinigung der aufgesplitteten Mengen F_i wieder die komplette Ausgangssemantik enthalten muß!
Dieses Kriterium der Unabhängigkeit von Zerlegungen ist kompliziert (benutzt den komplizierten Begriff der abgeleiteten Funktionalen Abhängigkeit). Wegen dieser Komplexität kann dieses Kriterium hier nicht besprochen werden! Man sollte aber sensibel dafür sein, daß es dieses Problem bzgl. Schema-Zerlegung gibt!

Kapitel II Structured Query Language SQL

Für relationale Datenbanksysteme gibt es nur eine einzige Kommunikations-möglichkeit, die Abfragesprache SQL. SQL ist eine Sprache mit den folgenden Eigenschaften:

- SQL ist deskriptiv, d.h. man beschreibt das Ausführungsergebnis, nicht den Algorithmus, der zum Ergebnis führen soll;
- SQL operiert auf Mengen (Relationen, Tabellen);
- Das Ergebnis eines SQL-Statements ist wieder eine Menge (Relation);
- SQL-Statements können ineinander geschachtelt sein;
- SQL ist abgeschlossen, d.h. das Resultat eines SQL-Statements ist wieder ein relationales Datenbankobjekt
- SQL-Abfragen sind optimierbar (nach bestimmten Regeln).

SQL besteht aus 2 Komponenten: der Data Definition Language (DDL) und der Data Manipulation Language (DML).

Mit den Statements der DDL (create, alter, drop, grant, revoke) werden Datenbankobjekte (Tabelle, Indizes, Views,...) erzeugt, verändert, gelöscht bzw. Rechte gesetzt oder zurückgenommen.
Die Statements der DML-Komponente (insert, update, delete, select) dienen dazu, Datensätze (Tupel) zu erzeugen, zu verändern, zu löschen oder anzuzeigen.

SQL ist standardisiert und dieser Standart wird kontinuierlich weiterentwickelt. Natürlich fügt jeder Datenbankhersteller hauseigene Zusätze oder Änderungen zum Standard hinzu, die es zu beachten gilt, zumal wenn in einem Unternehmen Datenbanken mehrerer Hersteller im Einsatz sind!. Wir werden hier zunächst die Standard-Syntax vorgeben und dann Spezifika des hier im Einsatz befindlichen Datenbanksystems der Firma ORACLE benennen.

Beginnen wir mit der DDL-Komponente:

II.1 Data Definition Language - Statements:

Standard:

Im Standard vorgegeben sind die folgenden Statements:

create table	create view
alter table	
drop table	drop view

Syntax:

create table <name> (<Attributliste>);
<Attributliste> ::= <Attributname> <Datentyp> [Attribut constraint]
[, <Attributname>...]

alter table <name> <Attributänderung> | <constraint-Änderung>;
<Attributänderung> ::= add <Attributname> <Datentyp>
alter <Attributname> set
<Attributdatentypänderung>
drop <Attributname>

<constraint-Änderung> :: = add <constraint definition>
drop constraint <constraintname>

Drop table <name>;

create view <viewname> [(Attributkommaliste)]
as <select-statement>

drop view <viewname>

Oracle-spezifisch:

Beispiel: create table <name> <Attributkommaliste>
[storage-Klausel]

Storage-Klausel:

Die storage-Klausel dient dazu, zur erzeugten Tabelle den optimal konfigurierten Speicherplatz zuzuweisen.

Syntax:

storage (initial N K next M K minextents X maxextents Y pctincrease n)

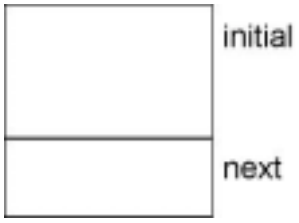
Dazu:

Datenbank-Objekte werden als Folge von Extents gespeichert. Dazu muß bei der Erzeugung die Anzahl von Extents,die beim Anlegen erzeugt werden sollen (minextents), deren Größe (initial), die Größe der Folge-Extents (next), die Maximalzahl von Extents (maxextents) und deren prozentuales Wachstum (pctincrease) angegeben werden.

Beispiel:

Create table kunde (kdnr number, name varchar2(30), gebdat date)
Storage(initial 500 K next 100 K minextents 2 maxextents 200 pctincrease 0);

Legt eine Tabelle Kunde so an, daß nach Erzeugung zwei Extents mit 500K und 100K zur Aufnahme von Datensätzen zur Verfügung stehen:



Sind die angelegten Extents gefüllt, wächst die Tabelle automatisch sukzessiv durch weitere extents, bis insgesamt maxextents Extents verbraucht sind.

Wird keine storage-Klausel angegeben, werden Defaultwerte genommen (plattformabhängig!):

Der zu verbrauchende Speicherplatz sollte pro Tabelle berechnet werden, um Verschwendung von Speicherplatz zu vermeiden.
Für buchhalterische Anwendungen könnte man folgenden Ansatz wählen:

initial = Datensatzgröße ∗ (Anzahl der Datensätze im nächsten Jahr)
next = Datensatzgröße ∗ (mittlere Anzahl Datensätze in den nächsten 5 Jahren)

Zum Arbeiten mit dem ORACLE-Datenbanksystem gebe ich auf den folgenden Seiten beispielhaft die Syntax für gängige DDL-Statements an und auch die Sichten auf Systemtabellen, über die sich die einzelnen user Informationen über die ihnen gehörenden Datenbankobjekte selektieren können:

- **create table** <tabellenname> (<Attributliste>)
  [tablespace <Tablespacename>]
  [storage (initial <Größe> next <Größe> minextents <#> maxextents <#>
  pctincrease 0)];

  Attributliste: <Attributname> <Datentyp> [,<Attributname> <Datentyp>]

  alter table <tabellenname>
  [add / modify (Attributliste)]
  [add constraint <constraintname> <constraint-Spezifikation>]
  [drop constraint <constraintname>]

  drop table <Tabellenname>;

- **constraint-Spezifikation**:
  primary key (<Attributkommaliste>) |
  foreign key (<Attrib.kommaliste<) references <Tabellenname>
  (<Attrib.kommaliste>)
  [on delete cascade] |
  check (<Attributname> <Attributbedingung>)
  z.B. check (geschl in ('M','W','m','w'))

- **insert into** <tabellenname> [(Attribut-Komma-Liste)]
  values (Attributwerte-Liste);

- **update** <Tabellenname>
  set <Attributname> = <Wert>
  [where <Bedingung>];

- **delete** from <tabellenname>
  [where <Bedingung>];

- **select** <Attribut-Komma-Liste> from <Tabellen-Komma-Liste>
  [where <Bedingung>]
  [order by <Attribut-Komma-Liste>]

- **create sequence** <sequencename>
  [increment by <Größe>]
  [start with <Größe>];

  alter sequence <Sequencename> increment by <Größe>;

  drop sequence <Sequencename>;

- **alter user** <Username>
  [identified by <Password>]

[default tablespace <Tablespacename] [temporary tablespace <Tablespacename]
[quota <Größe> | unlimited on <Tablespacename>];

- **alter session** set nls_date_format = 'dd.mm.yyyy';

- **grant** all | select | insert | update | delete on <Tabellenname> to <username>;

- **create [public] synonym** <Synonymname> for <Objektname>;
  drop [public] synonym <Synonymname>;

- **create view** <viewname> as <select-Statement>;
  drop view <viewname>;

- **create index** <Indexname> on <Tabellenname> (Attributliste)
  [tablespace <Tablespacename>]
  [storage (storage-Klausel)];
  alter index <Indexname> storage (storage-Klausel);
  drop index <Indexname>;

- **create role** <Rollenname> [identified by <password>];
  alter role <Rollenname> identified by <password>;
  drop role <Rollenname>;

Dictionary-Sichten für user:

---

user_catalog	Informationen über Tabellen, views, synonyme, sequences
user_constraints	Informationen über Definitionen der constraints
user_cons_columns	welche Spalten in welchem constraint
user_dependencies	Informationen über gemachte Referenzen
user_free_space	wieviel Speicherplatz ist noch im Zugriff
user_indexes	Informationen über angelegte Indices
user_ind_columns	welche Spalten in welchem Index
user_objects	Informationen über alle vom user angelegte Objekte
user_segments	Informationen über vom user angelegte Segmente
user_sequences	Informationen über angelegte sequences
user_source	Informationen über stored procedures/functions
user_tables	Informationen über angelegte Tabellen
user_tablespaces	Beschreibung zugreifbarer Tablespaces
user_tab_columns	welche Spalten in welcher Tabelle
user_triggers	Informationen über angelegte Datenbanktrigger
user_ts_quotas	Tablespace-Quoten des users
user_views	view-Definitionen des users

DD-Sichten bzgl Rechte:

user_col_privs	Rechte auf Spalten als Besitzer, grantor oder grantee
user_col_privs_made	vergebene Rechte auf Spalten
user_col_privs_recd	erhaltene Rechte auf Spalten
user_role_privs	Rollen, die dem user gegrantet sind
role_sys_privs	Systemrechte der Rolle
role_tab_privs	Tabellenrechte der Rolle
user_sys_privs	Systemrechte des users, direkt gegeben
user_tab_privs	Objektrechte des users, direkt vergeben
user_tab_privs_made	vergebene Rechte auf Tabellen
user_tab_privs_recd	erhaltene Rechte auf Tabellen

Zu den DDL-Komandos: create, alter, drop, grant, revoke, gibt es im wesentlichen die folgenden **ORACLE-spezifischen Datenbankstrukturen**:

	Create	drop	Alter	grant	revoke
Database	•		•		
Index	•	•	(•) stroage-Klausel		
Function	•	•			
Procedure	•	•			
Role	•	•	(•) nur Passwort ändern	•	•
Sequence	•	•	•		
Synonym	•	•			
Table	•	•	•		
Trigger	•	•	(•) nur enable, disable		
User	•	•	•		
View	•	•			
Objektrecht				•	•
Systemrecht				•	•

zu den einzelnen Objekttypen:

Database:

ist die umfassende Struktur, die alle Objekte enthält. Bei Installation der Software muß eine Datenbank mit dem create database-Statement erzeugt und kann mit alter database modifiziert werden (im Aufgabenbereich der Administration / Management).

role:

Jeder Benutzer der Datenbank muß Rechte (Systemrechte und Objektrechte) haben. Wird ein neuer Benutzer mit dem create-Statement erzeugt, hat er so erstmal keine Rechte, er kann nicht einmal ein connect machen. Dazu braucht er das 'create-session'-Systemrecht. Dann kann er eine Sitzung aufmachen, darf aber keine Tabellen oder ähnliches erzeugen, hat keine Ressourcen auf irgendeinen Tablespace usw.
Meist ist es so, daß in einem Unternehmen viele Nutzer im Prinzip gleiche Rechte brauchen:
create session, create table, create synonym, create sequence
Damit nicht jeder Nutzer immer wieder alle Rechte einzeln zugeordnet (gegranted) werden müssen, können verschiedene Rechte in sogenannten Rollen zusammengefaßt werden. Eine Rolle ist also eine Menge (ein Container) von Rechten. Diese kann als solche einem Nutzer zugeordnet werden.

user:

Alle Benutzer einer DB müssen der DB bekannt sein und werden mit gewissen Rechten ausgestattet. Damit werden Sicherheitsmechanismen installiert, die den Zugang zur DB regeln und den Zugriff auf interne Objekte!

Einem user können von anderen usern Zugriffsrechte auf ihre Objekte gewährt werden: lesen, löschen, neu einfügen, ... usw.
Aus der Sicht der Objekte unterteilt eine DB diese also bzgl. der user:
welche Objekte gehören dem user
auf welche Objekte kann der user wie zugreifen
auf welche Objekte kann der user nicht zugreifen
user können auch gelöscht werden. Dann sollten alle Objekte, die ihm gehören, entweder mit gelöscht oder einem anderen user zugeordnet werden!

tables:

entsprechen den Objekttypen (Relationen) der realen Situation,
die Spalten der Tabelle sind die Attribute.

Indizes:

Problem:
bei allen Tabellen mit langen Datensätzen und vielen Tupeln kann es sein, daß zur Suche nicht die komplette Tabelle ins Memory geladen werden kann, sondern sukzessiv in Teilen bearbeitet werden muß. Das kostet Zeit.
Deshalb kann man eine neue Tabelle anlegen, die nur aus den ausgewählten Spalten der Ursprungstabelle besteht (z.B. den Spalten, nach denen in der where-Klausel oft gesucht wird), und einem Verweis auf den entsprechenden Datensatz der Ursprungstabelle. Solche zusätzlich erzeugten Tabellen heißen Indexe oder Indizes.
Beispiel:
Produkt sei eine Tabelle, deren Datensätze 1 KB lang sind! es gibt 100 000 Einträge in der Tabelle, d. h. Produkt ist ca. 100 MB groß. Als Speicher stehen 30 MB zur Verfügung, d. h. die Tabelle kann zum Suchen nach bestimmten Datensätzen nicht komplett ins Memory geladen werden.
Folge:
Es werden 1 oder mehrere Indizes angelegt: einer bzgl. der Schlüsselspalte (Produktnr), usw. Produktnr hat nur 10 Bytes
$\Rightarrow$ Dieser Index braucht nur ca. $10 * 100\,000 = 1$ MB Memory!
Indizes sind also spezielle Suchtabellen und werden auch so angelegt (mit Verweis auf die Ursprungstabelle).

views:

views sind spezielle Sichten auf eine oder mehrere Tabellen. Es könnte zum Beispiel die Produktpalette des Unternehmens im Internet angeboten werden. Dabei sind aber viele Attribute wie z. B. Produktnr oder andere firmeninterne Spalten (Herstellungsdatum, Stückzahl,...) für die potentiellen Kunden uninteressant. Damit man nun nicht neue Tabellen anlegen muß, definiert man einen view, der aus genau den Spalten der Tabelle besteht, die man zeigen will. Man kann views auch benutzten, um Spalte mehrerer Tabellen (über einen join) als eine (Schein-)Tabelle anzubieten.
Eine weitere Absicht bei der Verwendung von views kann sein, daß gewissen Anwendern Zugriffsbeschränkungen auf Datenbestände auferlegt werden,

einfach dadurch, daß er nicht komplette Tabellen, sondern nur ausgewählte Spalten und Zeilen in einem view zusammengestellt bekommt!

Synonyme:

Alle Tabellen, Indizis, views, usw. werden von bestimmten usern angelegt und gehören diesen. Der volle Name der Tabelle Produkte, die von der Administratorin Maier angelegt wurde, ist maier.produkte. Es kann also auch eine Tabelle Produkte geben, die der Sachbearbeiter Zimmerman angelegt hat, eben zimmerman.produkte. Das kann in einem Unternehmen zu großen Verwirrungen führen. Diese Verwirrungen können nicht auftreten, wenn der Administrator sofort nach Anlage seiner Tabelle maier.produkte ein sogenanntes public synonym mit dem Namen Produkte bzgl. der Tabelle maier.produkt erzeugt hat. Damit gibt es ein Datenbankobjekt mit dem datenbankweit gültigen Namen ‚Produkte'.

Gibt Maier jetzt einem dritten user Schmidt das Recht, auf seine Tabelle zuzugreifen, so kann der das, indem er z. B. select prodnr from produkte eingibt. Der Sachbearbeiter Zimmerman kann zwar auch dem user Schmidt Zugriffsrechte auf seiner Tabelle zimmerman.produkte granten, kann aber kein public synonym mehr anlegen und Schmidt muß die Tabelle von Zimmerman explizit mit zimmerman.produkte ansprechen. Synonyme bestimmen also den globalen Charakter lokaler Objekte!

sequences:

Sequences sind Generatoren von fortlaufenden Numerierungen, die für Primärschlüssel gebraucht werden können. Hat z. B. die Tabelle Produkte ein Attribut Produkt.nummer, das einfach eine laufende Nummer mit einer bestimmten Schrittweite ist, so kann man eine sequence erzeugen, die genau diese Nummern mit der Schrittweite erzeugt.
Diese sequence wird immer dann aufgerufen, wenn ein neuer Datensatz (ein neues Produkt) in die Tabelle Produkte eingelesen wird. Der Vorteil ist, daß hier die Datenbank die Vergabe der Nummern kontrolliert! Jede Nummer wird nur genau einmal vergeben und es kann nie passieren, daß zwei Benutzer in einer Applikation verschiedene Datensätze mit der gleichen Nummer erzeugen!

stored procedures / functions:

Ab der Version 7.x bietet ORACLE die Möglichkeit, Funktionen und Prozeduren der Prozeduralen Erweiterung PL/SQL der Sprache SQL persistent und kompiliert in der Datenbank zu speichern! Der Vorteil ist, daß die Benutzung schon kompilierter Programme wesentlich performanter ist und zum anderen, daß solche Programmodule auch von anderen Applikationen benutzt werden können (Vermeidung von Redundanzen von Programmen!). Mehr davon später.

Datenbank-Trigger:

Trigger sind Ereignis-gesteuerte Programme; d. h. tritt ein bestimmtes Ereignis ein (etwa update eines Datensatzes einer Tabelle), so wird der Trigger aktiv und das Programm, das er enthält, wird ausgeführt.
Datenbank-Trigger sind Schalter, die in der Datenbank bzgl. bestimmter Tabellen abgelegt werden. Es gibt insert-, update- und delete-Datenbank-

Trigger. Soll also beim Ändern der Tabelle Produkte (Anlage eines neuen Datensatzes, Ändern eines bestehenden Datensatzes, Löschen von Datensätzen) z. B. eine Notiz in einer Protokoll-Tabelle geschrieben werden (wer hat wann was wie geändert bzgl. der Tabelle Produkte), so werden die drei DB-Trigger für die Tabelle Produkte erzeugt und die PL/SQL-Anweisungen zum Eintrag in die Protokolltabelle eingehängt.
Auch dazu später mehr!

Neben diesen Objekten, die explizit über die DDL-Statements erzeugt, gelöscht oder geändert werden, gibt es weitere Objekte, die in der Datenbank bereits definiert sind (als leeres Blatt) und verschiedenen anderen Objekten zugeordnet werden können.

Constraints

Eine wichtige Gruppe solcher Objekte sind die Integrity-Constraints, die in der Regel den Tabellen einer Anwendung zugewiesen werden, entweder im create-table-Statement oder besser nachträglich im alter-table-Statement!

Integrity-Constraints sind Regeln, die aus der Ausgangssituation stammen und denen die Daten in einer Tabelle genügen müssen, um sinnvoll zu sein.

Beispiele:
Ein Preis eines Produkts ist immer > 0,
Kundennummer soll Primärschlüssel sein,
usw....

Es gibt bei Oracle die folgenden Arten von Tabellen- bzw. Spalten-Constraints:

- NOT NULL (NULL)	Spalten/Tabellen-Constraint
- UNIQUE	"
- PRIMARY KEY	"
- FOREIGN KEY	nur Tabellen-Constraint
- CHECK	"

1. NOT NULL

Syntax:
alter table <name> modify <spalte> [constraint <name>] NOT NULL;

Beispiel:
Alter table Kunde modify gebdat gebdat_nn NOT NULL;

Damit wird für jeden Datensatz der Tabelle Kunde ein Wert für das Attribut gebdat erzwungen.
Enthält ein insert in Kunde keinen Wert für gebdat, wird eine Fehlermeldung ausgegeben.

Das NOT NULL-Constraint kann auch als Spalten-Constraint direkt bei create- oder alter table-Statements angegeben werden:

create table Kunde
(name, varchar 2 (40) **NOT NULL**,....);

Nachteil:
Da vom Erzeuger kein Name angegeben wurde, wird dem constraint vom System ein kryptographischer String als Name gegeben! Das bedeutet, daß bei Verletzung dieses Constraints vom System eine Meldung zurückkommt: 'das constraint $$$$xxxx wurde verletzt. Da man den Namen $$$$xxxx nicht identifizieren kann, kann man auch die constraint-Verletzung nur schwer lokalisieren! Deshalb sollte man immer constraints mit eigenen, sprechenden Namen versehen!

2. UNIQUE

Dieses Constraint definiert eine Spalte oder eine Menge von Spalten als eindeutig! d. h. keine zwei Zeilen einer Tabelle können denselben Wert bzgl. dieser Spalte (n) haben.

Ist eine Spalte unique, kann sie NULL-Werte enthalten!

Ausgeschlossen sind Spalten mit den Datentypen LONG oder LONG RAW. Auch UNIQUE kann als Spalte- oder Tabellen-Contraint erzeugt werden:

create table Kunde (Kundennr number (4) **unique**);
oder
alter table Kunde add constraint Kundennr_u **UNIQUE** (Kundennr);
oder
alter table Kunde add constraint Kunde_u **UNIQUE** (Name, Vorname, Gebdat);

3. PRIMARY KEY

Dieses Constraint setzt die angegebenen Attribute als Primärschlüssel der Tabelle. Sie bekommen damit die Eigenschaften

- unique
- Not Null

Zu jeder Tabelle kann höchstens ein Primary Key-Constraint deklariert werden!

a) als Spalten-Constraint erzeugen:
create table <name> (<attr.name> <datentyp> **primary key**, ...)

Beispiel:
create table kunde (kdnr number (8) primary key, name varchar 2 (40), ...);

b) als Tabellen-Constraint:
alter table <name> add constraint <name> **primary key** (<attr. name>);

Beispiel:
alter table Kunde add constraint pk_kunde primary key (kdnr);

Bei Tabellen-Constraints können auch zusammengesetzte Primärschlüssel deklariert werden:
alter table positionen add constraint pk_pos primary key (anr, prodnr);

Bei der Deklaration eines Primary Key-Constraints wird automatisch ein Index angelegt!

4. FOREIGN KEY

Durch dieses Constraint werden Tabellen zueinander in Beziehung gesetzt.

Syntax:
alter table <name> add constraint <name> **foreign key** (<attr.kommaliste>) references <tabellenname> (<attr.kommaliste>) [on delete cascade];

Beispiel:
alter table adresse add constraint fk_kun_adr **foreign key** (kdnr) references kunde (kdnr);

Hier wird die kdnr, die in der Tabelle Kunde Primärschlüssel ist, in der Tabelle Adresse als Fremdschlüssel deklariert.

Eigenschaften:
- Vor dem Erzeugen eines Fremdschlüssels-Constraints muß der darin referenzierte Primärschlüssel erzeugt sein!
- Fremdschlüssel können zusammengesetzt sein.
- Null Werte sind erlaubt!

Wird ein Foreign Key-Constraint ohne die "**on delete cascade**"-Option deklariert, können Masterdatensätze nur gelöscht werden, nachdem zuvor alle davon abhängigen Datensätze in anderen Tabellen gelöscht wurden!
Wird die Option "on delete cascade" gesetzt, werden beim Löschen eines Masterdatensatzes automatisch alle abhängigen Datensätze mitgelöscht!

5. CHECK-CONSTRAINT

Syntax:
check (<Bedingung>)

Beispiele:
i) alter table kunde add constraint c_geschlecht

check (geschlecht in ('w', 'm', 'W', 'M'));

ii) alter table produkt add constraint c.preis
check (preis > 0);

iii) alter table mitarbeiter add constraint c.prämie
check (gehalt + prämie ≤ 7 500DM AND gehalt > prämie)

Check-Constraints können in der deklarierten Bedingung nur Bezug nehmen auf die Attribute der Tabelle, bzgl. der sie deklariert sind, nicht Tabellen-übergreifend!!

Constraints sind sofort nach Erzeugung aktiv, auch rückwirkend.
Sind also bereits Daten in der Tabelle vorhanden, kann das Constraint nur dann erzeugt werden, wenn die vorhandenen Daten es nicht verletzen!
Constraints können gelöscht, aktiviert oder deaktiviert werden:

- löschen: alter table <name> drop constraint <name>;
- deaktivieren: alter table <name> disable constraint <name>;
- aktivieren: alter table <name> enable constraint <name>;

Anmerkung:

Die Check-Bedingung kann in der dictionary-Sicht *user_constraints* im Attribut *search condition* nachgesehen werden!

Rechte

Neben den Constraints, die garantieren, daß keine "falschen" Daten in die Datenbank kommen, gibt es eine weitere Gruppe von Objekten, die für die Sicherheit und den guten Zustand einer DB notwendig sind und den Benutzern überhaupt erst die Möglichkeit von erfolgreichen DDL- und DML-Statements eröffnen: **Rechte**!

Die Rechte-Verteilung:
Man unterscheidet sogenannte System- und Objektrechte.
Die konkreten Rechte und ihre Vergabemöglichkeiten sind natürlich DB-System spezifisch!
Oracle bietet ca. 80 Systemrechte und 8 Objektrechte an.
Objektrechte sind:

Privilege	**Table**	**View**	**Sequence**	**Procedure Function**
alter	•		•	
delete	•	•		
execute				•
index	•			
references	•			
select	•	•	•	
update	•	•		
insert	•	•		

Speziell für Tabellen sind also folgende Operationen zu autorisieren:

alter:	erlaubt dem Autorisierten,	die Tabellendefinition mit alter table zu ändern
delete:	erlaubt dem Autorisierten,	Datensätze zu löschen
index:	erlaubt dem Autorisierten,	einen Index auf der Tabelle mit create index ... zu erzeugen
insert:	erlaubt dem Autorisierten,	Datensätze einzufügen
references:	erlaubt dem Autorisierten,	ein FK-Constraint zu erzeugen, das auf diese Tabelle refereziert
select:	erlaubt dem Autorisierten,	Anfragen an die Tabelle zu stellen
update:	erlaubt dem Autorisierten,	Veränderungen von Daten in der Tabelle vorzunehmen

Die Syntax zur Vergabe von Objektrechten ist:

```
grant  ALL | <Objektrechtliste> on <objektname>
to <role | user | public> [with grant option]
```

Rechte auf Objekte können nur die Besitzer der Objekte vergeben!
Ausnahme:

Jemand hat ein Objektrecht verliehen bekommen mit der Option "with grant option"!

Wird ein Objekt nicht an einen user oder eine Rolle, sondern an Public verliehen, steht dieses Recht allen derzeitigen und zukünftigen usern zur Verfügung (also Vorsicht!)
Alle Objektrechte können auch wieder zurückgenommen werden mit dem Befehl:

```
revoke All | <Objektrechtliste> on <objektname>
from <user | role | public> [cascade constraints]
```

Die Bedeutung ist klar bis auf die Option "cascade constraints":

Es kann user Müller das Objektrecht "references" auf die Tabelle paul.KUNDE zugewiesen bekommen haben, die also dem user Paul gehört.
Müller legt jetzt eine Tabelle Adresse an und einen FK (Kdnr) in dieser Tabelle, der die Kundennummer in der Tabelle KUNDE referenziert.

Nun möchte Paul dem Müller das Referenzier-Recht auf KUNDE wieder wegnehmen.
Aus Konsistenzgründen muß er auch die Möglichkeit haben zu sagen, daß auch alle bisher erzeugten Referenzen über FK's weggenommen werden müssen! Das geschieht mit dem Zusatz "cascade constraints"!

Neben diesen acht Objektrechten gibt es ca. 80 **Systemrechte**, die jedem Benutzer je nach Bedürfnissen zugeteilt werden müssen (explizit oder über Rollen!)

Syntax:

grant <systempriv> to <user | role| public> [with admin option]

die "with admin option" besagt, daß der Autorisierte dieses Recht weiter granten kann.

Wegnahme:

revoke <systempriv> from <user | role | public>

Besonderheiten:

- Alter table ist kein System- sondern ein Objektrecht!
- Create-Index-Recht auf einer fremden Tabelle wird als Objektrecht verliehen!
- Jedes Objektrecht erlaubt dem Autorisierten, die Tabelle mit einem lock-table-Statement zu sperren (das select-Recht reicht also schon!)

Systemrechte werden in der Regel vom DB-Administrator vergeben, wir brauchen darauf jetzt noch nicht einzugehen!
Wichtiger für uns zur momentanen Arbeit sind die Objektrechte!

II.2 Data Manipulation Language - Statements

Neben den DDL-Statements zum Erzeugen, Ändern und Löschen von DB-Objekten (Tabellen, Indizes,...) muß es auch Statements geben zum Füllen der DB-Objekte, zum Ändern und Löschen dieser Daten. Dies sind die Statements der DML-Komponente von SQL.

Als DML-Statements stehen also standardisiert zur Verfügung:

- insert
- update
- delete
- select

Wir beginnen mit dem Statement mit dem die DB-Objekte abgefragt werden können, dem **select-Statement**. (Grund: in den anderen DML-Statements kann so ein select-Statement eingebaut werden!)

Zweck des select-Statements ist die Suche und Darstellung von Datensätzen (Tupeln) aus DB-Tabellen.

Formaler Aufbau (Standard):

- select ?
- from ?
- [where ?]
- [group by ?]
- [having ?]
- [order by ?] ;

Ein select-Statement kann also aus bis zu sechs Klauseln aufgebaut sein! Dabei ist die Bedeutung der einzelnen Klauseln wie folgt:

Select-Klausel:	definiert das Layout der Ergebnistabelle
from-Klausel:	definiert die Quelle (n) aus denen die Ergebnistabelle zusammengestellt wird.
Where-Klausel:	definiert Kriterien, denen die Datensätze der Ergebnistabelle genügen müssen
Group-by-Klausel:	definiert eine Gruppierung der Attribute bzw. ihrer Werte in der Ergebnistabelle
Having-Klausel:	definiert besondere Bedingungen, die bzgl. der Gruppierung gelten müssen
order-by-Klausel:	definiert die Sortierreihenfolge in der Ergebnistabelle

Ein select-Statement muß eine select- und eine from-Klausel enthalten, alle anderen Klauseln sind optional!

Beispiel:

```
select Kunde.kdnr, name, ort
from Kunde, Adresse
where Kunde.kdnr = adresse.kdnr
and geschlecht = "M";
```

Ergebnistabelle:

Eine Tabelle mit den Attributen kdnr, name, ort und
allen Datensätzen aus der Tabelle Kunde mit männlichen Kunden, die mindestens eine Adresse haben!

Wir werden im Folgenden die einzelnen Klauseln besprechen:

a) select-Klausel:

Standard-Syntax:

```
select [ALL | distinct] <select-item-commalist>
```

Erläuterung:

ALL ist default und bedeutet, daß alle Datensätze, die den in den anderen Klauseln spezifizierten Bedingungen genügen, in der Ergebnistabelle stehen, auch doppelt oder mehrfach.
Distinct bedeutet, daß in der Ergebnistabelle nur unterschiedliche Datensätze stehen.

Beispiel:

Sind in der Tabelle Kunde 20 Kunden mit Namen Müller, so werden durch
Select name from kunde where Name = ‚Müller‘;
20 Datensätze selektiert.
Durch
Select distinct name from Kunde where name = ‚Müller‘;
wird dagegen nur ein Datensatz selektiert!

<select-item-commalist> ist eine Auflistung durch Kommata getrennte sogenannter select-items.

Ein select-item hat die Form:

<scalar expression> [[AS] column] | ∗

∗ bedeutet:	alle Attribute aller beteiligten Quellen (Tabellen) sind Attribute der Ergebnistabelle
<scalar expression>:	kann ein Numerischer Ausdruck oder ein String-Ausdruck sein

Numerischer Ausdruck kann sein:

i) Attributname
ii) Parameter, host-variable, Konstante
iii) Skalare Funktion oder Aggregatfunktion
iv) Skalarer Ausdruck in Klammern
v) durch arithmetische Operationen +, –, ∗, / verknüpfte Elemente aus i) – iv).
wobei der Wert der Elemente i) – v) numerisch sein muß!

Ein string-Ausdruck kann sein:

gleiche Liste von Elementen i) – iv)
v) Konkatenation von Elementen aus i) – iv)
wobei hier der Wert der Elemente von i) – v) ein Character-String sein muß:

Zu klären ist noch, was skalare Funktionen bzw. Aggregatfunktionen sind.

Skalare Funktionen:

Arithmetische Operatoren: +, –, ∗, /

(Oracle hat eine Menge von mathematischen Funktionen hinzugefügt:
sin(x), cos(x), tan(x), abs(x), sqrt(x), power(x,y), exp(x), ln(x), usw.....)

char_length (bei oracle: length)
current_ date (bei oracle: sysdate)
current_time (bei oracle: keine Entsprechung)
current_user (bei oracle: user)
lower (string), upper (string)
substring (extrahiert Teilstrings); (bei Oracle:substr (string, Anfangspos, # Zeichen))

Auch hier sind u. U. weitere Funktionen von den DB-Herstellern hinzugefügt, das muß dann in den einzelnen Handbüchern nachgesehen werden!

Aggregatfunktionen:

Der SQL-Standard sieht fünf Aggregatfunktionen vor:

count (∗ | Attributname):	Anzahl der Datensätze der Quelle; ∗ : alle Attributname: nur die Datensätze, die im angegebenen Attribut keinen Null-Wert haben
sum (scalar expression):	Summe der Werte dieses Ausdrucks
Avg (scalar expression):	Mittelwert über alle Werte dieses Ausdrucks
Max (scalar expression):	Maximaler Wert über alle Werte dieses Ausdrucks
min (scalar expression):	Minimaler Wert über alle Werte dieses Ausdrucks

Zu den Aggregatfunktionen später mehr, vor allem im Zusammenhang mit der group by- bzw. having-Klausel!

Jedes select-item definiert eine Spalte der Ergebnistabelle. Diese Spalten können in der select-Klausel mit neuen Namen versehen werden über:

select <scalar expression> [as] <neuer Name>, ...

Beispiele:

select name from kunde;

Ergebnis: Name
alle Namen aller Kunden; 20 Meier $\Rightarrow$ 20 x Name Meier

select distinct name from kunde;

Ergebnis: Name
jeder Name genau 1 x

select name kundenname from kunde;

Ergebnis: Kundenname
alle Namen aller Kunden

select * from kunde;

Ergebnis: die Kundentabelle

select "Dies ist ein Kunde:" Text, name from kunde;

Ergebnis:

Text	Name
Dies ist ein Kunde:	Meier
Dies ist ein Kunde:	Schulze
Dies ist ein Kunde:	Schmidt
....	....

select "Dies ist der * || k.seq.nextual || "-te kunde" Text, name from kunde;

Ergebnis:

Text	Name
Dies ist der 1-te Kunde	Meier
Dies ist der 2-te Kunde	Schulze
....	....

Voraussetzung: vorher wurde die sequence k.seq erzeugt!

select substr (vorname,1,1) || ". " || name kunde from kunde;

Ergebnis:

Kunde
H. Meier
S. Schulze
....

select preis, (preis ∗ 0.1) + preis neupreis from produkt;

Ergebnis:

Preis	Neupreis
1.00	1.10
2.50	2.75
....	....

select sin(kdnr) ∗ length(name) quatsch from kunde;

Ergebnis:

quatsch
....
....

Bezüglich Aggregatfunktionen:

select max(preis) Maxpreis from produkt;

Ergebnis:

Maxpreis
24.30

select avg(preis) Mittlerer_preis from produkt;

Ergebnis:

Mittlerer_Preis
11.50

select count(∗) from kunde;

Ergebnis:

Count(∗)
15

⇐ Anzahl aller Datensätze in der Tabelle Kunde!

b) from-Klausel:

Syntax:

from <Quellenkommaliste>

<Quellenkommaliste> ::= <Quellenname> [<Alias>] [, ...]

A) Einfachster Fall: Quelle besteht nur aus einer Tabelle:
Dann braucht natürlich in der where-Klausel kein join-Kriterium angegeben werden:

Beispiel:
Select prodnr, preis, stckz from produkt;

Ergebnis:

Prodnr	Preis	Stckz
....	....	....
....	....	....
....	....	....

B) Wurden durch Normalisierung oder sonstwie die Informationen der Ausgangssituation auf mehrere Tabellen (Relationen) verteilt, kann man mit Hilfe eines joins solche verteilten Informationen wieder zusammen sammeln. Dazu müssen zwei Dinge getan werden:

i) in der from-Klausel die notwendigen Quellen angeben
ii) in der where-Klausel die notwendigen join-Kriterien angeben!

Diese join-Kriterien beziehen sich in der Regel auf die Primär-Fremdschlüssel-Beziehungen, wie sie in den Constraints zu den Tabellen definiert wurden!

Beispiel 1:

Gesucht: Name und Wohnort aller Kunden;
Methode: join über die Tabellen Kunde, Adresse längs Kdnr

Statement:
```
select name, ort from kunde, adresse
where kunde.kdnr = adresse.kdnr
```

So ist auch die standardisierte Syntax:
im join-Kriterium wird mit der Punkt-Notation navigiert.

Komplizierteres Beispiel:

Gesucht: Kdnr, Name und Produktbezeichnung aller Produkte, die von den Kunden bisher bestellt wurden.

benötigte Quellen: Tabelle Kunde: für Kdnr und Name
Tabelle Produkt: für die Produktbezeichnung

um auch wirklich nur die Produkte zu bekommen, die bisher bestellt wurden, müssen also die Produkte und Kunden in Beziehung gesetzt werden. Das geht nur durch Hinzunahme der Tabellen Auftrag (in der die Kdnr der Kunden steht, die was bestellt haben)
und Position (in der zu jeder Auftragsnr die bestellten Prodnr stehen)

Statement:
```
select kunde.kdnr, name, bez
from kunde, auftrag, position, produkt
where kunde.kdnr = auftrag.kdnr
and auftrag.anr = position.anr
and position.prodnr = produkt.prodnr;
```

Besonderheiten:

- Jedes Attribut in der select-Klausel muß eindeutig sein, d. h. gegebenenfalls muß der Tabellenname vorgestellt werden!
- Die Attribute, längs derer die join-Kriterien formuliert sind, müssen nicht in der select-Klausel stehen
- Wird bei mehr als einer Tabelle in der from-Klausel kein join-Kriterium in der where-Klausel angegeben, besteht die Ergebnistabelle aus dem **kartesischen Produkt aller Datensätze aller beteiligten Quellen!**
- Sind in Attributen, längs derer der join gebildet wird, NULL-Werte, werden die zugehörigen Datensätze nicht mit in die Ergebnismenge übernommen!

C) Outer Join:

Sollen auch die Datensätze, die in join-Attributen NULL-Werte haben, in der Ergebnistabelle aufgeführt werden, oder solche Datensätze, zu denen in der anderen Tabelle keinen Eintrag gibt, muß ein sogenannter outer join formuliert werden:
Syntax: ein " (+) " bei der Tabelle des join-Kriteriums, wo der Datensatz fehlt bzw. ein NULL-Wert steht.

Beispiel:

Die Tabelle Kunde enthält den Kunden Karl Hinz, der noch keine Adresse eingetragen hat

Statement:

```
select name, ort from kunde, adresse
where kunde.kdnr = adresse.kdnr
order by name;
```

liefert alle entsprechenden Kundendatensätze bis auf den Kunden Hinz!

Dagegen liefert:

```
select name, ort from kunde, adresse
where kunde.kdnr = adresse.kdnr (+)
order by name;
```

alle Kundendatensätze, mit oder ohne Ort bzw. Adresse, also auch den Kunden Hinz!

D) self-join

Unter Umständen ist es notwendig, in einem select-Statement auf die gleiche Tabelle mehrmals zuzugreifen, in dem Sinne, daß die gleiche Tabelle mehrfach als Quelle gebraucht wird. In so einem Fall wendet man den self-join an, der syntaktisch darin besteht, daß die fragliche Tabelle mehrfach in der from-Klausel aufgelistet wird, allerdings jedesmal mit einem anderen Aliasnamen:

Syntax:

from tabelle alias1, tabelle alias2

Beispiel:

Gesucht:	Kdnr und Ort aller Kunden, die jeweils im gleichen Ort wohnen
Bezug:	nur die Tabelle Adresse
Besonderheit:	für jeden Datensatz müssen die Orte mit anderen Datensätzen verglichen werden!
Statement:	select distinct a.ort, a.kdnr from adresse a, adresse b where a.ort = b.ort order by a.ort, a.kdnr;

E) Es gibt auch noch joins längs Attributen, die nicht mit dem Vergleichsoperator '=' arbeiten, sondern mit Ungleich-Operatoren '≤','<','≠',...
dies definiert sogenannte Theta-joins
Wer mehr über diese besondere Art join wissen will, sollte sich anhand der aktuellen Literatur informieren!

c) where-Klausel:

Die where-Klausel enthält allgemein die Suchkriterien, manchmal auch bedingte Ausdrücke (conditional expressions) genannt.

Syntax:

where conditional expression

mit:

Conditional expression :=	conditional term | conditional expression OR conditional term
Conditional term :=	conditional factor | conditional term AND conditional factor
Conditional factor :=	[NOT] conditional test
Conditional test :=	conditional primary [[NOT] TRUE | FALSE]
Conditional primary :=	simple condition | (conditional expression)
Simple condition :=	Vergleichs-Bedingung (comparisation condition) | between-Bedingung (between-condition) | like-Bedingung (like-condition) | in-Bedingung (in-condition) | match-Bedingung (match-condition) | exists-Bedingung (exists-condition)

Vereinfacht gesagt, besteht ein Suchkriterium aus einer Ansammlung von sogenannten simple conditions, verbunden durch die logischen Operatoren AND, OR, NOT und entsprechend geklammert.

simple conditions können also wie oben aufgelistet sein:

- comparisation-condition
- between-condtion
- like- condition
- in- condition
- match- condition
- exists- condition

i) comparisation condition

Syntax:

Where <scalar expression> <comparisation operator> <row constructor>

scalar expression: (<Attribut> | <scalar expression>)
scalar expression: siehe select-Klausel!
row constructor: <scalar expression> | NULL | (select-statement)
comparisation operator : = | < | ≤ | > | ≥ | < > | IS | NOT

Beispiel:

a) Gesucht: alle Kunden mit kdnr, die Meier heißen und vor dem 1.1.80 geboren sind:

Statement:
```
select kdnr from kunde
where name = 'Meier'
AND gebdat < '01-JAN-80';
```

b) Gesucht: die Kundennr aller Kunden, die in der gleichen Stadt wie Kunde 007 wohnen (welche Stadt das ist, ist nicht bekannt!)

Methode: Subquery

Statement:
```
select kdnr from adresse
where ort = (select ort from adresse where kdnr = 007)
```

Was passiert, wenn 007 mehrere Adressen hat?
Fehlermeldung, da einer Variablen auf der linken Seite mehrere Werte auf der rechten Seite zugeordnet werden müßten! Sind mehrere Werte zu erwarten, muß statt mit '=' dann mit 'in' arbeiten!
Die Operatoren IS, IS NOT: sind für NULL-Wert-Vergleiche:

```
Select name from kunde where gebdat IS NULL;
```

ii) between-condition:

Syntax:

Where <scalar expression> [NOT] BETWEEN <row constructor> AND <row constructor>

Bemerkung:

die between-condition ist nichts weiter als eine abkürzende Schreibweise:
y between x and z entspricht $x \leq y$ AND $y \leq z$
insbesondere sind die beiden Eckwerte mit enthalten!

Beispiel:

Gesucht: alle Produkte mit prodnr, deren Preise zwischen dem des Produkts 474 und dem mittleren Preis liegen.

Statement:
```
select prodnr from produkt
where preis between (select preis from produkt where prodnr = 474)
AND (select avg (preis) from produkt);
```

iii) like-condition:

Syntax:

Where <character-string-expression> [NOT] like <pattern> [escape <zeichen>]

character-string-expression ::= Attribut | Konkatenation | scalar expression
pattern ::= beliebiger string-Ausdruck

Die Zeichen in <pattern> werden wie folgt interpretiert:

- _ steht für irgendein einzelnes Zeichen (ist Platzhalter für genau 1 Zeichen)
- % steht für eine beliebige Folge von Zeichen (ist Platzhalter für einen String)
- alle anderen Zeichen stehen für sich!

Will man die besondere Bedeutung von _ und/oder % ausschalten, muß man ein escape-Zeichen angeben, das dann diesem Zeichen vorangestellt wird.

Beispiele:

a)
```
select name from kunde
where name like '%Lüden%';
```
Ergebnis: alle Kunden, die in ihrem Namen einen String der Form Lüden haben, also auch Müller-Lüdenscheid, etc.

b)
```
select farbe, prodnr from produkt
where farbe like ' gr_ _';
```
Ergebnis: grau, grün, nicht: gruen

c)
```
select kdnr, name from kunde
where name like '=_%' escape ,=‘;
```
Ergebnis: alle Kunden deren Namen mit ‘_‘ beginnen!

iv) in-condition

1. Form:

Syntax:

Where <scalar expression> [NOT] IN (<scalar-expression-kommaliste>)

Beispiel:

Gesucht: alle Kunden mit Postadresse in Bonn, oder Wismar oder Hamburg

Statement:
```
select kndr from adresse
where ort in ('Bonn', 'Wismar', 'Hamburg');
```

2. Form:

Syntax:

Where <scalar expression> [NOT] IN (select-statement)

Beispiel:

Gesucht: alle Aufträge mit anr, in denen Produkte aus Eisen bestellt wurden

Statement:
```
select distinct anr from positionen
where prodnr in (select prodnr from produkt where material = 'Eisen');
```

v) match-condition: (gibt es bei oracle nicht!)

Syntax:

Where <scalar expression> MATCH [UNIQUE] (select-statement)

Fall a) wird unique nicht angegeben,
ist die match-condition äquivalent zur in-condition:
where <scalar expression> in (select-statement)!

Fall b) ist unique spezifiziert,
ergibt die match-condition den Wert true, falls das select-statement genau einen Wert zurückbringt, der mit dem Ausdruck des scalar expressions übereinstimmt

Beispiel:

```
select * from kunde
where kdnr match unique (select kdnr from auftrag)
```

Ergebnis: alle Kunden, die nur genau einen Auftrag haben!

vi) exists-condition:

Syntax:

Where [NOT] exists (select-statement)

Anwendung: Test, ob in einer Tabelle ein Datensatz vorhanden ist

Beispiel:

Gesucht: alle Kunden, die was bestellt haben
Statement: select kdnr from kunde
where exists (select * from auftrag where kdnr = kunde.kdnr);

Gesucht: alle Kunden, die noch nie was bestellt haben
Statement: select kdnr from kunde
where not exists (select * from auftrag where kdnr = kunde.kdnr);

Anmerkung:

a) Getestet wird die Existenz ganzer Datensätze, d. h. das subselect ist immer in der Form 'select * ....'
b) Es muß immer eine äußere Referenz geben zwischen dem select, das exists aufruft und dem subselect (hier: adresse.kndr = kunde.kdnr)

d) group-by-Klausel:

Die Standardisierung SQL 92 (und 93) setzt die Implementation der folgenden Gruppenfunktionen fest:

- count
- sum
- avg
- max
- min

Der Effekt des Aufrufs dieser Gruppenfunktionen ist, daß ein Wert gebildet wird aus allen Zeilen längs des in Frage kommenden Attributs einer Tabelle, die die where-Bedingung erfüllen!

Diese Gruppenfunktionen können dazu verwendet werden um bzgl. gewisser anderer Attribute Gruppenwerte (Anzahl, Summe, Mittelwert, Maximum, Minimum) zu bilden.

Beispiel:

Frage: Wie viele Adressen haben die einzelnen Kunden?

Statement:
```
select kdnr count (*) from adresse
group by kdnr;
```

Ergebnis:

Kdnr	Count(*)
1	3
2	1
3	4
..	..

In der Tabelle POSITIONEN gibt es zu jeder Auftragsnr eine Reihe von Produkten und den zugehörigen Betrag (bestellteStückzahl * Preis):

```
select sum (betrag) from POSITIONEN
```
summiert alle Beträge auf, unbeachtet verschiedener Auftragsnummern!

Will man aber die Summe der Beträge pro Auftragsnr wissen, muß man die Summenbildung gruppieren, eben nach der Auftragsnr:

```
select sum (betrag) from Positionen
group by auftragsnr
```

Ergebnis:

Summe (Betrag)
721 DM
348 DM
....

oder:

```
select auftragsnr, sum (betrag) from positionen
group by auftragsnr
```

Ergebnis:

Auftragsnr	Summe (Betrag)
007	721 DM
008	348 DM
....	....

e) having-Klausel:

Die having-Klausel spezifiziert Einschränkungen bzgl. einer vorgenommenen Gruppierung.

Beispiel:

a) Gesucht: alle Kunden (kdnr) die mehr als zwei Adressen haben
   Statement:
```
select kdnr, count (*) from adresse
group by kdnr
having count (*) > 2;
```

b) Gesucht: alle Produktnr, die mehr als 10 mal bestellt wurden
   Statement:
```
select produktnr from positionen
group by prodnr
having count (*) > 10;
```

Warnung:

Gruppenfunktionen sind bei oracle keine scalar expressions, können also nicht einfach in einer where-Bedingung stehen (etwa where count (*) > 10)

f) order-by-Klausel:

Syntax:

```
order by <ausdruckkommaliste>
```

Alle in der Kommaliste erscheinenden Namen oder Ausdrücke müssen genau so in der select-Klausel aufgerufen worden sein!

Beispiel:

```
select name, vorname, gebort from kunde
order by gebort, name, vorname;
```

oder:

```
select substr (vorname, 1, 1) || '.  ', name from kunde
order by name, substr (vorname, 1, 1) ||'.  ';
```

UNION, INTERSECT, EXCEPT

Die mengentheoretischen Operationen $\cup$, $\cap$, \ werden durch SQL 92 (93) wie folgt implementiert:

a) Die beiden (Ergebnis-)Tabellen, die verknüpft werden sollen, müssen die gleiche Anzahl von Spalten haben.
b) Die zugehörigen Spalten müssen kompatible Datentypen haben.

Syntax:

(select-statement 1)
UNION | INTERSECT | EXCEPT
(select-statement 2);

Beispiele:

a) (select prodnr from produkt)
UNION
(select distinct prodnr from positionen);
Ergebnis: alle Produktnummern aus Produkt oder Positionen (jede nur 1 mal)

b) (select produktnr from produkt)
INTERSECT
(select distinct produktnr from positionen);
Ergebnis: alle Produktnummern, die schon mal bestellt wurden

c) (select produktnr from produkt)
EXCEPT
(select distinct produktnr from positionen);
Ergebnis: alle Produktnummern, die noch nicht bestellt wurden

auch:

d) (select kdnr, name from kunde)
UNION
(select prodnr, bezeichnung from produkt);
- sinnlos, aber syntaktisch ok!

Insert, update, delete:

Neben dem select-statement bietet die DML-Komponente von SQL noch die folgenden Statements an zum Einfügen, Aktualisieren oder Löschen von Datensätzen:

insert:

allgemeine Syntax:

INSERT INTO <tabellenname> [(Attributkommaliste)] source

source ist ein Tabellenausdruck und hat eine der folgenden Formen:

a) values (Werte-Kommaliste)
b) (select-statement)

In beiden Fällen muß die Anzahl der einzufügenden Werte und der jeweilige Datentyp mit der Tabellenstrukturvorgabe übereinstimmen!

Beispiel:

```
Insert into Kunde values(234, ‚Meier‘, ‚Horst, ’11.5.1979‘, ‚M‘);
```

Will man nur einen Datensatz mit ausgewählten Attributen einfügen, muß man diese unmittelbar nach dem Tabellennamen auflisten:

```
Insert into Kunde(kdnr, name) values(345, ‚Schmidt‘);
```

update:

allgemeine Syntax:

```
update <tabellenname>
set <Zuweisungskommaliste>
[where <conditional expression>]
```

Ist die where-Klausel nicht spezifiziert, werden alle Zeilen entsprechend der Zuweisungsliste modifiziert.

Zuweisungskommaliste:
Jede Zuweisung in der Zuweisungsliste hat die Form:

```
<Attribut> = <update-item>
```

wobei update-item entweder ein skalarer Ausdruck oder select-statement ist!

Beispiel:

```
update produkt
set (proddat, bezeichnung) = (select sysdate, 'Neuer Schalter' from dual)
where prodnr = 4712;
```

update oracle spezifisch:

1) Update <tabellenname>
   Set <attributname> = <expression> [, <attributname> = <expression> , ...]
   Where <conditional expression>;
2) Update <tabellenname>
   Set (attributkommaliste) = (select-statement) [, (attributkommaliste) = (select-statement)]
   Where <conditional expression>;

delete:

allgemeine Syntax:

```
delete from <Tabellenname>
[where contitional expression];
```

wird die where-Klausel nicht angegeben, wurden alle Tupel der Tabelle gelöscht!

Kapitel III DB-Organisation

III.1: Die 3-Ebenen-Architektur moderner Datenbanksysteme:

Mit der Einführung relationaler Datenbanksysteme wurde es möglich, drei relativ unabhängiger Aspekte eines Datenbanksystems von einander zu trennen:

- Die Sicht, die ein Anwender durch seine Datenbank-Applikation auf die Datenbank hat (Externes Schema)
- Die Konzeption und das Design, das der Datenbank-Applikation eigentlich zugrunde liegt; z. B. das relationale Datenbankschema der Applikation, (konzeptuelles Schema)
- Die Art und Weise, wie die Daten physisch gespeichert werden (internes Schema)

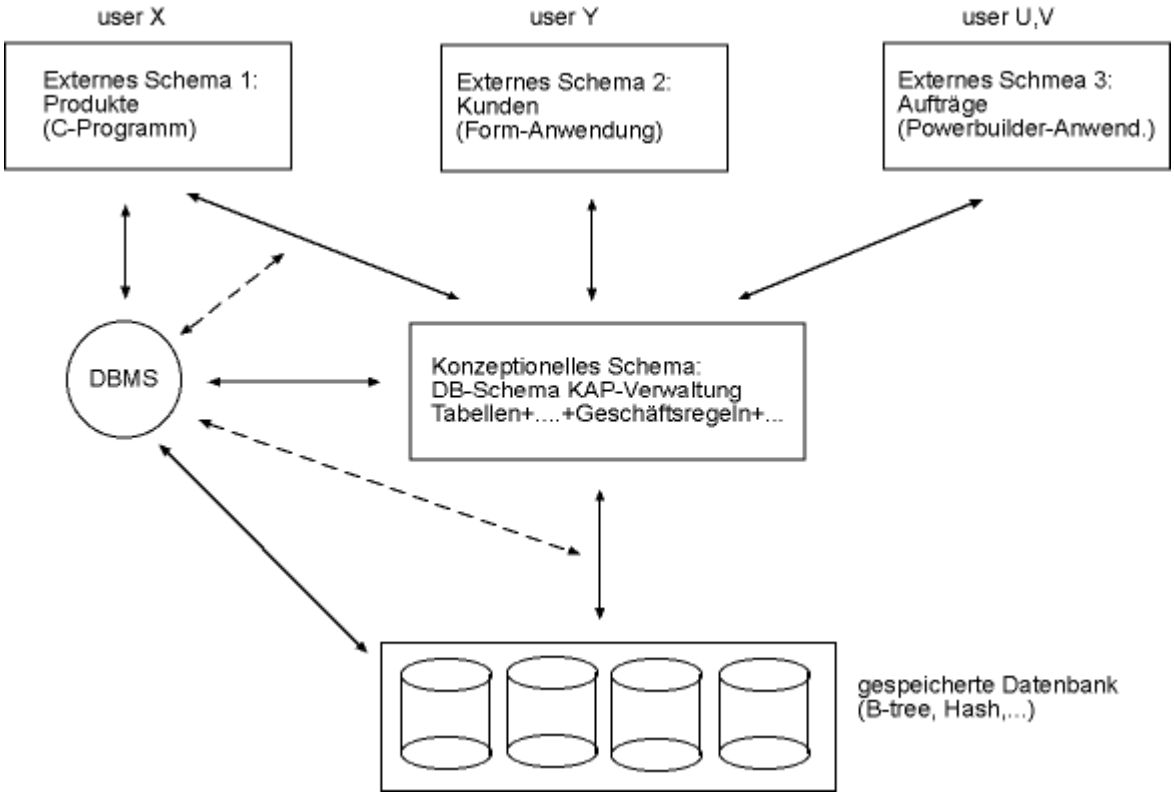
Diese 3 Aspekte waren bei den Vorgängern des relationalen Datenbanksystems noch nicht zu trennen!

Beispiel:

Betrachten wir wieder unsere Kunden-Auftrag-Produkt-Verwaltungs-Datenbank:
Funktional gibt es 1 Sachbearbeiter X, der nur die Produkte verwaltet, 1 Sachbearbeiter Y, der die reinen Kundendaten verwaltet, 2 Sachbearbeiter U und V, die die Auftragsdaten mit Positionen und Produkten bearbeiten.
Das Datenbankdesign umfaßt die 5 Tabellen Kunde, Auftrag, Adresse, Produkt, Positionen.
Sachbearbeiter X erhält eine Pflegemaske, in der nur Produktdaten vorhanden sind (z. B. ein C-Programm, oder eine Oracle-Forms-Applikation)
Sachbearbeiter Y hat eine Maske, mit der Kunden und Adressdaten bzgl. der Tabellen Kunde, Adresse gepflegt werden.

Sachbearbeiter U, V erhalten Sichten auf die Tabellen Auftrag, Positionen, Kunde, Produkt, aber eben nicht alle Daten.

Das läßt sich dann wie folgt darstellen:



Alle den relationalen Datenbanksystemen nachfolgende Datenbanksysteme sind so aufgebaut, also z. B. auch objektorientierte Datenbanksysteme. Dort ist dann ein konzeptionelles Schema kein relationales Schema mit 5 Tabellen, sondern ein objektorientiertes Schema mit z. B. 5 Klassen in einer hierarchischen Struktur. Die externen Schemata sind C++ oder Java-Programme auf bestimmte Teilstrukturen des objektorientierte Schemas und die Art der Speicherung der Objekte definiert das interne Schema.

Datenbanksysteme heutzutage trennen also klar zwischen einem logischen und einem physischen Datenmodell. Dies impliziert einen hohen Grad an physischer Datenabhängigkeit, d. h. man kann sowohl an der logischen wie an der physischen Seite Veränderungen vornehmen, ohne daß die jeweils andere Seite betroffen ist:

Beispiel:

Der Hersteller bringt ein neues Release mit neuer Speicherstruktur heraus! Das konzeptionelle Schema einer konkreten Anwendung muß nicht geändert werden!

Zum andern besitzen solche Datenbanksysteme auch logische Datenabhängigkeit, d. h. man kann auch Änderungen vornehmen an speziellen externen Schemata, ohne die Gesamtstruktur im konzeptuellen Schema verändern zu müssen und umgekehrt!

Beispiel:

Sachbearbeiter X bekommt auch noch die Positionen in den pflegenden Zugriff. (Am Datenbankdesign muß nichts geändert werden!)

oder:

Die KAP-Verwaltung wird erweitert um eine Lagerhaltung mit Regalpositionen, Raumangabe, usw.
Die bestehenden externen Schemata bleiben wie sie sind!

Mit diesem Konzept der 3 Ebenen Architektur eines Datenbank-Systems kommt man dem Ideal eines Datenbanksystems schon ziemlich nahe und ist vor allem in der Lage, durch neue Konzepte hinsichtlich der konzeptuellen Ebene (dem Datenbankmodell) die Anforderungen immer besser erfüllen zu können! Diese Anforderungen beinhalten insbesondere:

- Alle Daten einer realen Situation werden in einem integriertem Bestand einmal gespeichert
- Es gibt eine zentrale Kontrollinstanz, die dem Datenbestand und alle Vorgänge verwaltet und so Inkonsistenz vermeidet
- Alle Benutzer arbeiten auf der physischen Ebene mit dem gleichen Datenbestand. Damit sind einheitliche Integritätskontrollen und Schutz- und Sicherheitsmechanismen anwendbar
- Der Datenbestand kann logisch anwendungsbezogen strukturiert werden durch verschiedene Sichten.

III.2 Komponenten eines Datenbanksystems

Zu den wesentlichen Komponenten eines Datenbanksystems gehören:

- Datenbankmanagementsystem (DBMS)
- Data Dictionary (DD)
- Logdateien
- Datendateien
- Verschiedene Puffer
- Endbenutzerschnittstellen (I/O-Prozessor, SQL-Interpreter, API's)

Das DBMS (Datenbankmanagementsystem)

Das DBMS ist der Kern jedes Datenbanksystems. Es vermittelt zwischen den 3 Ebenen der Datenbank-Architektur, d. h. zwischen User, konzeptueller Ebene und der physischen Speicherung der Daten bei jedem Zugriff auf die Datenbank.
Damit es all diese Verwaltungs- und Vermittlungsaufgaben wahrnehmen kann, arbeitet das DBMS auf 3 Datenbeständen:

- den Datendateien
- dem Data Dictionary
- den Log-Dateien

Um die Arbeit und die benötigten Teilkomponenten eines DBMS zu verstehen, stellen wir uns die Frage:

Was muß bei der Bearbeitung einer Userabfrage alles ablaufen?

Prinzipiell das gleiche wie bei jedem anderen Programm auch:
Das Programm wird ins Memory geladen und sukzessive vom Prozessor abgearbeitet. Es gibt aber Unterschiede:

- Das Datenbanksystem kann nicht komplett in den Hauptspeicher geladen werden
- Die Abfragesprache (z. B. SQL) ist deskriptiv, d. h. sehr maschinenfern!

Beispiel:

Bezogen auf das KAPV_Beispiel könnte eine typische Anfrage lauten:

Anfrage: select * from kunde where name = 'Meier';

Was muß das DBMS tun:

- Anfrage aufnehmen
- Syntax prüfen
- Feststellen, ob das Objekt 'Kunde' in der Datenbank existiert und der User Zugriffsrechte hat
- Feststellen, welche Operationen zur Beantwortung der Anfrage auszuführen sind
- Feststellen, wie und wo das Objekt 'Kunde' gespeichert ist und welche Zugriffsmöglichkeiten es gibt
- Erstellen eines möglichst effizienten Programms zur Berechnung der Antwort
- Holen der konkreten Kunden-Objekte mit dem Namen 'Meier' aus dem Sekundärspeicher
- Auswahl der gewünschten Informationen
- Sicherstellen der Konsistenz während dieser Ausführung

Um diese Teilaufgaben abarbeiten zu können, kann man das DBMS in eine Folge von Komponenten zerlegen (siehe Schaubild).
Hier ist die Reihenfolge so gewählt, daß die Abarbeitung einer Benutzerabfrage nachvollzogen werden kann.

Quelle: G. Vossen, Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme

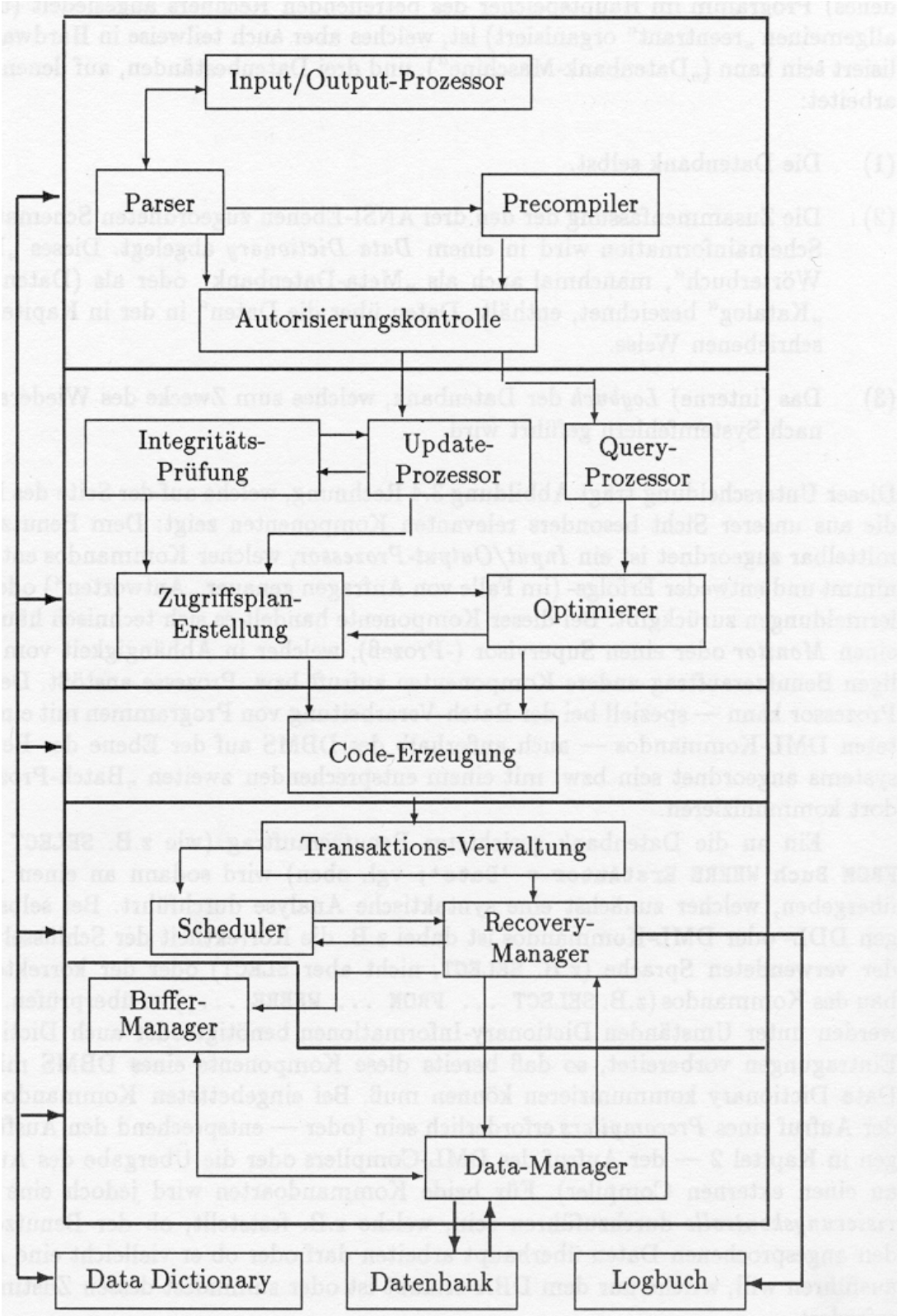


Abbildung 3.4: Komponenten eines DBMS.

Beschreibung:

- I/O-Prozessor: nimmt Befehle entgegen und gibt Antworten oder Fehlermeldungen zurück. (kann auch ausgelagert sein: z.B. bei Oracle:SQL-Plus). Ruft zur Bearbeitung andere Komponenten / Prozessoren auf.

- Parser:
  - führt syntaktische Analysen durch (korrekte Schlüsselwörter, korrekter Aufbau des SQL-Statements, ...)
    hier werden u. U. Datenbank-Informationen benötigt!
  - bei embedded SQL: Aufruf eines Precompilers
    sonst: Aufruf des DML-Compilers

- Autorisierungs-Kontrolle: darf der User mit den Daten arbeiten, darf er die Aktion auch ausführen

bisheriges Ergebnis: eine interne Form des ursprünglichen Benutzerauftrags
Zwischencode wie bei Compilern

- Update-Prozessor: erweitert den Zwischencode unter Einschaltung der Integritätsprüfung: Einbindung der Constraints
  oder
- Query-Prozessor: Integritätsprüfung bzgl. konkurrierender Zugriffe
  Übersetzung der Aufträge eines externen Schemas in das konzeptionelle Schema (Abkürzung von Tabelle, Spalte, o. ä.)

- Optimierer: Änderung der Anfrage (SQL-Statement) anhand vorgegebener Regeln (rule-based) oder statistischer Analyse (cost-based)

- Erstellung eines Zugriffprogramms: Codegenerierung für den Benutzerauftrag (Folge von Lese- und Schreibbefehlen)
  Resultat: 'Transaktion'

- Transaktions-manager: Synchronisation parallel laufender Transaktionen in einem Multi-User-System (optimale Ressourcenausnutzung: parallele Nutzung der Daten, Sperren setzen, ...

- Recovery-manager: Bewahren der Konsistenz der Datenbank bei abgebrochenen Transaktionen: Rollback; Informationen darüber im Log-Buch (Log-Dateien): welche Transaktion hat welche Daten wann wie geändert

- Geräte- und Speicher- Manager: physische Zugriffe auf die Datenbank unter Kontrolle des Transaktionsmanagers

- Dictionary Manager: liegt eigentlich in der 3. Dimension so über den Schema, daß er auf alle Komponenten Einfluß haben kann und diese mit Informationen versorgt.

Man kann ein DBMS nicht direkt managen, aber man kann die Arbeit des DBMS beeinflussen durch das Management der übrigen Komponenten, wenn das Datenbanksystem dies zuläßt.
Eine dieser wichtigen Komponenten ist das Data Dictionary (DD).

Das Data Dictionary (DD)

Das DD ist eine Metadatenbank (manchmal auch catalog genannt), die alle Informationen über die Datenbank strukturiert vorhält. Diese Informationen kann man unterteilen in statische und dynamische Informationen:

statisch: welche Objekte (Tabellen, Indizes, Sequences, User)
welche Rechteverteilung, Quoten, ...
welche Datenfiles, Logfiles existieren, usw...

dynamisch: welcher User ist aktiv mit welchen Transaktionen, welche Transaktionen sind offen, abgeschlossen, in welche Logdateien wird geschrieben,
usw.

All diese Informationen sind z. B. in relationalen Datenbankystemen wieder in Tabellen abgelegt. Privilegierte User wie der DBA können über views Einsicht in diese interne Struktur der Datenbank haben und daraus Informationen über Tuning-Möglichkeiten oder potentielle Fehlerquellen erhalten.
Das DBMS muß das komplette DD jederzeit in unmittelbarem Zugriff haben, wenn nicht die Abfragebearbeitung stocken soll, weil auf Informationen gewartet werden muß.
Je aktiver und größer eine Datenbank ist, desto umfangreicher ist das DD. Ziel muß sein, das DD komplett im Hauptspeicher des Datenbank-Servers zu halten!

Managementaufgabe:

- Einstellen gewisser Datenbank-Parameter, so daß das DD komplett ins Memory paßt
- Überwachung der Datenbank durch regelmäßiges Kontrollieren bestimmter Informationen des DD.

Memory-Puffer:

Das DBMS arbeitet nicht direkt mit Daten- und Logdateien, sondern über Puffer. Es gibt im wesentlichen die beiden Arten Datenpuffer und Logpuffer.
Pufferverwendung dient der Zeitersparnis bei Zugriff auf Daten. Die optimale Größe der Puffer hängt stark ab von der Aktivität des Datenbank-Systems:
- wieviele User sind parallel aktiv
- welche Datenbewegung findet statt

Puffer werden automatisch regelmäßig oder auf bestimmte Befehle hin geleert, d. h. in die Dateien weggeschrieben. Sind also Puffer zu klein gewählt, kann es sein, daß

Datenbank-updates warten müssen, weil die Puffer schon voll, aber noch nicht weggeschrieben wurden.
Zu groß sollten Puffer aber auch nicht gewählt werden, da dann im Falle eines Systemcrashs der Datenverlust zu groß ist.

Strategie:

Bei der Implementation der Datenbank klein anfangen und anhand regelmäßiger Kontrolle bestimmter DD-Werte eventuell korrigieren.

Managementaufgabe:

- Einstellen der Parameter in der Konfigurationsdatei des Datenbanksystems (sofern möglich)
- Regelmäßige Kontrolle des Datenbank-Verhaltens bzgl. Puffer anhand von DD-Informationen

Logdateien:

Logdateien sind ein wichtiges Instrument einer Datenbank zur Protokollierung aller Änderungen im laufenden Betrieb. Logdateien enthalten die Informationen darüber, welcher Benutzer mit welcher Transaktion welche Werte welcher Objekte wie verändert hat (geändert, gelöscht, neu hinzugefügt). Das heißt insbesondere, daß bei jeder Datenänderung der Wert vor und der Wert nach der Änderung festgehalten wird.
Da der Wert vor der Änderung protokolliert wird, hat der user (oder das DBMS) die Möglichkeit, seine Aktionen bis zu einem gewissen Punkt wieder rückgängig zu machen. Dies ist vor allem im laufenden Betrieb ein notwendiges Instrumentarium.
Da die Werte nach der Änderung nicht nur in den Datendateien, sondern auch in den Logdateien gespeichert werden, kann das DBMS auch bei Verlust einer Datendatei einen fast-aktuellen konsistenten Datenbankzustand wiederherstellen. Man kann also z. B. ein delete from Kunde-Statement unter gewissen Bedingungen durch den rollback-Befehl wieder rückgängig machen! Dabei liest dann das DBMS alle gelöschten Datensätze aus der Logdatei zurück in die Datendatei (über die Log.- bzw. Datenpuffer!)

Wie an diesem Beispiel schon zu sehen ist, wachsen die Logdateien in einer aktiven Datenbank sehr schnell: jedes Statement mit seinen Auswirkungen (z.B. 1000 gelöschte Datensätze) wird festgehalten.

Nun sind aber z. B. die Werte vor einer Änderung ab einer gewissen Zeit nicht mehr interessant, weil z. B. die Tabelle inzwischen mehrfach geändert wurde, o. ä. Analoges gilt für die Datenwerte nach der Änderung. Dieser Punkt wird u. a. auch bei Oracle dazu benutzt, das unbegrenzte Wachstum der Logdateien durch zyklisches Überschreiben zu verhindern.
Man arbeitet also hier mit zwei oder mehr Logdateien fester Größe: eine wird beschrieben; ist sie voll, wird die zweite beschrieben; ist diese voll, wird die erste überschrieben, usw. Dadurch geht natürlich, wenn diese Logdateien zu klein gewählt werden, schnell Information verloren. Hier ist also die Managementaufgabe, die Anzahl und Größe der Logdateien zu bestimmen und im laufenden Betrieb zu überwachen. Zusätzlich bietet Oracle auch noch die Möglichkeit, die Logdateien vor

dem Überschreiben zu archivieren (etwa auf ein Band). Dies ist für recovery-Operationen unter Umständen notwendig! Siehe hierzu auch Kap. III.5.

Insgesamt gesehen, sind die Logdateien so etwas wie das Gedächtnis des DBMS, in denen alle Aktionen protokolliert sind nach ihrem Wertgehalt.

Managementaufgabe:

- Erzeugung einer geeigneten Anzahl von Logdateien mit einer geeigneten Größe!
- Überwachung des Datenbank-Verhaltens bzgl. dieser Logdateien und gegebenenfalls Änderung (Anzahl, Größe)

Datendateien:

Es gibt zwei Arten: die eigentlichen Datendateien und Indexdateien.
Datendateien nehmen die Daten der Datenbankobjekte (Tabellen, Klassen) physisch auf. Der DBA muß als darauf achten, daß immer genügend Speicherplatz für die Datenbankobjekte zur Verfügung steht. Parallel dazu ist es vor allem aus Gründen der Performanceverbesserung sinnvoll, Indizes zu benutzen, die den Zugriff auf die Daten schneller machen.
Sinnvollerweise ist hier darauf zu achten, daß Daten-, Index- und Logdateien auf physikalisch verschiedenen Platten liegen, damit parallel gesucht und geschrieben werden kann.
Liegen die drei Arten von Dateien auf einer Platte, muß z. B. beim Lesevorgang der Schreib/Lesekopf der Platte beim Wechsel zwischen Index- und Datendatei jedesmal neu positioniert werden. Das kostet Zeit, die nicht notwendig ist!

Managementaufgabe:

- Bestimmung von Größe und Anzahl von Daten- und Indexdateien bei der Planung der Datenbank.
- Verteilung auf verschiedene Platten
- im laufenden Betrieb: - eventuell neuen Speicherplatz reservieren
  - optimale Verwendung der Indizes kontrollieren (Tuning)

Abfragesprachen: (Kommunikationsschnittstelle)

Die Abfragesprache wird im wesentlichen von den Anwendungsprogrammierern benutzt, um Endbenutzern die Kommunikation mit der Datenbank zu ermöglichen. Selten dürfen die Endbenutzer selbst mit Hilfe dieser Abfragesprache Anfragen stellen. Nun kann es sein, daß die Anwender sich beschweren, daß bestimmte Datenbankabfragen zu lange dauern. In diesem Fall sollte sich der DBA mit dem Anwendungsprogrammierer zusammensetzen und versuchen, diese Funktionalität z. B. durch Tunen des Statements zu verbessern. (siehe auch Kapitel III.6 Tuning)

III.3 Transaktionen

Wie in Kapitel I.2.2 bereits besprochen, genügen alle Daten einer konkreten Anwendungssituation, gewisse Regeln und Bedingungen. Diese Geschäfts- oder Integritätsregeln müssen also auch für den Datenbestand im modellierten Datenbankdesign dieser Anwendung gelten, bei der Erzeugung der Datenbank, im laufenden Betrieb und auch nach Systemabstürzen. Wir haben weiter in Kapitel II.1 gesehen, daß ein relationales Datenbanksystem, das den Standardisierungen SQL92 bzw. SQL3 genügt, verschiedene Arten von Constraints in der Datenbank ablegen kann, die solche Regeln enthalten:

PK-FK-Constraints bzgl. interrelationalen Integritätsbedingungen
Check-Constraints
PK-Constraints } bzgl. intrarelationalen Integritätsbedingungen

Darüber hinaus können weitere Integritätsregeln durch stored procedures bzw. Datenbank-Trigger in der Datenbank realisiert werden. Die Kontrolle aller so implementierten Integritätsregeln obliegt dem DBMS, also unabhängig von irgendwelchen Anwendungen!
Eine Datenbank heißt konsistent, wenn alle enthaltenen Relationen den inter- und intrarelationalen Regeln genügen.
Durch die Konzepte constraint, trigger und stored procedures kann also weitgehend Konsistenz beim Design der Datenbank erzwungen werden.
Fraglich ist, ob diese Konzepte auch ausreichen, im laufenden Betrieb oder bei Abstürzen Konsistenz zu erhalten!

Laufender Betrieb bedeutet für ein Datenbanksystem, daß Datenmanipulations-Operationen den Datenbestand ändern.
Da ist also zu verlangen, daß diese Operationen immer einen konsistenten Zustand in einem erneut konsistenten Zustand überführen!

Betrachten wir als Beispiel die Relationen Kunde und Adresse mit der Geschäftsregel: Jeder Kunde muß mindestens einen gültige Adresse haben.

Soll ein neuer Kunde angelegt werden, überführt ein alleiniges insert in die Tabelle Kunde die Datenbank **nicht** in einen erneut konsistenten Zustand, da die oben angeführte Geschäftsregel verletzt ist! Erst ein Anlegen einer gültigen Adresse über ein insert in die Tabelle Adresse führt zu einem konsistenten Datenbankzustand.
Oder:
Soll ein Kunde gelöscht werden, müssen auch seine sämtlichen Adressen, Aufträge usw. gelöscht werden. Wieder sind also mehrere DML-Statements notwendig

Die Forderung nach Konsistenzerhaltung im laufenden Betrieb führt also notwendig dazu, einzelne DML-Operationen zu einer Gruppe zusammenzufassen, einer sogenannten Transaktion:

Definition:

> Eine Transaktion ist eine Folge von Datenmanipulations-Operationen, die eine Datenbank von einem konsistenten Zustand in einen erneuten konsistenten Zustand überführt.

Transaktionen haben einen wohldefinierten Anfang (begin, oder einfach die erste Operation) und ein wohldefiniertes Ende: das Statement commit oder das Statement rollback.

Damit eine Transaktion wieder konsistente Zustände liefert, ist unbedingt notwendig, daß sie entweder komplett oder gar nicht ausgeführt wird! Kann also eine Transaktion nicht ordnungsgemäß ausgeführt werden, müssen alle bisher in dieser Transaktion gemachten Änderungen im Datenbestand zurückgenommen werden (rollback).
Die Systemkomponente, die diese Atomizität der Transaktion überwacht, ist der Transaktionsmanager des DBMS.

Das **commit-Statement** signalisiert eine erfolgreiche Durchführung der Transaktion: alle gemachten Datenänderungen können gespeichert werden.

Das **rollback-Statement** signalisiert ein erfolgloses Transaktionsende: alle bereits vollzogenen Änderungen dieser Transaktion müssen zurückgenommen werden.

Damit der Transaktionsmanager arbeiten kann, braucht er also zu jedem Zeitpunkt eine Menge Informationen über alle noch offenen und bereits abgeschlossenen Transaktionen: Beginn, welche Daten wurden wie geändert, welche Statements, ... Diese Informationen führt die Datenbank in den Logdateien und im Data Dictionary mit. Diese spielen im laufenden Betrieb und beim Recovery eine große Rolle.

Das Konzept, das hinter der Transaktionsverarbeitung steht und als Ziel die Konsistenz hat, beinhaltet insbesondere, daß alle Transaktionen die folgenden vier sogenannten ACID-Bedingungen erfüllen müssen:

Atomicity: Alle Transaktionen werde komplett oder gar nicht ausgeführt

Consistency: AlleTransaktionen erhalten die Konsistenz einer Datenbank

Isolation: Die von einer Transaktion benötigten Daten müssen vor der Konsistenz-gefährdenden Nutzung parallel laufender Transaktionen geschützt werden.

Durability: Ergebnisse einer erfolgreichen Transaktion bleiben erhalten, auch bei Systemausfällen!

III.4 Kontrolle konkurrierender Zugriffe

Der Transaktionsmanger eines DBMS verarbeitet mehrere Transaktionen in einem Multi-User-Betrieb nicht hintereinander (Single-User-Betrieb) sondern parallel. Damit ist sofort klar, daß es in einem Mehrbenutzerbetrieb große Probleme gibt, die Isolationsbedingungen in der Transaktionsverarbeitung einzuhalten. Diese Probleme

sind unter dem Begriff konkurrierende Zugriffe zusammengefaßt und betreffen vornehmlich die Eigenschaft Isolation von ACID.

Betrachten wir drei typische Probleme P1, P2, P3:

(generelle Voraussetzung für die folgenden besprochenen Transaktionen:

- Eine Transaktion beinhaltet read- und write-Operationen mit Peripheriegeräten: r(x), w(x), x ein Datenbankobjekt;
- Daneben sind elementare Zuweisungen und updates u(x) erlaubt und zusätzlich die Befehle begin, commit, rollback.)

P1: Lost update

Zwei Transaktionen t_1 und t_2 werden nach folgendem Plan ausgeführt:

t_1	Zeitpunkt	t_2
Begin t_1	1	
r (x)	2	
	3	Begin t_2
	4	r (x)
u (x)	5	
	6	u (x)
w (x)	7	
commit t_1	8	
	9	w (x)
	10	commit t_2

Erläuterung:

Beide Transaktionen lesen den gleichen x-Wert;
t_1 schreibt vor t_2 seine Änderung zurück, d. h. diese Änderung ist in der DB nicht sichtbar! Der Sachbearbeiter, der t_1 gestartet hat, wird sich also wundern! Der Sachbearbeiter, der t_2 ausführt, wird nie gewahr, daß er eine Änderung von t_1 überschreibt!

P2: Dirty Read:

Ablaufplan für die beiden Transaktionen t_1 und t_2 :

t_1	Zeitpunkt	t_2
Begin t_1	1	
r (x)	2	
u (x)	3	
w (x)	4	
	5	Begin t_2

	6	r (x)
	7	u (x)
rollback t_1	8	
	9	w (x)
	10	commit t_2

Erläuterung:

t_2 liest einen von t_1 geänderten x-Wert. Nach dem Lesen wird die Änderung von t_1 zurückgenommen (rollback), d. h. der von t_1 veränderte Wert x' = t_1 (w (x)) hat sich nie in der Datenbank befunden!

$\Rightarrow$ der von t_2 zurückgeschriebene Wert ist inkonsistent, da er sich auf ein x bezieht, das es nicht gibt!

P3: Unrepeatable Read:

Anmerkung:

x, y, z sind numerische Objekte mit den Werten
x = 40, y = 50, z = 30, also x + y+ z = 120
(z. B. Kontostände einer Bankanwendung)

Ablaufplan von t_1 und t_2 :

t_1	**Zeitpunkt**	**t_2**
Begin t_1	1	
sum: = 0	2	
r (x)	3	
r (y)	4	
sum: = sum + x	5	
sum: = sum + y	6	
	7	Begin t_2
	8	r (z)
	9	z: = z – 10
	10	w (z)
	11	r (x)
	12	x: = x + 10
	13	w (x)
	14	commit t_2
r (z)	15	
sum: = sum + z	16	
commit t_1	17	
	18	

Erläuterung:

t_1 berechnet die Summe x + y
t_2 transferiert 10 Einheiten von z nach x (sozusagen Bank-intern), so daß die Summe konstant bleibt
t_1 liest z nach diesem Transfer und berechnet x + y+ z, die falsche Summe (110) ! (obwohl der Wert insgesamt konstant geblieben ist!)

Bei allen drei Problemen ist die Bedingung *Isolation* der Transaktionen verletzt: t_1 und t_2 greifen konkurrierend auf das gleiche Objekt zu und produzieren so Inkonsistenzen.

Lösung:
Es werden drei neue Aktionen für Transaktionen definiert:

a) Lese- Sperre- setzen rlock (sperrt x im shared mode)
b) Schreib- Sperre- setzen wlock (sperrt x im exclusive mode)
c) Sperre aufheben unlock

Wirkung der Sperren:

a) Hat t ein Objekt x mit rlock (x) gesperrt, kann sie danach höchstens lesend auf x zugreifen
b) Hat t ein Objekt x mit wlock (x) gesperrt, kann sie danach lesend oder schreibend auf x zugreifen.

Für das Setzen von Sperren auf das gleiche Objekt von mehreren Transaktionen gilt folgende Erlaubnistabelle (1 = erlaubt; 0 = nicht erlaubt)

t_1 / t_2	rlock	wlock
rlock	1	0
wlock	0	0

D. h. ein Objekt x, das von t_1 mit rlock (x) gesperrt wurde, darf von beliebig vielen anderen Transaktionen mit rlock (x) gesperrt (und damit gelesen) werden.

Ist ein Objekt x von t_1 mit wlock (x) gesperrt, kann keine andere Transaktion auf x eine Sperre setzen (und damit auch nicht auf x zugreifen!)

Lösung des Problems P3:

Und zwar so, daß beide Transaktionen t_1 und t_2 immer noch parallel arbeiten und nicht hintereinander!

t_1	**Zeitpunkt**	**t_2**
Begin t_1	1	
	2	Begin t_2
sum: = 0	3	
rlock (y)	4	

r (y)	5	
sum: = sum + y	6	
	7	wlock (x)
	8	wlock (z)
	9	r (x)
	10	r (z)
	11	z : = z – 10
	12	x: = x + 10
	13	w (z)
	14	Unlock (z)
rlock (z)	15	
r (z)	16	
sum: = sum + z	17	
	18	w (x)
	19	Unlock (x)
rlock (x)	20	
r (x)	21	
sum . = sum + x	22	
unlock (x)	23	
unlock (y)	24	
unlock (z)	25	
	26	Commit t_2
commit t_1	27	

Die Lösung des Isolierungsproblems durch Setzten von Schreib- bzw. Lesesperren bringt aber ein neues Problem mit sich : sogenannte Deadlocks.

Beispiel Deadlock:

Zwei Transaktionen t_1 und t_2 wollen die gleichen Objekte x und y bearbeiten (verändern), aber in umgekehrter Reihenfolge:

t_1	**Zeitpunkt**	**t_2**
Begin t_1	1	
rlock (x)	2	
r (x)	3	
	4	Begin t_2
	5	Rlock (y)
	6	R (y)
?→ wlock (y)	7	
	8	Wlock (x) ← ?

Erläuterung:

t_1 liest x und möchte anschließend y ändern.

t_2 liest y und möchte anschließend x ändern.

Beide können wlock erst setzen, wenn die jeweils andere ihr rlock aufgehoben hat.

Folge: t_1 und t_2 warten aufeinander (bis der Strom ausfällt).

1.Lösung:

Das DBMS bzw. der Transaktionsmanager protokolliert welche Transaktion auf eine unlock-Aktion einer anderen Transaktion wartet. Dann können solche Zykel (t_1 wartet auf t_2 und t_2 wartet auf t_1)erkannt werden.
Eine oder beide Transaktionen werden zurückgesetzt (rollback) und zu einem anderen Zeitpunkt neu gestartet.

Fazit:

Mit Hilfe der Sperren ist das DBMS zusammen mit den Transaktionskonzept in der Lage, im laufenden Betrieb Konsistenz zu wahren, allerdings um den Preis von deadlocks! Zu jedem beliebigen Zeitpunkt ist eine Datenbank in inkonsistentem Zustand, weil es (zumindest in einer aktiven Multiuser-Datenbank) offene Transaktionen gibt!

Die gesamte Problematik läßt sich entzerren durch die beiden folgenden Forderungen / Konstruktionen:

Def.: (Serialisierbarkeit)
Die Ausführung einer Menge von Transaktionen ist serialisierbar, falls das gleiche Ergebnis erzielt werden kann durch Hintereinanderausführung der einzelnen Transaktionen.

Dies impliziert, daß dann natürlich auch keine Konkurrenzprobleme im Datenbanksystem mehr existieren.

Wie kann man Serialisierbarkeit garantieren:
Zum Beispiel durch das Zwei-Phasen-Sperrprotokoll:

1. Phase: Alle für die entsprechenden Zugriffe einer Transaktion auf Datenbankobjekte notwendigen Sperren werden angefordert und gesetzt. Jetzt kann die Transaktion arbeiten.

2. Phase: Nach der Bearbeitung des ersten Datenbankobjekts wird dessen Sperre von der Transaktion freigegeben. Ab jetzt darf diese Transaktion keine neuen Sperren mehr anfordern, sondern nur noch die von ihr gesperrten Objekte bearbeiten oder weitere Sperren freigeben.

Durch das folgende Theorem ist damit Serialisierbarkeit garantiert.

Theorem:

Gehorchen alle Transaktionen eines Datenbanksystems dem 2-Phasen-Protokoll, ist jede Ausführung jeder Menge paralleler Transaktionen serialisierbar!

Folge: Konkurrenzproblem gelöst!

Es bleibt aber immer noch ein Problem, und zwar das der **Systemabstürze.**
Wie eben gesagt, gibt es in einer aktiven Datenbank zu jedem Zeitpunkt mit hoher Wahrscheinlichkeit offene Transaktionen. Stürzt das System ab und fährt das DBMS die Datenbank in den Zustand wieder hoch, der unmittelbar vorher bestand, ist die Datenbank inkonsistent!

Lösung im einfachen Fall zeitlich nacheinander laufender Transaktionen:

t_1	Zeitpunkt
Begin t_1	1
wlock (x)	2
wlock (y)	3
wlock (z)	4
u (x)	5
u (y)	6
u (z)	7
w (x)	8
w (y)	9
w (z)	10
unlock (...)	11

→ Absturz!

Vor dem korrektem Ende von t_1 stürzt das System ab.
Zur Lösung muß die gesamte Transaktion nach dem Neustart zurückgerollt werden. (Bei z. B. einem noch ausstehendem Insert oder Update weiß das DBMS ja nicht, welche Werte eingefügt bzw. ersetzt werden sollen! d. h. komplettes Zurücknehmen ist die einzige Reaktionsmöglichkeit!) Dazu ist notwendig, daß das DBMS alle Werte aller Daten vor der Änderung in einer Logdatei abspeichert. Mit diesen Informationen kann nach einem Neustart ein konsistenter Zustand hergestellt werden.

Bei konkurrierenden Transaktionen kann aber folgendes Problem auftreten:

t_1	Zeitpunkt	t_2
Begin t_1	1	
wlock (x)	2	
r (x)	3	
u (x)	4	
w (x)	5	
	6	Begin t_2
	7	wlock (y)
	8	r (y)
	9	u (y)
	10	w (y)
	11	unlock (y)

	12	commit
wlock (y)	13	
r (y)	14	
u (y)	15	
w (y)	16	
....	17	.....

→ crash!

Bei Neustart wird t_1 zurückgerollt und der Zustand der DB unmittelbar vor t_1 wird hergestellt.

Folge:

t_2 wird mit zurückgesetzt, obwohl t_2 vor dem crash erfolgreich beendet wurde. Widerspruch zur Bedingung *Durability* von Transaktionen!

Lösung:

Nach dem Zurückrollen aller offenen Transaktionen müssen alle dadurch betroffenen und mit zurückgerollten abgeschlossenen Transaktionen wieder nachgezogen werden (roll forward).

Dazu benötigt das DBMS neben den Werten der Daten vor der Änderung durch die Transaktion auch alle Werte der Daten nach der Änderung, die in der Logdatei gespeichert werden.

Fazit:

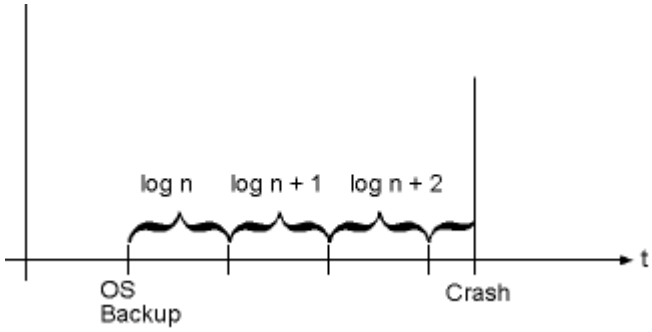
Alle Transaktionen einer Datenbank müssen den ACID-Bedingungen genügen. Das garantiert, daß ein konsistenter Datenbankzustand durch eine Transaktion wieder in einen konsistenten Zustand überführt wird. Diese Forderung, insbesondere die Bedingung I (solation) führt in multiuser-Datenbanken zu den angesprochenen Konkurrenzproblemen, die dann über Lock-Mechanismen gelöst werden sollen. Aber auch dabei können Probleme auftreten: Deadlocks, die wiederum über das 2-Phasen-Sperrprotokoll verhindert werden.

III.5 Backup and Recovery

Grundlage eines jeden Datenbank-Recoverys ist ein möglichst aktuelles Betriebssystem-Backup (OS-Backup) der Datenbank-Dateien (: Datendateien, Konfigurationsdateien, etc...). Dieses OS-Backup muß natürlich erstellt werden, wenn die Datenbank geschlossen ist. Mit diesem möglichst tagesaktuellen OS-Backup kann man also immer den Zustand der Datenbank vom Vortag wiederherstellen, wenn es einen Systemfehler gegeben hat.

Eine wichtige Managementaufgabe ist also die Abstimmung des DB-Administrators mit dem Systemmanager bzgl. der Aktualität des OS-Backups.

In vielen Betrieben reicht diese Backup-Strategie aus!
Es gibt aber auch eine Reihe von Datenbank-Anwendungen, die auf aktuelle Daten angewiesen sind. In diesem Fall muß also die Lücke, die zwischen dem letzten OS-Backup und dem Systemcrash klafft, irgendwie geschlossen werden. Diese Lücke könnte gefüllt werden von all den Logdateien, die in diesem Zeitraum beschrieben wurden:



log_n, log_{n+1}, log_{n+2}, sind vollgeschriebene Logdateien, die wieder zyklisch überschrieben werden; log_{n+3} ist das Logfile, das zum Zeitpunkt des Crashs aktiv war!

Hätte man also die Logdateien log_n, log_{n+1} und log_{n+2}, so könnte man die Lücke zwischen OS-Backup und Crash schließen und alle Tagesarbeit würde gerettet! Die Hersteller großer Datenbanksysteme wie z.B. Oracle bieten einen solchen Mechanismus an.

Beispiel ORACLE:
Die Datenbank kann in 2 Modi gefahren werden: NOARCHIVELOG und ARCHIVELOG.
Defaultwert ist NOARCHIVELOG.
Im Archivelog-Modus werden alle Logdateien vor dem Überschreiben in einem speziellen Verzeichnis gespeichert. Dazu müssen zwei Schritte getan werden:

1) In der Datei initSID.ora müssen drei Parameter eingetragen werden:
   log_archive_start = TRUE
   log_archive_dest = < Pfad des Verzeichnisses der archivierten Logdatei>
   log_archive_format = log%s.arc (= fortlaufende Numerierung)

2) Nach dem Neustart der Datenbank muß die Datenbank in dem ARCHIVELOG-Modus versetzt werden durch den Befehl: alter database archivelog;

Durch diese Maßnahme sind alle Daten vor einem Systemfehler sicher! Im Falle eines Systemabsturzes kann nach dem Wiederhochfahren der Datenbank ein recovery gestartet werden durch den Befehl: alter database recover; Dabei werden dann sukzessive alle benötigten archivierten Logdateien eingespielt und die Datenbank so in einen aktuellen konsistenten Zustand überführt.

Mit diesen Maßnahmen hat man das ORACLE-Datenbanksystem abgesichert und kann beliebige Systemfehler (fast) lückenlos wieder aufarbeiten!
Der Befehl

alter database recover (vorher: startup alter database exclusive)

kann mit verschiedenen zusätzlichen Parametern aufgerufen werden
Fährt die Datenbank im ARCHIVELOG-Modus, braucht man in der Regel keinen zusätzlichen Parameter anzugeben. Das Datenbanksystem sucht sich anhand interner Nummerierung selbst die archivierten Logdateien, die es braucht, um die Lücke ab dem OS-Backup zu schließen!

Ein besonders problematischer Fall ist, wenn ein noch nicht archiviertes logfile zerstört wurde.
Hier kann man natürlich nur bis zur Logdatei unmittelbar vor der zerstörten Logdatei recovern! Es gehen also Daten verloren!
Diesen schlimmen Fall kann man allerdings dadurch nahe zu ausschließen, daß man Gruppen von Logdateien bildet, die Spiegelbilder (Sicherheitskopien) enthalten.

Fazit:

Große Datenbanksysteme bieten heute i. d. Regel umfangreiche Sicherheitsmechanismen, die ein zeitaktuelles Recovery nach einem Crash ermöglichen.
Sie sollten als Datenbank-Manager diese Mechanismen allerdings testen und sich damit vertraut machen, bevor diese wegen eines Crashs eingesetzt werden müssen!

Neben diesen physischen Recovery-Mechanismen bietet Oracle noch die Möglichkeit, sogenannte logischen Backups der Datenbank zu erzeugen. Dies sind die Funktionen Exp(ort) und Imp(ort).

Mit exp können Sie
- einzelne Tabellen
- einzelne User mit ihren Datenbankobjekten
- eine komplette DB

exportieren, d. h. alle strukturellen Informationen (Name, storage-Klausel, ...) und Daten aller Datenbankobjekte (Tabellen, Sequences, Indizes, ...) werden in eine OS-Datei geschrieben.
Diese Datei kann mittels der Imp-Funktion wieder in die Datenbank gelesen werden!

Die Exp/Imp-Funktionen kann man benutzen:
um
- jeden Abend einen full database export zu machen.
  Dann kann man am nächsten Tag im laufenden Betrieb, falls mit einer Tabelle etwas nicht stimmt, diese Tabelle über imp neu (alt) erstellen.
- Fragmentierung aufheben, indem man die Datenbank exportiert, die Datendateien löscht und neu erzeugt und dann den Export mittels imp wieder einliest!
- Tabellen / User / Userobjekte allgemein zwischen verschiedenen Datenbanken transportieren

III.6 Tuning

Was kann bei einem Datenbanksystem getunt werden?
Was soll erreicht werden: daß die Anfragen an ein Datenbanksystem schnell beantwortet werden!
Wir wollen dies am Beispiel eines relationalen Systems untersuchen:

Wer oder was ist an einer Abfrage beteiligt?!
1) Der Benutzer (das SQL-Statement)
2) Datenbank-Organisationen (Puffergröße, Memory-Belegung, Indexierung, Datenbankdesign)
3) Der Client (Rechner) (lokale SQL-Maschine bei gewissen Datenbank-Konfigurationen)
4) Das Netz (Kabel, Router, Bridges, Packetgröße, ...)
5) Der Server (Prozessor, Memory, Platten)
6) Zusammenarbeit zwischen Betriebssystem und Datenbank
7) Log.Verb. zwischen Server und Client: (native, ODBC, ...)

Jede der genannten Komponenten kann die Performance beeinflussen!

zu 1) Tuning der Abfrage:

SQL-Abfragen sind Abfragen mittels des select-Statements.
Das select-Statement umfaßt :
select-Klausel, from-Klausel, where-Klausel, group by-Klausel, having-Klausel, order by-Klausel.
Die select-Klausel ist für Tuning unerheblich, sie legt nur das Layout der Ergebnismenge fest!

Die from-Klausel: hier müssen alle Quellen angegeben werden, aus denen die Daten zusammengesetzt werden!
Ist nur eine Tabelle beteiligt, gibt's nichts zu beachten!

2-Tabellen-join:
immer die kleinste Tabelle zuletzt auflisten, da diese zuerst bearbeitet wird!

n-Tabellen-join und n > 2:
immer die Schnittabelle als letzte, d. h. die Tabelle, die die meisten join-Bedingungs-Spalten enthält!

Diese Faustregeln hängen natürlich auch noch davon ab, auf welchen Tabellen und Spalten Indizes liegen! (siehe Punkt 2)

where-Klausel:
Hier gibt es Hersteller-spezifische Vorgaben, die beachtet werden müssen. Wir beziehen uns im Folgenden auf Oracle!

a) Es gibt eine Reihe von SQL-Funktionen und Operatoren, deren Benutzung explizit Indizes ausschließen:
Beispiele:
- SQL-Funktion substring (...)
statt dessen: like-Operator verwenden!
- != -Operator; statt dessen < oder > verwenden
- || - Konkatenation von Spalten
- arithmetische Operatoren in Ausdrücken (+, ·, - , /)

Vermeidet man also obige Funktionen und Operatoren, werden vorhandene Indizes benutzt und damit die Abfrage beschleunigt!

b) Auch Subselects verlangsamen die Verarbeitung eines select-statements. Schneller ist hier die Verwendung von joins!

order by-Klausel:

Ist eine order by-Klausel spezifiziert, werden Indizes nur dann benutzt, wenn beide folgenden Bedingungen erfüllt sind:

i) alle Spalten der order by-Klausel sind in der gleichen Reihenfolge in einem einzigen Index enthalten!

ii) alle Spalten der order by-Klausel müssen in der Tabelle als NOT NULL deklariert sein!
Beispiel:

```
select name, vorname from kunde
where geschlecht = 'w'
order by name, vorname
```

```
Name1 varchar 2 (40) not null
vorname varchar 2 (40) not null   } in Kunde deklariert
```

Fall 1: Es existiert ein Index, erzeugt durch: create index I1 on kunde (name, vorname);
Dieser wird benutzt!
Fall 2: Es existieren statt dessen die beiden Indizes I_name und I_vorname, erzeugt durch create index I_name on kunde (name); bzw create index I_vorname on kunde (vorname);
Keiner der beiden wird benutzt!

zu 2) Datenbank-Organisation

Die Datenbank-Organisation betrifft die physische Installation der Datenbank als Programm bzgl. des OS und die innere Logik (Datenbankdesign).

Physische Organisation:
- Data Dictionary muß komplett im Memory liegen (d. h. für das Datenbanksystem muß ein größeres Memory-Stück reserviert werden!
- Einige Datenbanken sind so organisiert, daß sie einen eigenen Bereich im Memory halten, wo die letzten SQL-Statements kompiliert gehalten werden. Kommt eine neue Abfrage, die ein solches Statement enthält, wird dieses benutzt! Das spart

Zeit! Auch dafür muß Platz im Memory sein!

- Bei jedem Sortiervorgang werden temporäre Tabellen angelegt. Dafür kann ein eigener Bereich im Memory benutzt werden. Dieser muß also auch berücksichtigt werden bei der Reservierung von Memory beim Hochfahren der Datenbank!
- Schließlich schreibt die Datenbank nie direkt in Dateien, sondern benutzt wie jedes andere Programm Puffer, die im Memory angelegt sind: Datenpuffer, Logpuffer
  Diese werden zyklisch geleert. Sind sie zu klein, kann es zu Datenbank-Stillstand kommen, weil ein Datenbank-Prozeß warten muß, bis er in einen noch nicht geleerten Puffer schreiben kann!
  Also: Puffer sollten großzügig angelegt sein im Memory.
  Es ist gar nicht so unwahrscheinlich, daß eine große Multiuser-Datenbank langsam ist genau wegen zu kleinem Puffer!
  Bei Oracle gibt es eine Tabelle V$ SYSSTAT, in der man sich Informationen über solche Fragen abholen kann! (siehe später)

Fazit:

eine große Multiuser DB braucht Memory-Allokation für:
Data Dictionary, SQL-Area, Sortier-Area, Puffer, pro User einen Arbeitsbereich

Logische Organisation:
Eine wichtige Tuning-Komponente ist das Datenbankdesign.
Generelle Faustregel: je kleiner das Datenbankschema (d.h. .je weniger Relationen), desto performanter die Datenbank. Natürlich muß man wegen gewisser Eigenheiten des Relationalen Modells viele Relationen in Kauf nehmen, die ursprünglich zu einem Objekt gehörten!
D. h. bei jeder Normalisierung ist abzuwiegen und zu testen, ob die Performance darunter leidet und wie stark!
Je mehr joins notwendig sind um eine Information zu gewinnen, desto langsamer ist die Verarbeitung!
Tuning von Anfragen kann auch über das Anlegen von Indexen geschehen: diejenigen Attribute, die sehr oft in where-Bedingungen vorkommen, sollten indexiert werden! Entweder in einem oder in verschiedenen Indexen!

zu 3) Der Client

Graphische Anwendungen brauchen viel Memory. Außerdem benutzen sie unter Umständen eine eigene SQL-Engine, die die select-statements vorcompiliert.
D. h. man muß aufpassen, daß die Client-Rechner genügend Memory und Prozessorleistung haben

zu 4) Das Netz

Netzkonfiguration spielt eine wichtige Rolle! Falsch eingestellte Roter, Bridges, ... , Packetgröße führen zu Kollisionen und Verlangsamung!

zu 5) Der Server

Datenbanken sind "Ressourcenfresser".

- Bei normalen kaufmännischen Anwendungen rechnet man mit mind. 2MB RAM pro User für den Server. Bei Multimedia-Anwendungen muß entsprechend mehr vorhanden sein!
- Es gibt viele OS-Dateien für verschiedene Zwecke:
  Datendateien, Logdateien, Indexdateien, ...
  D. h. für die Installation eines Datenbanksystems ist es bzgl. Performance eher günstig, viele kleinere Platten zu haben, und die verschiedenen Dateien darauf zu verteilen, als wenige große Platten! Insbesondere sollten Index-, Daten und Logdateien immer auf verschiedenen Platten liegen

zu 6) Zusammenarbeit zwischen Betriebssystem und DB

Datenbank und Betriebssystem (OS) müssen aufeinander abgestimmt sein:
z. B. die Größe der Datenbankblöcke sollte ein Vielfaches der Blockgröße des OS sein!
weiter gibt es eine Reihe von OS-Kernelparametern, die bzgl. eines Datenbanksystems neu eingestellt werden müssen:

- Anzahl der Prozesse pro Benutzer
- Größe der maximalen Memory-Segmente
- Anzahl der Semaphoren
- Prioritäten von Datenbank-Hintergrundprozessen

zu 7) Log. Vorb. zwischen Server und Client

Es gibt immer mehrere Möglichkeiten, Anfragen an eine Datenbank zu stellen.
Sehr oft werden die Programme und Tools benutzt, die ein Hersteller anbietet. Diese sind aufeinander abgestimmt und optimiert, so daß hier kaum noch Performance-Verbesserungen möglich sind!
Bei Oracle z. B. Developer 2000, SQLNet
Oft werden aber auch Applikationen mit anderen Tools geschrieben oder es werden 3GL-Programme benutzt zur Kommunikation mit einem Datenbanksystem.
Dabei sind verschiedene Wege möglich:

- Sind verschiedene Datenbanken im Spiel, die von einer Applikation bedient werden sollen, bleibt nur ODBC oder JDBC, d. h. ein definierter SQL-Schnittstellen-Standard (quasi ein kleinster gemeinsamer Nenner des SQL-Sprachumfangs aller beteiligten DB-Systeme (ACCESS, ORACLE, DB2, ...)).
  ODBC stellt eine Reihe von Funktionen in C, JDBC in Java zur Verfügung, mit Hilfe derer man Datenbank-Abfragen formulieren kann!

- Ist nur ein Datenbank-Hersteller beteiligt, kann man die von ihm angebotenen API's benutzen (bei Oracle: OCI, Oracle Call Interface für C)

- Datenkommunikation unter Windows z. B. zwischen Datenbank und Textverarbeitung oder Tabellenkalkulation kann auch über DDE hergestellt werden (Oracle bietet ein DDE-Package als API an)

- Es gibt noch andere Möglichkeiten: Embedded SQL:
  Vorcompilierte Statements werden in 3GL-Programme eingebunden.

Die angegebene Reihenfolge entspricht auch ungefähr dem Performanceverhalten: Am schnellsten sind die Tools eines Herstellers, am langsamsten ist vermutlich Embedded SQL!

Einschub: DB-Tuning Oracle-spezifisch:

1) Tuning des DD-Cache:

Tabelle: V$ROWCACHE

Attribute: gets: Zugriff aufs DD
getmisses: fehlgeschlagene Zugriffe aufs DD über den Cache (SGA)

Statement: select sum (getmisses) / sum (gets) x from V$ROWCACHE;

Bewertung: Wert sollte < 0,15 sein! (15%)
falls nicht: shared_pool_size vergrößern

Im Test: x = 0.01403394, d. h. ca. 1%, also gut!

2) Tuning des Data-Buffer-Cache

Tabelle: V$SYSSTAT

Attribute: Name, Value und

Name: db block gets, consistent gets } zusammen die Gesamtsumme der Zugriffe auf die Daten inklusive Cache-Zugriffe

physical reads: Gesamtzahl der Datenzugriffe über Datendateien.

Statement: select name, value from V$SYSSTAT
where name in ('db block gets', 'consistent gets', 'physical reads');

Bewertung: Trefferverhältnis z
z = 1 - (physical reads / (db block gets + consistent gets))
z sollte ≥ 0,7 sein! (d. h. ≥ 70 %)

Im Test:
db block gets = 52951
consistent gets = 951705 } 1004656
physical reads = 21494

⇒ z = 0,9786 (= ~ 98%), also gut!

3) Tuning des Log-Buffer-Cache:

Tabelle: V$SYSSTAT

Attribute: Name, Value

Parameter: redo log space requests

Statement: select name, value from V$SYSSTAT
where name = 'redo log space requests';

sollte bei 0 liegen.

Der dadurch bestimmte Wert muß regelmäßig über einen längeren Zeitraum beobachtet werden!
Steigt er ständig an, heißt das, daß Prozesse auf freiwerdenden Logbuffer space warten mußten!
Reaktion: Wert von log_buffer in der Konfigurationsdatei init <SID>.ora erhöhen

Im Test: 5

4) Tuning der Rollback-Segment-Nutzung:

a) Richtlinie für die Anzahl von Rollback-Segmente:

n = # konkurrierende Transaktionen	Anzahl Rollbacksegmente
n < 16	4
16 < n <32	8
n > 32	n/4, aber ≤ 50

b) Zugriffskonkurrenz auf die Rollbacksegmente:

Tabelle: V$WAITSTAT

Attribute: class, count

Parameter: system undo header
system undo block
undo header
undo block

Statement:
```
select class, count from V$WAITSTAT
where class in ('system undo header', 'system undo block', 'undo
header', 'undo block');
```

Die Werte von count bzgl. dieser Klassen müssen verglichen werden mit dem Wert der gesamten Zugriffe (db block gets + consistent gets aus der Tabelle V$SYSSTAT):

Bewertung:
Liegt der count-Wert einer Klasse über 1 % der Gesamtzugriffsanzahl, müssen mehr Rollbacksegmente erzeugt werden.

Im Test: db block gets + consistent gets = 1 004 656
1 % davon: 10 046,56

alle count-Werte sind 0, also braucht nichts gemacht zu werden!

III.7 Datenbanken in Client-Server-Umgebungen

In diesem Kapitel befassen wir uns mit einem besonderem Einsatzgebiet zentraler Datenbanksysteme: In einer Client-Server Architektur in einem heterogenen Rechnernetz.

Gegeben ist also eine Reihe von Rechnern (hosts), die über ein Netz entsprechend dem OSI-Modell verbunden sind. Jeder host kann also Server oder Client sein, je nachdem, ob er Dienste im Netz anbietet oder nutzt.

Ein Datenbankserver ist dann also ein host (Server), der Datenbankdienste (persistentes Speichern und konsistentes Verwalten von Daten) im Netz anbietet.

Ein Datenbankclient nutzt als Netz-Client die angebotenen Datenbankdienste eines (oder mehrerer) Datenbank-Server.

III.7.1 Remote Database Access RDA

Die spezielle Kommunikation zwischen Datenbank-Server und Datenbank-Client in einem Netz ist von der ISO 1993 in einer ersten Version als Remote Database Access (RDA) als internationaler Standard verabschiedet worden.
RDA spezifiziert Kommunikationsdienste und Kommunikationsprotokolle zum flexiblen Zugriff von Clients auf entfernte Datenbanken in heterogenen Netzen; allerdings nur auf der logischen Ebene! (d. h. die logische Struktur, die Funktionalität und das Aufrufprotokoll). Die konkrete Beschreibung der Schnittstellen (call interface) oder konkrete Implementierung ist nicht Teil dieses Standards.

Der ISO-RDA-Standard besteht aus zwei Teilen:

a) generischer RDA:
   Hier sind die Funktionen zum Initialisieren, Öffnen und Schließen einer Session, zum Anstoßen der Ausführung, zum Commit usw. beschrieben, die unabhängig vom Typ eines eingesetzten DBMS gelten.
b) spezifischer RDA:
   Hier ist beschrieben, welche SQL-Syntax unterstützt wird (im Prinzip ein Verweis auf den SQL-Standard).

Typischer RDA-Ablauf:

- Initialize — generische RDA
- Open
- Begin Transaktion
- Execute
  - select ...., fetch ... — spezifische RDA
  - insert ....., update...., delete
  - .....
- Status
- commit
- close

Die Implementierung eines RDA ist in den Schichten 5 (session layer) und 6 (presentation layer) des OSI-Modells angesiedelt und bedeutet die Installation zusätzlicher Software auf Client- und Serverseite.

⇒ ein Datenbank-Server ist ein Netzserver mit Datenbankdiensten und Serverseitiger RDA-Software.
Ein Datenbank-Client ist ein Netz-Client mit RDA-Software

• Herstellerabhängige RDA-Software am Beispiel ORACLE:

ORACLE realisiert den RDA in zwei Produkten:
- protocol adapter in Schicht 5 (session layer)
- SQLNet in Schicht 6 (presentation layer)

Auf Schicht 7 gibt es dann Interfaces zu Oracle tools (Client-Seite) bzw. zum Oracle RDBMS (Serverseite).

Außerdem stellen die protocol adapter ein Interface zur Schicht 4 (Transportsicht) zur Verfügung, um mit verschiedenen Protokollen arbeiten zu können (TCP/IP, SPX/IPX, ...)

Ein ORACLE-Datenbank-Client muß also folgende Software installiert haben:
- Protocoladapter
- SQLNet
- Oracle Anwendungssoftware (SQLPlus, Forms, OCI, JDBC/OCI, ...)

Ein ORACLE-Datenbank-Server hat folgende Software installiert:
- ORACLE RDBMS
- Protocoladapter
- SQLNet

Bestandteil der ORACLE-Server-RDA-Software ist ein sogenannter Listenerprozeß, der einen bestimmten Port abfragt und dort ankommende requests als Zugriffsversuche auf das RDBMS interpretiert.
Für jeden solchen von einem DB-Client ausgesandten request startet der Listener einen eigenen Serverprozeß, der dann die Sitzung des Client mit dem RDBMS darstellt bzw. unterhält.
Dieser Listenerprozeß muß in einer eigenen Datei konfiguriert werden.

Beispiel:

listener.ora:

```
audi = (ADDRESS 0 (Protocol = TCP) (Host = <name>) (port = 1521))
sid_list_audi =  (SID_DESC = (SID.NAME = <Instanz-Name>)
                 (ORACLE_HOME = <absoluter Pfad>))
```

Hier wird ein Listener Namens 'audi' konfiguriert, der den Port 1521 bewacht.

Eine weitere wichtige Datei ist tnsnamens.ora, in der die Alias-Namen der ORACLE-Datenbank festgelegt sind, die von Clients bzw. im Falle einer verteilten Datenbank (siehe später) von beteiligten anderen RDBMS für Datenbank-Links benutzt werden.

Auf der Client-Seite muß ähnliches konfiguriert werden (natürlich kein Listener). Dies geschieht bei Win95 mit dem Tool Easy SQLNet.

• Herstellerübergreifende RDA-Software

Natürlich muß es in einem heterogenen Netz möglich sein, verschiedene Hersteller-RDBMS als Datenbank-Clients zu kontaktieren, ohne auf den Clients für jeden DB-Hersteller die spezifische Client-Software des RDA installieren zu müssen. Dies sind die später in Kapitel IV.2 besprochenen CALL Level Interface-API's.

Im Falle von ORACLE kann man ohne ORACLE-RDA-Software als allgemeiner Netz- Client z. B. mit dem JDBC-Thin-Treiber von Oracle mit dem ORACLE RDBMS kommunizieren:

ORACLE JDBC THIN-Beschreibung:

- Typ 4 JDBC-Treiber
- Benötigt keine zusätzliche Oracle RDA-Software auf Client-Seite
- Hat implizit eine eigene TCP/IP-Version von SQLNet
- Braucht auf Serverseite die komplette Oracle-RDA-Software
- Verbindungsaufbau:
  connection con = DriverManager.get CONNECTION
  ("jdbc: oracle: thin: @ <host>: <port>: <SID>", "<user>", "<pw>");
  ansonsten wird der Treiber wie andere JDBC-Treiber ins Programm geladen bzw. die Klassen importiert.

Für das Datenbank-Management ergeben sich damit folgende Aufgaben:

- Erstellen eines Ist-Zustand-Überblicks: welche Netzprotokolle werden verwendet, welche Datenbank-Hersteller sind im Einsatz, welche Netzclients sollen welche Datenbank-Server erreichen, ...
- Konfiguration der jeweiligen Datenbank-Hersteller-RDA-Software auf den Server

- Konfiguration der Datenbank-Clients
- Im laufenden Betrieb:
  - neue Clients konfigurieren
  - Anwendungsprogrammierer beraten und auf Spezifika hinweisen
  - Engen Kontakt zur Netzadministration halten
- Sicherheitsprobleme beachten (ist z. B. Zugriff von anderen Netzen in das Intranet möglich, müssen die Datenbank-Server besonders geschützt werden, usw. ...)

Ist im Unternehmensnetz nur ein Datenbank-Hersteller mit seiner Software aktiv, ist der spezielle Netz-Management-Aufwand bzgl. Datenbank-Clients und Datenbank-Server nach der Installation gering.
Der Aufwand steigt natürlich mit der Zahl der beteiligten unterschiedlichen Herstellersoftware.

III.7.2 Verteilungsebenen von Datenbank-Applikationen

Client-Server-Architekturen (C-S-A) sind heute in vielen Unternehmen im Einsatz. Historisch sind sie entstanden aus dem Gedanken, teure Ressourcen wie z. B. Festplattenspeicher oder Prozessorleistung zentralisiert zur Verfügung zu stellen und nicht jedes Einzelsystem damit auszurüsten. Es sollte also ein Serverrechner installiert werden, der Plattenplatz und/oder Prozessorleistung anbietet und verschiedene andere (Arbeitsplatz-) Rechner nutzen diese Dienste durch Zugriffsmöglichkeiten über ein Rechnernetz. Die ersten Systeme hatten reine Dateiserver (z. B. Novel-Netzwerke).
Weiterentwickelt wurde diese Idee in Richtung Datenbankserver oder Kommunikationsserver, die über Ressourcenkapazitäten auch sichere Datenverwaltung bzw. sichere Kommunikationsverwaltung anbieten. Wir wollen uns hier auf die Frage konzentrieren, inwieweit eine Programmierung einer Datenbank-Anwendung für eine Client-Server-Datenbank-Architektur konzeptionell berührt ist!

Unter diesem Gesichtspunkt ist interessant, daß eine C-S-A aus zwei logischen Komponenten besteht, eben den Clients, den Dienst-Nutzern und den Servern, den Dienst-Anbietern. Das sind logische Rollen die nicht Rechner-gebunden sind, sondern beide Rollen können auch von einem Rechner ausgeübt werden: es werden beispielsweise zusätzlich zu den Sachdaten auch Bilder, Karten, Graphiken in einem eigenen CD-ROM-Server gehalten. Bei einer Anfrage von einem Client aus an den Datenbank-Server werden Sach- und Bilddaten benötigt. Da wird der Datenbank-Server zum Client, der vom CD-ROM-Server das Bildmaterial abruft.

Eine Datenbank-Anwendung benutzt eine Reihe von Diensten:
Datenhaltung, Datenverwaltung, Textverarbeitung, Druckdienste, Menüsteuerung, Masken zur Dateneingabe, Trigger, usw. ...
All diese Dienste können prinzipiell auf verschiedene Rechner verteilt werden. Um das ganze zu strukturieren, unterscheidet man 5 verschiedene Verteilungsebenen einer Datenbank-Anwendung:

i)	Ebene der Präsentation (P):	Funktionen zur Anzeige am Bildschirm (Fenstertechnik, ...), Funktionen zum Drucken,...
ii)	Ebene der Steuerung (S):	Funktionen der Ablauflogik der Anwendungssituation (Menüstruktur mit den Maskenaufrufen, Dialogsteuerung mit dem Benutzer)
iii)	Ebene der Anwendungslogik (A):	alle fachlichen Funktionen: z.B. bei der KAPV:Stammdatenpflege, Rechnungen, Aufträge,... verwalten, ...
iv)	Ebene der Datenverwaltung (G):	alle Regeln, die für die fachliche Datenstruktur gelten, also die fachlichen Geschäftsregeln, die über Constraints oder Datenbanktrigger im DBMS bzw. DD eingetragen sind oder eben nicht, d. h. in der Applikation programmiert werden müssen
v)	Ebene der Datenhaltung (D):	alle Funktionen eines Datenbanksystems, realisiert in einem DBMS

Alle Funktionen einer Datenbank-Anwendung (z. B. beschrieben in einem Funktionsmodell) werden den 5 Ebenen zugeordnet. Dann hat man die Möglichkeit, aufgrund der Arbeitsorganisation im Betrieb und/oder den Hardwarevoraussetzungen (Arbeitsplatzrechner) die Anwendung bzw. ihre Funktionalitäten auf die jeweils in Frage kommende Hardware zu verteilen!

Der triviale Fall ist der, daß lediglich die Präsentationsebene auf dem Client liegt, alles andere vom Datenbank-Server übernommen wird. Dieser Fall liegt bei allen Terminal-Host-Architekturen vor:
Die Clients fungieren als X-Terminals, als reine Ein- und Ausgabestationen, alles andere liegt auf einem Datenbank-Server.

Die am häufigsten anzutreffende Form ist die, daß Präsentation und Steuerung auf den Clients liegen, Datenhaltung von einem Datenbank-Server übernommen wird und Anwendungslogik und Geschäftsregeln teilweise im Clientprogramm und teilweise im Serverprogramm (stored procedures, etc...) realisiert sind:

Client: (P, S, A, G)
Server: (A, G, D)

Natürlich kann auch die Datenhaltung über mehrere Server verteilt sein (Verteilte Datenbanken), die jeweils bei einer Anfrage als Server und/oder Client für andere Server arbeiten.

Wird eine Datenbank-Applikation für eine Client-Server-Architektur programmiert, muß sie also bzgl. ihrer Funktionen auf die 5 Ebenen verteilt werden. Dabei ist darauf zu achten, daß die einzelnen Module, die dabei entstehen, so beschaffen sind, daß sie wiederverwendbar sind, d. h. auch von anderen Applikationen genutzt werden können. Das setzt natürlich im einzelnen eine Reihe von Standardisierungsvorschriften des Unternehmens voraus:

- alle Schnittstellen zwischen den 5 Verteilungsebenen müssen klar standardisiert sein;

- Kommunikation zwischen Applikationen (z. B. DB-Applikationen und Textverarbeitung) ist fest definiert (z.B nur über DDE);
- für jede Kommunikation mit dem Datenbanksystem ist das dort angebotene API (CLI) zu nutzen, usw. ;
- Styleguide für alle Softwareentwicklungen im Unternehmen.

Erst wenn alle diese Details geregelt sind, können Programmodule geschrieben werden, die wiederverwendbar sind.

Voraussetzungen für C-S-DB-Appl.-Programmierung sind also:

i) Kenntnis der EDV-Landschaft, die zur Verfügung steht

ii) Kenntnis der Arbeitsorganisation des Bereichs, für den die Applikation programmiert wird.

iii) Kenntnis aller fachlichen Funktionalitäten

iv) Kenntnis des Leistungsumfangs der einzusetzenden Standardsoftware (Datenbanksystem, Textverarbeitung,...)

v) Kenntnis der Programmierstandards des Unternehmens

vi) Kenntnis der bisher vorhandenen Module, die eventuell nutzbar sind!

III.8 Views

Wir kehren noch einmal zur 3-Ebenen-Architektur moderner Datenbanksysteme zurück und beschreiben ein wichtiges Hilfsmittel zur Definition externer Schemata. Externe Sichten auf den eigentlichen Datenbestand des im konzeptionellen Schema definierten Datenbankdesigns können auf viele Arten erstellt werden; im Wesentlichen immer durch Beschränkung auf Teile des Bestandes in Masken oder Programmen, unabhängig vom zugrundeliegenden Datenmodell (relational, objektorientiert,...).

Auf der Datenbankseite gibt es die Möglichkeit, sogenannte Views zu definieren.

Allgemeine Syntax:

Create view <viewname> as <select-statement>

Beispiel:

```
create view kundenad as
select kdnr name, ort, plz, straße
from kunde, adresse
where kunde.kundennr = adresse.kundennr
and adrnr =   (select max (adrnr) from adresse
                where kdnr = kunde.kdnr
                group by kdnr);
```

Bei der Ausführung dieses DDL-Statements wird der Ausdruck hinter 'as' nicht ausgewertet, sondern im Data Dictionary gespeichert unter dem Namen Kundenad. bzw. user.kundenad. Bei jedem Aufruf des views z. B. in der FROM-Klausel eines select-statements wird dann der Ausdruck ausgewertet. Ein view stellt also eine virtuelle Relation dar! Konkret wird bei jedem Erscheinen des viewnamens in einem SQL-Statement dieser Name ersetzt durch den Ausdruck, der den view definiert.

Gelöscht werden views wie andere Datenbankobjekte:

Drop view <viewname>;

Wozu sind views gut?

i) Views sind ein gutes Mittel, um logische Datenabhängigkeiten zu erreichen: es kann sein, daß das Datenbankdesign geändert werden muß, Attribute gewisser Relationen werden auf neue Relationen verteilt. Dabei kann der Fall entstehen, daß die ursprüngliche alte Relation jetzt ein join gewisser neu entstandener Relationen ist. Wird jetzt also ein view mit dem Namen der alten Relation erzeugt, braucht kein Anwendungsprogramm geändert werden!
(Beispiel:
ursprünglich waren Kunden- + Adressdaten in einer Tabelle gespeichert; dann erfolgte eine Änderung der Geschäftsregel: mehrere Adressen sind zulässig. Folge: Die Adressen der Kunden müssen in einer eigenen Tabelle gehalten werden. Damit bereits existierende Anwendungsprogramme nicht geändert werden müssen, wird nun ein view erzeugt, der das Vorhandensein der früheren Kundentabelle mit jeweils einer Adresse vorgaukelt!)

ii) Views beschränken die Sicht des Anwenders auf die Attribute, zu deren Änderung er autorisiert ist!
Beispiel:
Eine Mitarbeiterrelation mit Stammdaten, Firmendaten wie Eintrittsdatum, Gehalt, usw. soll nur von einem Sachbearbeiter bearbeitet werden können. Also bekommen andere Sachbearbeiter views auf dieser Mitarbeiterrelation, die z.B. nicht die Firmendaten enthalten!

Das große Problem bei der Verwendung von Views sind die Update-Operationen (insert, delete, update) darauf. Zu diesem Problem gibt es eine umfangreiche Literatur.

View Update-Mechanismen müssen einer Reihe von Prinzipien genügen. Einige sollen exemplarisch auflistet werden:

- Update eines view muß unabhängig vom konkreten (SQL-) Ausdruck sein, der den view definiert!

  Beispiel:

  i) create view V1 as
  select * from kunde where name like 'M%'
  or gebdat y '01.01.69';

ii) create view V2 as
select * from kunde where name like 'M%'
UNION
select * from kunde where gebdat < '01.01.69';

Die views V1 und V2 sind semantisch äquivalent und sollten aktualisierbar sein, allerdings würden viele SQL-Produkte heute den view V2 nicht aktualisieren können!

- updates (modify) sollten mittels delete-then-insert gemacht werden
- ist ein view aktualisierbar, müssen alle update-Operationen erlaubt sein (delete, insert, update)

Betrachten wir zunächst den einfachsten Fall: Ein view ist durch eine Tabelle definiert. Dann existieren kaum Probleme falls der Primärschlüssel als Attribut auch zum view gehört.

Der nächste, bereits wesentlich schwieriger zu behandelnde Fall ist der, daß das den view definierende select-statement UNION, INTERSECT oder EXCEPT enthält.
In diesem Fall gelten die folgenden Regeln:

Es seider view V definiert durch den Ausdruck A UNION B, A INTERSECT B, oder A EXCEPT B, mit A, B beliebige SQL-Ausdrücke bzw. select-statements.

Betrachten wir exemplarisch den Fall UNION: A UNION B

insert-Regel: Das neue Tupel muß den Ausdrücken A oder B oder beiden genügen. Genügt es A, wird es in A eingefügt, B analog.

Beispiel:
Gegeben sei der view V2 von eben:
Ein insert der Form: insert into V2 (kdnr, name) values (2111, 'Meier');
erfüllt die erste Komponente (A), wird also in A eingefügt

Was aber ist mit einem insert der Form:
Insert into V2(kdnr, name, gebdat) values (4711, 'Schmidt', '12.10.71')?
Dieses insert erfüllt weder die Komponente A, noch B, kann also nicht eingefügt werden!

Delete-Regel: gehört das zu löschende Tupel zu A, wird es aus A entfernt. Ist es (immer noch) in B, wird es aus B gelöscht.

Update-Regel: das geänderte Tupel muß A oder B oder beiden genügen. Dann wird es aus A oder B entfernt und neu eingefügt.

bzgl. INTERSECTION und EXCEPT gelten die analogen Regeln, bezogen auf die logischen Verknüpfungen der Mengenoperatoren.

Der schwierigste Fall ist der von Views, die durch einen join mehrerer Tabellen entstehen.

Betrachten wir als Beispiel den view kundenadr vom Beginn dieses Abschnitts:

```
Create view kundenad as
select kunde.kdnr, name, ort, plz, str
from kunde, adresse
where kunde.kdnr = adresse.kdnr
and adrnr = (select max (adrnr) from adresse
             where kdnr = kunde.kdnr
             group by kdnr)
```

Dieser view faßt also alle Datensätze aus den Tabellen KUNDE und ADRESSE zusammen, die den where-Bedingungen genügen. Beispielhaft sei er wie folgt dargestellt:

Kundenadr.:

kdnr	**name**	**ort**	**plz**	**str**
1	Meier	Mist	54321	Miststr.
⇒ A		⇒ B		

Kunde:

kdnr	**name**	**Vorname**	**gebdat**	**geschl**
1	Meier	Horst	1.1.59	M

Adresse:

adnr	**plz**	**ort**	**str**	**staat**	**kdnr**
1	12456	Hwl	Teststr	D	1
2	54321	Mist	Miststr	D	1

Allgemein: Es sei der view V entstanden aus dem Ausdruck A join B
Dann sind folgende Regeln vorgegeben:

insert-Regel: Das neue Tupel t muß dem join-Ausdruck A join B genügen (d. h. der eigentliche join-condition (längs gewisser Attributwerte) und eventuell zusätzliche Bedingungen in A und/oder B!)
Kommt der A-Teil von t nicht in A vor, wird er dort eingefügt; analog B

delete-Regel: Der A-Teil des zu löschenden Tupels t wird aus A gelöscht oder der B-Teil aus B oder beides, in Abhängigkeit von der join-Bedingung und den Bedingungen des delete-Statements.

update-Regel: Die geänderte Version des Tupels t muß dem join-Ausdruck genügen (siehe insert). A- und B-Teile von t werden gelöscht, ohne checks.
Existiert dann der A-Teil der geänderten Version von t nicht in A, wird er in A eingefügt, analog dem B-Teil. (deshalb: delete then insert!)

Beispiel:

Betrachten wir wieder den view kundenadr, erzeugt durch den Ausdruck A join B:

```
select kdnr name, ort, plz, straße
    from kunde, adresse
    where kunde.kundennr = adresse.kundennr
    and adrnr = (select max (adrnr) from adresse
                    where kdnr = kunde.kdnr
                    group by kdnr);
```

insert:

```
insert into kundenadr values (4711, Müller, Trier, Null, Null);
```

4711, Müller gehört zum A-Teil des join, Trier, Null, Null zum B-Teil

Falls Primär-Fremdschlüssel-Constraints auf Kunde und Adresse definiert sind, kann dieses Statement nicht ausgeführt werden!
falls NOT NULL-Constraints auf Spalten bestehen, die nicht in view stehen, kann das statement ebenfalls nicht ausgeführt werden.

In den view kundenadr kann also theoretisch nur dann ein Tupel eingefügt werden, wenn alle Primärschlüssel aller beteiligten Tabellen auch als view-Attribute vorhanden sind und ebenso alle NOT NULL-Attribute der Tabellen!

delete:

```
delete from kundenadr
where name = 'Meier' and ort = 'wismar';
```

alle Meiers mit ihren über die join-condition erfaßten Adressen könnten gelöscht werden (aus Kunde und Adresse).

update:

```
update kundenadr
set ort = 'Bonn'
where ort = 'Wismar`;
```

alle Tupel, deren Ort-Attribut den Wert 'Wismar' hat, werden die join-Bedingung A join B auch nach der Änderung erfüllen. d. h. das update könnte vollzogen werden?
Genauer:
sei z. B. t = ('Schulze', 'Wismar', 23966, 'Lübsche Str. 24')

'Schulze' gehört zum A-Teil, 'Wismar', 23966, 'Lubsche Str. 24' zum B-Teil.

Der Kunde Schulze (mit Wohnort Wismar) könnte aus Kunde gelöscht, die Adresse Wismar, usw.... könnte aus Adresse gelöscht werden.
Eingefügt werden kann nichts, siehe oben!
D.h. insgesamt ist kein update möglich, da ja nach dem Prinzip 'delete then insert' vorgegangen werden muß!

Fazit:
Der betrachtete view ist nicht aktualisierbar! Da das update-Problem bzgl. views so kompliziert und nur aufwendig zu realisieren ist, ist auch längst nicht jedes Datenbank-Produkt in der Lage, Datenmanipulationen auf Views zuzulassen. Das hängt vom jeweiligen Hersteller ab!

Oracle-Views

1. Einfache Ein-Tabellen views:

Syntax:
Create view <viewname> as <select-statement>
[with check option constraints <constrain.name>];

Die 'with check option' wirkt wie folgt:
Bei inserts und updates werde alle Aktionen zurückgewiesen, die die where-Klausel des select-statements des Views verletzen.
Ist das Constraint nicht gesetzt, könne auch Datensätze eingefügt werden, die die where-Bedingung verletzten!

Dies gilt nur für Ein-Tabellen Views, join views mit check option können nicht aktualisiert werden!

Beispiel:

```
create view ktest as
select kdnr, name, vorname from kunde
where name like 'M%'
with check option constraint ktestcheck;
```

Ein select-statement der Form: select * from ktest; erbringt alle Kunden mit kdnr, name, vorname, deren name mit M beginnt

```
insert into ktest values (474, 'Wurst', 'Hans');
```

ohne check constraint: wird eingefügt!
mit check constraint: wird abgewiesen!

2. join-views:

Oracle gibt einen ganzen Satz von Bedingungen an das definierende select-Statement; dieses darf
- keinen distinct-Operator
- keine Gruppenfunktionen
- keine group by, having Klausel
- kein start with, connect by

enthalten!

Weitere Bedingungen sind:

- Update-, insert-, delete-statements dürfen jeweils nur auf eine Tabelle des joins wirken!

- Nur solche Spalten können geändert werden bzw. neu erzeugt werden, die zu einer Tabelle gehören, deren Primärschlüssel auch Primärschlüssel des Views ist (sein kann). Sogenannte Key-preserved tables.

Es gibt eine Data Dictionary-Sicht 'user_updatable_columns', in der alle Spalten aller Tabellen und Views aufgelistet sind, die prinzipiell aktualisierbar sind! Das bedeutet allerdings nicht, daß sie tatsächlich aktualisierbar sind, wenn sie z.B. zu nicht-key-preserved-tables gehören!

Delete und insert in einem join view sind möglich, falls der join nur eine key-reserved-table enthält.

Kapitel IV Prozedurale Erweiterungen von SQL

Wir haben bisher die Sprache SQL kennengelernt als einzige Möglichkeit, mit einem relationalen Datenbanksystem zu kommunizieren. Diese Sprache SQL enthält aber weder in der Standardisierung noch in herstellerabhängigen Derivaten Möglichkeiten zur Implementierung prozeduraler Abläufe wie Schleifen, bedingte Anweisungen usw... Andererseits besteht natürlich die Notwendigkeit, SQL-Operationen im Rahmen von Datenbankanwendungen in prozedurale Strukturen einbetten zu können. Prinzipiell gibt es dabei zwei Möglichkeiten: entweder SQL selbst um prozedurale features zu erweitern oder die Möglichkeit zu schaffen, in einem beliebigen 3GL-Programm Verbindung zu einer Datenbank herzustellen, SQL-statements zu formulieren und an die Datenbank zu schicken und schließlich die Ergebnisse der Datenbankoperation im 3GL-Programm weiterzuverarbeiten.
Wir werden hier beide Möglichkeiten behandeln.

Wir beginnen mit der ersten Möglichkeit: die Erweiterung von SQL um prozedurale features.

In der SQL-Standardisierung für Prozedurale Erweiterung von SQL, SQL3 / PSM (Persistent stored Modules), sind die wichtigsten prozeduralen Konstrukte definiert. Ausgangspunkt ist hier die Deklaration einer Prozedur bzw. einer Funktion:

Syntax:

```
create Procedure / Function
<name> ([<parameter-definition-kommaliste>])
[Returns <data-type>]
<SQL-Statement>;
```

mit <parameter-definition>::= [IN | OUT | INOUT] <parametername> <Datentyp>

<SQL-Statement> : besteht aus einem einzigen SQL-Statement:
entweder ein SQL DML-Statement (insert, update, delete, select into)
oder ein SQL Control-Statement.

SQL-Controlstatements sind zum Beispiel Return, Case, If-Then-Else-Blöcke, Loops, While- oder For-Schleifen, usw. ... SQL-Controlstatements müssen in einen Begin-End-Block eingebettet werden!

Auch eigene SQL-Variable können in einer sogenannten Declare-section deklariert werden:

```
Declare
    <SQL-Variablen-Kommaliste> (mit Angabe des Datentyps).
```

Ebenso ist ein Exception Handling vorgesehen.

Um solche prozeduralen Erweiterungen konkret kennenzulernen, folgt jetzt eine ausführliche Beschreibung der spezifischen Erweiterungen von SQL, die der Datenbankhersteller Oracle für sein System anbietet: PL/SQL. Mit Hilfe dieser erweiterten Sprache kann man dann Datenbanktrigger, Prozeduren und Funktionen und explizite Cursor anlegen. Gespeicherte PL/SQL-Prozeduren können von 3GL-

Programmen, SQLPlus oder anderen Oracle-Modulen aufgerufen werden. Damit ist also die Möglichkeit gegeben, Programmfunktionalität (fachliche Funktionen der Anwendung) unter die Kontrolle des Datenbanksystems zu stellen!

IV.1 PL/SQL von Oracle

PL/SQL ist eine Block-orientierte Sprache, d. h. syntaktisch werden SQL und PL/SQL in (eventuell verschachtelte) Blöcke eingebettet.

Syntax eines PL/SQL-Blocks:

```
[DECLARE
    <Deklarationsteil>]
BEGIN
    <Statements (PL/SQL oder SQL-DML)> (Ausführbarer Teil)
[EXCEPTION
    <Exception-Handling>]
END;
```

Obligatorisch ist also nur die Begin-End-Klammer, die anderen Blockelemente können bei Bedarf weggelassen werden.

IV.1.1 Sprachelemente

Die Sprachelemente werden wir in der Reihenfolge des Aufbaus eines PL/SQL-Blocks behandeln.

A) Deklarationsteil:

Im Deklarationsteil werden unter anderem alle benutzten (lokalen) Variablen des PL/SQL-Blocks deklariert, aber auch Prozeduren, Funktionen, Cursor und user-definierte Exceptions.
Dem Deklarationsteil vorangestellt ist das Schlüsselwort DECLARE:

Syntax:

```
DECLARE
    <Variablenname>   <Datentyp>;
    ...
```

Als Datentyp stehen alle SQL-Datentypen zur Verfügung (char, varchar2, date, boolean, number, ...)
Im Deklarationsteil sind auch Initialisierungen der Variablen und NOT NULL-Beschränkungen möglich.

Beispiele:

```
DECLARE
    Text varchar 2 (40);
    Testzahl number (4) := 7;
    Testwert varchar 2 (40) NOT NULL := 'Test';
```

Wenn allerdings NOT NULL-Beschränkungen deklariert sind, müssen die zugehörigen Variablen auch initialisiert werden! Die folgende Variablendeklaration ist also falsch!

```
DECLARE
    testwert varchar 2 (40) NOT NULL;
```

Sollen Konstante deklariert werden, muß das Keyword CONSTANT verwendet werden:

```
DECLARE
    Pi  constant  real := 3.14159;
```

Neben diesen Standardtyp-Deklarationen gibt es zwei Datenbank-spezifische:
Die Attribute %TYPE und %ROWTYPE

%TYPE:

Kann benutzt werden, um einer Variablen den gleichen Datentyp einer anderen Variablen oder einer Tabellenspalte zuzuordnen:

Beispiele:

```
DECLARE
    x       number (2);
    y       x%TYPE
    test    Kunde.name%TYPE
```

Die zweite Deklaration ordnet der Variablen y den Datentyp von x zu
Die dritte Deklaration ordnet der variablen test den Datentyp des Attributs name der Tabelle Kunde zu.

%ROWTYPE:

Deklariert eine Variable als Datensatztyp von der gleichen Struktur wie eine bestehende Tabelle.

Beispiel:

```
DECLARE
    kunde_rec     kunde%ROWTYPE
BEGIN
    select * into kunde_rec from kunde where kundennr = 1
END;
```

Hier bekommt die PL/SQL-Variable kunde_rec die gleiche Struktur wie die Tabelle Kunde! Im select-statement wird dieser Variablen dann als Wert ein Datensatz aus der Tabelle Kunde zugewiesen.

Die einzelnen Attribute (Felder) der Datensatz-Variablen können über die Punkt-Notation angesprochen werden:
kunde_rec.name enthält den Kundennamen aus dem select-Statement;
kunde_rec.gebdat enthält das Geburtsdatum aus dem select-Statement
usw. .....

Zuweisungen von Werten zu einer Variablen vom %ROWTYPE-Typ können nur über select-Statements gemacht werden (auch teilweise!).
Direkte Zuweisungen gehen nicht.
Die Zuweisung Kunde_rec.name:= 'Meier' erzeugt also einen Fehler!

Mit %ROWTYPE wird eine Variable als Record deklariert, der die gleiche Struktur trägt wie die bei %ROWTYPE spezifizierte DB-Tabelle. Dieser Record kann nicht modifiziert werden!

Man kann aber auch eigene Record-Datentypen definieren:

Syntax:

```
TYPE <name> IS RECORD
(<Feldname1>  <Datentyp>, <Feldname2>  <Datentyp>, ....)
```

Beispiel:

```
DECLARE
        TYPE Krec IS RECORD
        (Kname Kunde.name%TYPE, Kvorname char (1));

        kunden_rec Krec;

BEGIN
        select name, substr (vorname, 1.1) into Kunden_rec
        from Kunde where Kundennr = 1;
END;
```

Hier wird also zunächst ein eigener Recordtyp mit Namen Krec deklariert und anschließend eine Variable kunden_rec mit eben diesem Recordtyp.
Im ausführbaren Teil wird dann der Variablen kunden_rec über das select into-statement wieder ein Datensatz zugewiesen.
Bei eigenen definierten Recordtypen sind auch Zuweisungen möglich:
kunden_rec.kname := 'Meier';

Insgesamt kann ein Deklarationsteil enthalten:

- Variablen-Deklarationen
- user-spezifische Exception-Deklarationen
- Prozedur-/Funktions-Deklarationen
- Cursor-Deklarationen

} später!

B) ausführbarer Teil:

Innerhalb der BEGIN-END-Klammer stehen eine Reihe von PL/SQL-Statements, jeweils abgeschlossen durch ';'.

Mögliche PL/SQL-Statements sind:

i) SQL-DML-Statements (insert, update, delete, select into)
ii) PL/SQL-Block
iii) Zuweisungs-Statements
iv) IF-THEN-ELSE-Statements
v) Loop-Statements
vi) GOTO-Statements
vii) Prozeduraufrufe
viii) Return-Statements
ix) Raise-Statements

Nicht erlaubt in einem PL/SQL-Block sind DDL-Statements (create, drop, ...), diese müssen vorher formuliert werden!!

Beispiel:

i) select auftragsdat from auftrag; — SQL-statement

ii) PL/SQL-Block:

```
Declare
   adat date
Begin
   select auftragsdat into adat from auftrag;
END;
```

In SQL-Plus wird ein PL/SQL-Block (oder procedure-Block) ausgeführt, indem als letzte Zeile ein '/' angegeben wird.
Das bewirkt, daß der Block in den SQL-Puffer geladen und ausgeführt wird.

Zu den einzelnen PL/SQL-statements:

i) SQL-DML-Statements:
Zugelassen innerhalb eines PL/SQL-Blocks sind die normalen SQL-Operationen insert, update und delete. Das select-statement innerhalb eines PL/SQL-Blocks muß eine into-Klausel enthalten. Normale select-statements sind nicht erlaubt!

Beispiel:

```
DECLARE
        adat date;
BEGIN
        select auftrag.adatum into adat from auftrag
        where anr = 5002;
        update kunde set name = 'Meier' where kdnr = 1010;
        delete from adresse where adrnr = 12345;
END;
```

ii) PL/SQL-Block
Innerhalb eines PL/SQL-Blocks können weitere PL/SQL-Blöcke geschachtelt werden.

iii) Zuweisungs-Statements
Zuweisungs-Statements haben die Syntax:

```
<Variable> := <Wert>;
```

Beispiel:

```
DECLARE
        Test varchar2(20);
BEGIN
        Test := 'ein beliebiger Wert';
END;
```

iv) Bedingte Ausführung von Statements
Es gibt 3 Formen:

IF-THEN:
Syntax:

```
IF <Bedingung> THEN <Folge von Statements>;
END IF;
```

nicht: ENDIF!

<Bedingung> ist ein logischer Ausdruck, eventuell auch mit AND, OR, NOT gebildet. ist <Bedingung> vom Wert FALSE oder NULL, wird die Statementfolge nicht ausgeführt.

IF-THEN-ELSE

Syntax:

```
IF <Bedingung> THEN <1. Folge von Statements>;
     ELSE <2. Folge von Statements>;
END IF;
```

Hat <Bedingung> den Wert FALSE oder NULL, wird die 2. Folge von Statements ausgeführt.

Innerhalb der Statementfolgen 1 oder 2 können weitere IF-Statements stehen!

IF-THEN-ELSIF

Syntax:

```
IF <Bedingung 1> THEN <1. Folge von Statements>;
      ELSIF <Bedingung 2> THEN <2. Folge von Statements>;
   [ELSE <3. Folge von StatementsA>;]
END IF;
```

Hat Bedingung 1 den Wert FALSE oder NULL, wird Bedingung 2 getestet;
Hat Bedingung 2 den Wert TRUE, wird die 2. Folge von Statements ausgeführt.
Hat Bedingung 2 den Wert FALSE oder NULL, wird die 3. Folge von Statements ausgeführt.

Ein IF-Statement kann eine beliebige Anzahl von ELSIF's haben.

Die letzte ELSE-Klausel ist optional.

Wichtig: Ist irgend eine Bedingung TRUE, wird die zugehörige Statementfolge ausgeführt und das IF-Statement verlassen! Es ist also egal, was in eventuell folgenden ELSIF-Statments für Bedingungen stehen!

Beispiele:

1)
```
DECLARE
    bland char (3);
    Küste number (3);
BEGIN
    Küste := 0;
    select bundesland into bland from adressen
    where kundenr = 1 and postadr = 'TRUE';
    IF bland in ('NS', 'HH', 'SH', 'MV') then Küste := Küste + 1;
    END IF;
END;
```

2)
```
DECLARE
    bland char (3);
    Küste number (3);
    rest number (3);
BEGIN
        Küste := 0; rest := 0;
        select bundesland into bland from adresse
        where kundenr = 1 and postadr = 'TRUE';
        IF bland in ('NS', 'HH', 'SH', 'MV') THEN
            Küste := Küste +1
        ELSE
            rest := rest + 1
        END IF;
END;
```

3)
```
DECLARE
    bland char (3);
    Küste number (3);
    rest number (3);
    süden number (3);
BEGIN
        Küste := 0; rest := 0; süden := 0;
        select bundesland into bland from adresse
        where kundenr = 1 and postadr = 'TRUE';
        IF bland in ('NS', 'HH', 'SH', 'MV') THEN
            Küste := Küste +1
        ELSIF bland in ('BW', 'BAY') THEN
            süden := süden + 1
        ELSE
            rest := rest + 1
        END IF;
END;
```

v) Iterierte Ausführung von Statements: Loops

Es gibt in PL/SQL drei Arten von Loops:

Loop

Syntax:

```
LOOP
   <Folge von Statements>;
END LOOP;
```

Beispiel:

```
BEGIN
    LOOP
       insert into Kunde (kundennr, name)
       select kunde_seq.next val, 'Meier' from dual;
    END LOOP;
END;
```

Problem: Die obige Schleife hört nicht auf! Man braucht ein **exit-Statement**!

Beispiel:

```
DECLARE
    zahl number (2);
BEGIN
    zahl := 0;
    LOOP
       insert into kunde (kundennr, name)
       select kunde_seq.nextval, 'Meier' from dual;
       IF zahl = 20 THEN EXIT;
       END IF;
       zahl := zahl + 1
    END LOOP;
END;
```

Es gibt noch eine zweite Art des EXIT's: EXIT WHEN:

Beispiel wie eben:

```
        statt IF zahl = 20 THEN EXIT;
        END IF;

        EXIT WHEN zahl = 20;
```

EXIT-Statements sind nur innerhalb eines Loops erlaubt! Zum Verlassen eine PL/SQL-Blocks muß man das RETURN-Statement verwenden (später)!

Einschub: Lables

LOOPS und PL/SQL-Blöcke können mit Label versehen werden.

Syntax:

ein nichtdeklarierter Identifier in doppelten Klammern:
<<mein_label>>

Label müssen vor dem Block oder Loop stehen, der benannt werden soll und können zusätzlich auch am Ende des Loops stehen (zur Übersicht bei verschachtelten Loops).

Beispiel:

```
<<Schleife 1>>
LOOP
    ....
    <<Schleife 2>>
    LOOP
    ....
    END LOOP Schleife 2;
    ....
END LOOP Schleife 1;
```

Mit dem EXIT-Statement können beliebige Schleifen beendet werden:

```
<<Schleife 1>>
LOOP
    ....
    <<Schleife 2>>
    LOOP
    ....
    EXIT Schleife 1 WHEN.... ← beendet beide Loops!
    ....
    END Schleife 2;
    ....
END Schleife 1;
```

WHILE LOOP

Syntax:

```
WHILE <Bedingung> LOOP
    <Folge von Statements>;
END LOOP;
```

Beispiel:

```
DECLARE
    zahl number (3);
BEGIN
        zahl := 0;
        WHILE zahl < 20 LOOP
            insert into kunde (kundennr, name)
            select kunde_seq.nextual, 'Meier' || zahl from dual;
            zahl := zahl + 1;
        END LOOP;
END;
```

FOR LOOP

Syntax:

```
FOR <zähler> IN [REVERSE] <untere Schranke> .. <obere Schranke> LOOP
    <Folge von Statements>;
END LOOP;
```

Die Schranken (oben und unten) können konstante Zahlen, Variable oder Ausdrücke sein Zähler-Variablen müssen nicht explizit deklariert werden, sie können direkt benutz werden!

Beispiele:

1)
```
FOR zahl in 1..20 LOOP
        insert into kunde (kundennr, name, vorname)
        select kunde_seq.nextval, 'Meier' || to_char(zahl), vorname from dual;
END LOOP;
```

2)
```
select count (name) into name_zahl from kunde;
FOR I IN 1..name_zahl  LOOP
        update kunde set name := name || i
        where kundennr = i;
END LOOP;
```

Damit können Schranken also auch dynamisch gesetzt werden!

3)
```
<<Schleife 1>>
FOR I IN 1...20 LOOP
      <<Schleife 2>>
      FOR I IN 1...10 LOOP
        IF Schleife 1.I > 15 THEN EXIT Schleife 2;
        END IF;
      END LOOP Schleife 2;
END LOOP Schleife 1;
```

Schleifenzähler können also gleiche Namen haben, sie werden jeweils neu initialisiert. Um den Richtigen zu referenzieren, muß ein Label mit Dot-Notation benutzt werden!

vi) GOTO-Statement

Oracle bietet GOTO-Statements an, die es erlauben, zu bestimmten Statements oder PL/SQL-Blöcken zu springen.
Unmittelbar nach dem Label muß ein ausführbares Statement stehen

Beispiel:

```
1)  BEGIN
        ....
        GOTO insert_row
        ...
        <<insert_row>>
        insert into...
    END;

2)  DECLARE
      raus boolean;
    BEGIN
      ..
      FOR I in 1...100 LOOP
        IF raus then GOTO end_loop;
        END IF;
      ...
        <<end_loop>>
        NULL;
      END LOOP;
    END;
```

Das NULL-Statement bedeutet hier: gehe weiter und tue nichts.

Da die Verwendung von GOTO-Statements unschöner Programmierstil ist, werde ich darauf nicht weiter eingehen!

C) Fehlerbehandlung in Exceptions:

Tritt bei der Abarbeitung eines PL/SQL-Blocks (in einer Prozedur, in einem Datenbank-Trigger, in einem Script) ein Fehler auf, wird das Programm abgebrochen und alle bis dahin nicht mit commit bestätigten Statements zurückgerollt! Das kann zu unerwünschten Effekten führen! Deshalb ist es sinnvoll, mögliche Fehlerquellen zu behandeln und reguläre Programmende zu definierten. Das geschieht über sogenannte Exceptions: Eine Exception ist eine Fehlerbedingung. Es gibt intern-definierte und user-definierte Exceptions.

Der Ablauf ist wie folgt:

Tritt ein Fehler auf, übergibt Oracle die Kontrolle der weiteren Ausführung an den Exceptionteil des PL/SQL-Blocks. Intere Exceptions werden automatisch ausgelöst.
unser-definierte Exceptions müssen durch ein raise-Statement ausgelöst werden.
Im Exception-Teil des PL/SQL-Blocks stehen die speziellen Statementfolgen, die sogenannten Exception-Handlers, die dann abgearbeitet werden.
Ist der Exception-Handler ausgeführt, geht die Kontrolle an den nächst-äußeren Block bzw. ans Betriebssystem zurück.
Existiert keine Exception-Handler für einen auftretenden Fehler, wird das Programm abgebrochen, alles zurückgerollt und die Kontrolle geht ans Betriebssystem zurück!

Beispiel:

```
DECLARE
        z number (5, 2);
BEGIN
        z := 7
        FOR i in - 5..5  LOOP
                z := z / i
        END LOOP;
EXCEPTION
     WHEN ZERO_DIVIDE Then
          z := 0;
     WHEN OTHERS Then
          insert into protokoll (string)
          values ('Unbekannter Fehler aufgetreten!);
END;
```

Anmerkungen:

1) Ohne den Exceptionteil würde das Programm bei i = 0 irregulär abbrechen.
2) ZERO_DIVIDE ist eine intern-definierte Exception und wird benutzt, um eine Nulldivision zu vermeiden.
3) OTHERS ist eine interne Exception, die alle Exceptions behandelt, die nicht explizit benannt sind!

ORCLE gibt folgende intern-definierte Exception vor:

- CURSOR_ALREADY_OPEN
- DUP_VAL_ON_INDEX
- INVALID_CURSOR
- INVALID_NUMBER
- LOGIN_DENIED
- NO_DATA_FOUND
- NOT_LOGGED_ON
- PROGRAM_ERROR
- STORAGE_ERROR
- TIMEOUT_ON_RESOURCE
- TOO_MANY_ROWS
- TRANSACTION_BACKED_OUT
- VALUE_ERROR
- ZERO_DIVIDE

Beispiel:

Aufgabe:

In die PL/SQL-Variable testprodukt soll die Bezeichnung des Produkts, die mit 'Z' anfängt eingetragen werden. Es ist nicht bekannt, wieviele Produkte es gibt, die mit 'Z' anfangen.

Also:

```
DECLARE
     testprodukt varchar2 (40);
BEGIN
```

```
select bezeichnung into testprodukt from PRODUKT
where bezeichnung like 'Z%';
```

Mögliche Exceptions:

i) es ist gar kein solches Produkt da!
   d. h. Exception NO_DATA_FOUND
ii) es gibt mehrere solche Produkte:
   d. h. Exception TOO_MANY_ROWS

also:

```
EXCEPTION
when NO_DATA_FOUND then
    ?
when TOO_MANY_ROWS then
    ?
when OTHERS then ?
```

Was soll passieren, wenn ein solchen Fehler auftritt?

Einfachster Fall: nichts soll passieren, also steht für '?': NULL. Das ist aber nicht besonders geschickt, da nicht zu erfahren ist, ob und weshalb das Programm nicht ausgeführt wurde!

Dehalb zweite Möglichkeit: Über eine Protokolltabelle protokoll: string varchar 2 (100) Informationen sammeln:

```
when NO_DATA_FOUND Then
    insert into protokoll values ('nix dagewesen');
when TOO_MANY_ROWS Then
    insert into protokoll values ('zuviel dagewesen')
when OTHERS Then
    insert into protokoll values ('Nicht bekannter Fehler');
END;
```

Oracle bietet mit der build-in function raise_application_error (parameterliste) die Möglichkeit, eigen definierte Fehlermeldungen auszugeben.

Syntax:

```
raise_application_error (error_number, string)
```

mit error_number: ein Wert zwischen -20000 und -20999;
string: ein beliebiger Text.

Damit ließe sich das gestellte Problem wie folgt lösen:

```
DECLARE
    testprodukt varchar 2 (40);
BEGIN
    select bezeichnung into testprodukt from produkt
    where bezeichnung like 'z%';
EXCEPTION
```

```
        when NO_DATA_FOUND Then
            raise_application_error (-20 101, 'War nix da');
        when TOO_MANY_ROWS Then
            raise_application_error (-20 102, 'War zuviel da') ;
        when others then
            raise_application_error (-20 999, 'Unbekannter Fehler');
    END;
```

User-definierte Exceptions

Oracle bietet die Möglichkeit, user-definierte Exceptions zu verwenden, d. h. die Menge der eben aufgelisteten System-internen Exceptions zu erweitern.
Die Deklaration und Verwendung user-definierter Exceptions geschieht wie folgt

- Exceptions werden im Deklarationsteil eines PL/SQL-Blocks wie Variablen deklariert (* mit dem Datentyp Exception);
- Ausgelöst wird die Exception durch ein raise-Statement im ausführbaren Teil des PL/SQL-Blocks
- Behandelt wird die Exception wie die systemdefinierten im Exception-Teil des PL/SQL-Blocks

Beispiel:

```
    DECLARE
       zu_lang    exception;
       c          varchar2 (40),
       name1      varchar2 (40);
       Vname1     varchar2 (40);
    BEGIN
       select name, vorname into name1, vname1 from kunde
       where name like 'Z%';
       c := substr (vname1, 1, 1) || '. ' || name1;
       IF length (c) > 40 THEN
            raise zu_lang;
       END IF;
       insert into test (string) values (c);
    EXCEPTION
       when NO_DATA_FOUND Then
            insert into test (string) values ('Nix da');
       when TOO_MANY_ROWS Then
            insert into test (string) values ('zu viele da');
        when zu_lang Then
            c := substr (vname1, 1, 1) || '. ' || substr (name1, 1, 37);
            insert into test (string) values (c);
       when others Then
            insert into test (string) values ('unbekannter Fehler');
    END;
```

Exception Handling arbeitet wie folgt:

- Treten Fehler auf, die nicht im Exception-Handler behandelt werden, wird das Programm abgebrochen
- Ist für eine aufgetretenen Fehler (oder ein durch Benutzer definierter 'Pseudofehler') ein Exception-Handler vorhanden, werden die Statements im Exception-Handler ausgeführt. Danach wird das Programm beendet!

Insbesondere wird nicht zu der Stelle zurückgesprungen, an der der Fehler aufgetreten ist! Auch ein Goto-Statement in einem Exception-Handler ist illegal! Einzige Möglichkeit, ein Programm weiterlaufen zu lassen, ist, die mögliche Exception in einem Subblock zu behandeln.

Beispiel:

```
DECLARE
    z number (5, 2);
BEGIN
    z := 4
    FOR i in -2..5 LOOP
    BEGIN
       z := z / i
    EXCEPTION
        when ZERO_DIVIDE then
        z := 1;
    END;
    insert into test 1 (zahl) values (z);
    END LOOP;
    insert into test 1 (zahl) values (-100);
EXCEPTION
    when others then
        insert into protokoll (string)
        values ('unbekannter Fehler' );
END;
```

Nach dem Auftreten der Exception wird aus dem inneren Block herausgesprungen und der nächst-äußere Block weiter bearbeitet.

Wichtig:

Mit Exception-Handling werden keine Fehler vermieden!
Es wird lediglich dafür gesorgt, daß ein Programm auf Fehler angemessen reagiert!

Cursor: Allgemeiner SQL 92/93 Standard

Wir hatten beim Exception-Handling bereits festgestellt, daß es eine Exception TOO_MANY_ROWS gibt und geben muß für den Fall, daß z. B. ein select-Statement mehr als einen Wert zurückbringt und in eine Variable einfügen will!

Manchmal will man aber gerade so eine Variable sukzessive mit den durch das select bestimmten Werten füllen und damit ein Problem bearbeiten.
Dazu braucht man also einen Mechanismus, der durch so eine Menge von Datensätzen läuft und sie Datensatz für Datensatz abarbeitet. Cursor beinhalten so einen Mechanismus:

Ein Cursor ist ein SQL-Objekt, das über eine geeignete Deklaration mit einem select-Statement verbunden ist.

Genauer:

Ein Cursor besteht im Wesentlichen aus einem Zeiger, der durch eine geordnete Menge von Zeilen laufen kann, auf jede der Zeilen zeigt. Man kann also damit die einzelnen Zeilen adressieren! Weiterhin gehört zu jedem Cursor ein select-Statement. Ist der Cursor offen, spezifiziert er eine gewisse Menge von Zeilen, eine gewisse Ordnung dieser Zeilen und eine Position bzgl. dieser Ordnung. Mögliche Positionen sind:

- auf einer spezifischen Zeile
- vor einer spezifischen Zeile
- nach der letzten Zeile

Allgemeine Syntax:

Declare <cursorname> [INSENSITIVE] [SCROLL] CURSOR
FOR <cursor-spezifikation>;

<cursor-spezifikation>: select-Statement [ORDER BY-Liste]
[FOR {READ ONLY | UPDATE [OF Spaltenliste]}]

Erläuterung:

i) Ist INSENSITIVE gesetzt, wird eine Kopie der im select-Statement aufgeführten Tabellen gemacht und der Cursor arbeitet nur mit der Kopie, d. h., Änderungen an der Tabelle können von anderen Usern gemacht werden! Dann sind updates und deletes über diesem Cursor nicht erlaubt!

ii) Ist scroll gesetzt, sind folgende FETCH-Operationen möglich:

FETCH NEXT FROM <cursorname> INTO <Zielkommaliste>
PRIOR
FIRST
LAST
ABSOLUTE n (n-te Zeile des Cursors)
RELATIVE n (n-te Zeile relativ zur augenblicklichen Position des Zeigers

iii) FOR READ ONLY: mit dem Cursor können keine updates oder deletes gemacht werden

iv) FOR UPDATE: mit dem cursor können updates bzw. deletes gemacht werden (siehe später)

Zulässige Cursor- Operationen sind:
:

OPEN

Syntax: OPEN <cursorname>;

Dieses Statement bewirkt, daß das dem Cursor assoziierte select-Statement ausgeführt wird; es wird eine Ordnung der Ergebnismenge definiert, der Cursor wird in den Zustand 'offen' versetzt und vor die erste Zeile der Ergebnismenge gesetzt.

FETCH

Syntax:

FETCH [[row selector] FROM] <cursor> INTO Ziel-Kommaliste;

mit:

Ziel-Kommaliste: Kommaliste von Variablen, in die die Werte der Zeilen des Cursors eingelesen werden.

CLOSE

Syntax: CLOSE <cursor>;

Das Statement versetzt den Cursor in den geschlossenen Zustand.

Beispiel:

```
DECLARE c_kunde scroll cursor for
                select name, vorname
                from kunde
                order by name
                for read only;
        vname kunde.name % TYPE
        vvorname kunde.vorname % TYPE
BEGIN
        Fetch absolute 3 From c. kunde into vname, vvorname;
END;
```

ORACLE-spezifische Implementation des Cursor-Konzepts:

Syntax:

```
DECLARE
        CURSOR <name> IS
        <select-Statement>;
```

Beispiel:

```
DECLARE
   v_ort adressen.ort %TYPE
   cursor C1 IS
      select ort from adressen
      where kundennr = 1
BEGIN
   OPEN C1
   LOOP
      FETCH C1 into v_ort;
      EXIT when C1%NOTFOUND;
      .... weitere SQL-Statements
   END LOOP;
   CLOSE C1;
EXCEPTION
   when NO_DATA_FOUND Then .... ;
END;
```

Mit Hilfe dieses Cursors werden jetzt alle Adressen des Kunden mit kdnr 1 durchgegangen und die Orte sukzessive in die Variablen v_ort geschrieben.
Der im EXIT-Statement des LOOPs verwendeten Ausdruck %NOT FOUND ist ein Attribut des Cursors.

%NOT FOUND hat den Wert FALSE, solange das letzte Fetch einen Datensatz zurückbringt.
%NOT FOUND hat den Wert TRUE, wenn das letzte Fetch keinen Datensatz mehr bringt (weil die Ergebnismenge abgearbeitet ist).
Insbesondere ist Fetch so konstruiert, daß es keinen Datensatz zurückbringen kann und daß das nicht als Exception angesehen werden wird!

Andere Cursor-Attribute:

% FOUND:	Logisches Gegenstück zu %NOT FOUND (C1 %NOT FOUND ≅ NOT C1% FOUND
%ROW COUNT:	Hat den Wert 0 beim Öffnen des Cursors und zählt für jede gefetchte Zeile um 1 hoch

Beispiel:

```
IF C1 %ROW COUNT > 100
    then exit;
END IF;
```

Weiteres Beispiel:

```
DECLARE
    bland     char (3);
    Küste     number (3);
    Süden     number (3);
    rest      number (3);

    cursor C1 IS
       select bundesland from adressen
            order by kundennr;
BEGIN
    Küste = 0; Süden := 0; rest := 0;
    OPEN C1
    LOOP
       FETCH C1 into bland;
       EXIT when C1 %NOT FOUND;
       IF bland in (' NS', 'HH', 'SH', 'MV') then
              Küste := Küste + 1;
          ELSIF bland in ('BW', 'BAY') then
             süden := süden + 1;
          ELSIF rest := rest + 1;
       END IF;
    END LOOP;
    insert into Statistik (Küstenland, Südenland, Restland)
    values (Küste, Süden, Rest);
    CLOSE C1
END;
```

Natürlich können auch mehrere Spalten in einem Cursor selektiert und über Fetch in Variable eingelesen werden:

```
DECLARE
    cname    kunde.name%TYPE;
    cvname   kunde.vorname%TYPE;
    cursor C1 IS
             select name, vorname from kunde;
BEGIN
    OPEN C1
    LOOP
       Fetch C1 into cname, cvname;
       EXIT when C1%NOT FOUND;
    END LOOP;
    Close C1;
END;
```

Über die Punkt-Notation können auch einzelne Spalten eines Cursors angesprochen werden:

```
Fetch C1.name into cname;
```

Natürlich kann das select-Statement der Cursor-Deklaration beliebig kompliziert sein:

```
DECLARE
    Küste, süden, rest, ...
    Cursor CB IS
            select distinct ort from adressen
            where kundennr in (select distinct kunde.kudennr
                               from kunde, auftrag, positionen
                               where kunde.kundennr = auftrag.kundennr
                               and auftrag.produktnr = positionen.produktnr
    order by ort;
BEGIN
.....
END;
```

Benutzung eines Cursor für updates oder deletes

Allgemeiner SQL-Standard:

Syntax:

```
Declare <cursorname> CURSOR FOR
        <select-statement>
        [FOR UPDATE [OF Spaltenkommaliste]]
```

Positioniertes Update:

Syntax:

```
update <tabelle>
set <zuweisungskommaliste>
where CURRENT OF <cursorname>;
```

Positioniertes Delete:

Syntax:

```
delete from <tabelle>
where CURRENT OF Cursor;
```

Oracle spezifische updates/deletes mit Cursor

Um einen Cursor zum positionierten update zu benutzen sind zwei Schritte notwendig:

1) Bei der Deklaration des Cursors muß am Ende die FOR UPDATE-Klausel stehen:

```
DECLARE
        Cursor C1 IS
          select produktnr, bezeichnung, preis from produkt
          where preis > (select avg (preis) from produkt)
          for update;
        vari C1 %ROW TYPE;
```

2) Im Loop kann nun Bezug genommen werden auf die Zeile, die gerade gefetcht wurde, um ein update vorzunehmen!

```
BEGIN
    OPEN C1;
    LOOP
          Fetch C1 into vari;
          update produkt set preis = preis - 01 * preis
          where current of C1;
    END LOOP;
END;
```

Wirkungsweise:
Durch OPEN wird das select-Statement des Cursors C1 ausgeführt. Die FOR-UPDATE-Klausel bewirkt, daß alle Zeilen der aktiven Tabelle gesperrt werden. Beim commiten der Transaktion werden die Locks wieder aufgehoben.
Wird beim select-Statement in der FOR UPDATE-Klausel eine Liste von Spalten angegeben, werden nur die Tabellenzeilen gesperrt, in denen diese Spalten vorkommen!

Weiteres Beispiel:

```
DECLARE
      Cursor Curdel IS
        select auftragsnr from positionen
        where exists (select * from auftrag
              where auftragsdat < '01-JAN-80
              and auftragsnr = positionen.auftragsnr)
        for update;
      vari curdel%ROWTYPE;
BEGIN
      OPEN Curdel;
```

```
        LOOP
            FETCH Curdel into vari;
            delete from positionen where current of curdel;
            EXIT when Curdel %NOT FOUND;
            END LOOP
END;
```

IV.1.2 Prozeduren und Funktionen

Der Standard SQL92 bzw. SQL3 enthält eine minimale Standardisierung:
Syntax:

```
PROCEDURE < name> [(Parameterkommaliste)]
[lokale Variablendeklaration ]
BEGIN
   Ausführbarer Teil
END;
```

bzw.

```
FUNCTION < name> [(Parameterkommaliste)] RETURNS <Datentyp>
[lokale Variablendeklaration]
BEGIN
   Ausführbarer Teil
   Return Variable;
END;
```

Prozeduren und Funktionen: (Oracle-Spezifisch!)
PL/SQL erlaubt die Definition und Verwendung von Prozeduren (und Funktionen). Diese können durch eine kleine Syntaxerweiterung als stored procedures in der DB abgelegt werden.

Syntax:

```
Procedure <proz.name> [(parameterkommaliste)] IS
     <lokale Variablen-Deklaration>
BEGIN
     ausführbarer Teil
EXCEPTION
     Exception-Handling
END [proz.name];
```

parameter ::= <variablenname> [IN | OUT | IN OUT] <Datentyp>

- beim Datentyp darf keine Größe angegeben werden!
- Das Schlüsselwort DECLARE wird bei der lokalen Deklaration **nicht** benutzt!

Prozeduren werden wie PL/SQL-Statements aufgerufen in einem PL/SQL-Block:

```
BEGIN
        <prozedur (parameterkommaliste)>
END;
```

Durch den Vorspann
create [or repleace] procedure .....
wird die Prozedur als stored procedure im Schema des DB-Benutzers abgelegt!

Beispiel:

Normale PL/SQL-Prozedur:

```
DECLARE
        procedure test (stringna IN varchar 2, stringvna IN varchar 2,
                                stringerg OUT varchar 2)
                IS
                n number;
            BEGIN
                n := length (stringna);
                IF n ≥ 37 THEN
                        stringerg := substr (stringvna, 1, 1) || '. ' || substr (stringna,
                                                                                1, n-3);
                ELSE stringerg := substr (stringvna, 1, 1) || '. ' || stringna;
                END IF;
            END;
    c varchar 2 (50);
BEGIN
    test ('Mueller', 'Heinz', c);
    insert into protokoll values (c);
END;
```

Als Stored-Procedure:

```
create or replace procedure test (stringna IN varchar 2,
                                stringvna IN varchar 2,
                                stringerg OUT varchar 2)
IS
n number;
BEGIN
            n := length (stringna);
            IF n ≥ 37 THEN
            stringerg := substr (stringvna, 1, 1) || '. ' || substr (stringna, 1, n-3);
            ELSE stringerg := substr (stringvna, 1, 1) || '. ' || stringna;
            END IF;
END;
```

Aufruf der stored procedure in einem PL/SQL-Block:

```
DECLARE
    c varchar 2 (50);
BEGIN
    test ('Mueller', 'Heinz', c);
```

```
    insert into protokoll values (c);
END;
```

Funktionen:

Unterschiede zu Prozeduren:

- Funktionen haben eine Return-Klausel
- Funktionen werden in einem Zuweisungsstatement aufgerufen!

Syntax:

```
[create or replace] function <name> [parameterliste)] Return <datentyp>
IS
....
```

Im ausführbaren Teil muß ein RETURN-Statement stehen!

Beispiel:

```
DECLARE
    function testf (stringna IN varchar 2, stringvna IN varchar 2)
        Return varchar 2 IS
        n number
        BEGIN
            n := length (stringna);
            IF n ≥ 37 THEN
            stringerg := substr (stringvna, 1, 1) || '. ' || substr (stringna,
                                                                    1, n-3);
            ELSE
            stringerg := substr (stringvna, 1, 1) || '. ' || stringna;
            END IF;
            Return (stringerg);
        END;
    c varchar 2 (50);
BEGIN
    c := testf ('Mueller', 'Heinz');
    insert into protokoll values (c);
END;
```

Als stored function:

```
Create or replace function testf (stringna IN varchar 2, stringvna IN varchar 2)
Return varchar 2 IS
n number
BEGIN
    n := length (stringna);
    IF n ≥ 37 THEN
       stringerg := substr (stringvna, 1, 1) || '. ' || substr (stringna, 1, n-3);
    ELSE
       stringerg := substr (stringvna, 1, 1) || '. ' || stringna;
    END IF;
    Return (stringerg);
END;
```

Return-Statement:
Das Return-Statement beendet sofort die Ausführung der Prozedur oder Funktion und kehrt zum aufrufenden Block zurück!

Bei Prozeduren:
Ein return-Statement darf keinen Ausdruck enthalten.

Beispiel:
```
IF x > 50 Then Return;
END IF;
```

Bei Funktionen:
Ein return-Statement muß einen Ausdruck enthalten!
Der Wert dieses Ausdrucks wird dem Funktionsidentifier zugeordnet!

Beispiel:
```
IF x > 50 Then Return (erg1);
ELSE return (erg 2);
END IF;
```

IV.1.3 Datenbank-Trigger

Datenbank-Trigger sind ereignisgesteuerte Programme. Ereignisse sind hier update, insert oder delete einer Tabelle. Die Aktionen können vor (before) oder nach (after) dem Ereignis gestartet werden!

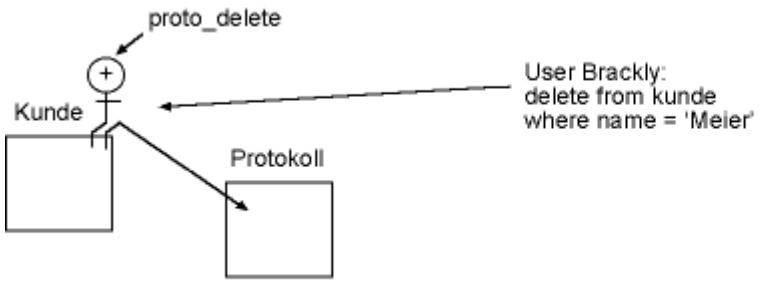
Datenbank-Trigger sind also immer bzgl. einer Tabelle definiert.
Für jede Tabelle gibt es bzgl. before und after jeweils genau einen Trigger für update, insert oder delete, d. h. pro Tabelle können maximal sechs Trigger existieren.

Beispiel:
Annahme: es existiert eine Tabelle Protokoll mit den Attributen Protnr, Tagesdatum, username, Tabellenname, Statement:

Ziel: Bei jedem delete auf die Tabelle KUNDE soll ein Eintrag in diese Tabelle Protokoll gemacht werden:

```
create trigger proto_delete
after delete on kunde
for each row
Begin
    insert into protokoll
    (select prot_seq.nextval, sysdate, user, 'KUNDE', 'Ein Kunde wurde
    gelöscht' from dual);
END;
```

Szenario:



Datenbank-Trigger sollen dazu dienen, globale Datenbankoperationen unabhängig vom einzelnen Benutzer auszuführen. Ein Beispiel für solche Aktionen sind Geschäftsregeln, die aus der praktischen Anwendung kommen und nicht über referenzielle Integritäten (Primär-Fremdschlüssel PK-FK) in der Datenbank selbst installiert werden können! Solche Geschäftsregeln sind zum Beispiel:

- Jede Person hat mindestens eine Adresse
- Jedes Stipendium darf höchstens drei Jahre laufen (mit Unterbrechungen)
- usw...

Datenbank-Trigger sollten nicht benutzt werden, wenn die gleiche Funktionalität schon mit anderen vom System bereitgestellten Mechanismen erledigt werden können. Es sollte also zum Beispiel keine PK-FK-Beziehung über die Datenbank-Trigger nachgebildet werden!

Vorsicht: Man muß aufpassen, daß keine rekursiven Trigger gebaut werden.!
Beispiel
Ein after update-Trigger wird auf der Tabelle Personen erzeugt, der im Triggerrumpf ein Update-Statement für eben die Tabelle Personen enthält.
Folge:
Das Update-Statement im Trigger veranlaßt den Trigger, wieder aktiv zu werden, also das Update-Statement auszuführen, das wieder den Trigger aktiviert, ... usw.

Um solche Fehlprogrammierungen zu vermeiden hat Oracle Sicherheiten eingebaut: im Allgemeinen kann man sagen, daß in einem Trigger auf der Tabelle T keine SQL-Statement (select, update) für diese T stehen darf!

Ein DB-Trigger wird immer dann neu kompiliert, wenn er zum erstenmal aktiviert wird oder im Laufe der Zeit aus dem Memory verschwinden muß (weil er lange nicht benutzt wurde). D. h. Trigger sollten keine zu langen Programme enthalten. Maximal 60 Zeilen ist ein guter Richtwert. Muß das auszuführende Programm mehr Zeilen enthalten, empfiehlt es sich, eine stored procedure zu schreiben (die kompiliert abgelegt wird) und diese vom Trigger aufrufen zu lassen.

Informationen über existierende DB-Trigger in der DB kann der User aus den Dictionary-views

- user_triggers
- all_triggers
- dba_triggers

entnehmen.

Syntax: (Oracle spezifisch!)

```
CREATE [or replace] TRIGGER <trigger_name>
BEFORE | AFTER
INSERT | DELETE | UPDATE ON <tabellenname>
[FOR EACH ROW]
[WHEN (<Ausdruck>)]
PL/SQL-Block
/
```

(Ereignisstatement: INSERT | DELETE | UPDATE ON <tabellenname>)

Erläuterungen:

i) Triggername: Kann irgendein Name sein, den der User noch nicht für einen anderen Trigger benutzt hat. Tabellen oder andere Datenbankobjekte mit gleichem Namen können parallel existieren!

ii) Before/after: Zeigen an, wann genau der Trigger zündet

iii) das Ereignis-Statement: Dieses spezifiziert

- welche der update-Operation die Zündung des Triggers veranlassen:
  die einzelnen Operationen insert, delete, update können jeweils einzeln oder durch 'OR' verknüpft gebündelt als Zündkriterien verwendet werden: z.B. after INSERT OR DELETE on Personen
- den Tabellennamen
  Es kann nur eine Tabelle angegeben werden. Insbesondere kein view!

Besonderheit: Beim Update-Kriterium kann eine Spaltenliste angegeben werden:
...
after update of straße, PLZ, Ort on adressen
...
Dies bewirkt, daß der Trigger nur dann zündet, wenn tatsächlich eine der angegebenen Spalten modifiziert wurde!

iv) Die Option FOR EACH ROW:

Diese Option steuert, wie oft der Trigger beim Eintritt des Ereignisses zündet.

Beispiel:

In der Tabelle Kunde gibt es 4 Meier:
Der Trigger proto_delete soll in der Tabelle Protokoll festhalten, wenn in der Tabelle Personen ein Datensatz gelöscht wurde durch den Eintrag: 'Eine Zeile wurde gelöscht'.

Sie setzten jetzt das Statement

```
delete from kunde where name = 'Meier';
```

ab.

Folge:
in die Tabelle protokoll wurde viermal der Eintrag 'Ein Kunde wurde gelöscht' gemacht, denn der Trigger mußte viermal zünden.

Schreibt man den Trigger proto_delete ohne das for each row:

```
create trigger proto_delete
after delete on kunde
Begin
insert into protokoll (select, ...);
END;
/
```

und setzt jetzt das Statement delete from Kunde where Name = 'Meier' ab, so steht in der Tabelle protokoll nur eine Zeile, da der Trigger nur einmal zu zünden brauchte, als das Statement zutraf!

v) Die when-Klausel

Für Trigger, die die for each row-Option haben (sogenannte Zeilentrigger), kann eine Einschränkung für's zünden definiert werden über die when-Klausel:
Diese Klausel darf nur ein SQL-Ausdruck sein. Subquerys und PL/SQL-Ausdrücke inclusive Prozeduren oder Funktionen sind nicht erlaubt.

Beispiel:

In der Tabelle Personen gibt es 10 Datensätze mit dem Namen Müller. 3 davon haben den Vornamen Claudia.
Auf Personen ist der folgende Trigger definiert:

```
create trigger mueller_weg
after delete on personen
for each row
when (old:vorname = 'Claudia' and old.name = 'Müller')
Begin
insert into protokoll values ('Ein Müller wurde gelöscht);
END;
/
```

Sie starten das Statement: delete from Personen where name = 'Müller';

Was passiert:

i) in der Tabelle Personen: alle datensätze mit Name Müller und Vorname Claudia sind gelöscht

ii) in der Tabelle Protokoll: 3 Einträge 'Ein Müller wurde gelöscht'!

Einschränkungen:

- Im Triggerrumpf dürfen keine DDL-Statements (create, alter, drop, grant, revoke) stehen und keine Transaktionsstatements (commit, rollback).
  Das gilt auch für Prozeduren/Funktionen, die aus Triggerrümpfen aufgerufen werden!
- Rekursive oder zyklische Trigger sind verboten

- Triggerfunktionen und definierte Constraints dürfen sich nicht widersprechen! (z. B. bei check-constraints, ...)
- Der Triggerrumpf sollte nicht mehr als 60 Zeilen Programmcode enthalten.
  Grund: Trigger werden nicht kompiliert gespeichert, d. h. bei jedem Zünden müssen sie neu kompiliert und geladen werden, sofern sie nicht noch im Memory des Servers sind!
  Besser: lange Codings als stored prozeduren schreiben und diese im Trigger aufrufen!
- Bzgl. der Tabelle, auf der der Trigger definiert ist, darf im Triggerrumpf kein SQL-Statement stehen (select, insert, update, delete).

Zum letzen Punkt:
Man möchte aber gerne z. B. mit einem watch-Trigger auch die gelöschten, geänderten oder neuen Werte in eine Protokolltabelle eintragen! Wie kommt man dann an diese Werte?

Der Oracle-Standard bietet folgende Optionen:

die Attributwerte der Tabelle des Triggers können, je nachdem ob es die alten oder die neuen Werte sind, gelesen werden über:
:old.<attr.name> bzw. :new.<attr.name>

Dies gilt nur für Zeilentrigger, d. h. solche Trigger mit der FOR EACH ROW-Option! Dann gilt:
- Beim insert-Ereignis hat der Trigger nur Zugriff auf die new-Werte! (old-Werte sind null!)
- Beim delete-Ereignis hat der Trigger nur Zugriff auf die old-Werte! (new-Werte sind null!)
- Beim update-Ereignis hat der Trigger Zugriff auf old- und new-Werte!
  Wichtig ist der **':'** vor dem old bzw. new Schlüsselwort!

Beispiel:

Bei jedem delete soll der watch-Trigger kdnr, name, vorname des gelöschten Kunden zusätzlich protokollieren

```
create or replace trigger watch
after delete on kunde
for each row
Begin
insert into protokoll
(select sysdate || ' ' || user || ' ' || :old.kdnr || ' ' || :old.name || ' ' || :old.vorname
from dual);
END;
```

Nächstes Problem:
Ursprünglich war watch ein Trigger, der bei jedem Ereignis (update, insert, delete) zünden sollte und entsprechende Einträge protokollieren sollte.
Das geht nicht mehr, wenn Werte der Tabelle protokolliert werden sollen, da im obigen Beispiel bei einem insert die old-Werte null sind!
Man braucht also Kontrollstatements, die unterscheiden können zwischen inserts, updates oder deletes:

Bedingte Prädikate im Triggerrumpf:

```
IF      inserting       then      statements...;      end if;
IF      updating        then      statements...;      end if;
IF      deleting        then      statements...;      end if;
```

in unserem Beispiel:

```
create or replace trigger watch
after insert or update or delete on kunde
for each row
Begin
   if deleting then
      insert into protokoll
      (select sysdate || ' ' || user || ' ' || :old.kdnr || ' ' || :old.name || ' ' ||
      old.vname from dual);
   end if;
   if inserting then
      insert into protokoll
      (select sysdate || ' ' || user || ' ' || :new.kdnr || ' ' || :new.name || ' ' ||
      new.vname
      from dual);
   end if;
   if updating then
      insert into protokoll
      (select sysdate || ' ' || user || 'alter Wert: ' || :old.kdnr || ' ' || :old.name || ' ' ||
      old.vname || 'neuer Wert: ' || :new.kdnr || ' ' || : new.name || ' ' ||
      :new.vname
      from dual);
   end if;
end;
```

Exception-Handling

Tritt ein Fehler auf bei der Ausführung des Triggerrumpfes, werden alle durch den Trigger bis dahin vorgenommenen Änderungen zurückgerollt; inklusive des Statements, das den Trigger gezündet hat!
Dies passiert nicht, wenn der auftretende Fehler durch einen Exception-Handler abgefangen wird!
Dieses Exception-Handling ist das im PL/SQL übliche (der Triggerrumpf ist ja auch nichts anderes als ein PL/SQL-Block!)

Rechte:

Um einen Trigger auf einer Tabelle erzeugen zu können, braucht man:

i) create trigger (oder create any trigger) Systemrecht

ii) Systemrecht auf der Tabelle:
entweder als Besitzer (Erzeuger) der Tabelle, oder das Alter table (alter any table)-Recht!

Infos:

Informationen über erzeugte Trigger in der DB stehen in den DD-Sichten dba_trigger, all_trigger, user_trigger

Ändern, Löschen, enable, disable von Triggern

- Ein Trigger kann (wie eine Prozedur/Funktion) nicht explizit geändert werden, sonder muß komplett neu erzeugt werden.
  Deshalb: create or replace trigger....
- Löschen einer Triggers mit
  drop trigger <name>;
- Abschalten einer triggers, ohne ihn zu löschen:
  Alter trigger <name> disable;
- Alter trigger <name> enable
  Macht den Trigger wieder scharf!
- Man kann auch alle Trigger enable'n oder disable'n, die auf einer Tabelle definiert sind, mit dem Statement:
  Alter table <tabellenname> DISABLE | ENABLE ALL Triggers;

Unterschiede zwischen Integritäts-Constraints, die in der DB abgelegt sind und Datenbank-Triggern, die Dateninput beschränken (z. B. Gehalt immer > 0, o. ä.):
Integritäts-Constraints gelten ab ihrer Erzeugung auch rückwirkend auf die Daten der Datenbanktabelle.
Datenbank-Trigger kontrollieren die Daten erst ab ihrer Erzeugung und lassen bereits existierende Daten außer acht!

Fazit

i) Trigger auf einer Tabelle T dürfen keine select, insert, update- oder delete-Statements auf dieser Tabelle T ausführen. Alle benötigten Werte aus Attribute von T müssen über die Variablen :new.<Attributname> bzw. :old.<Attributname> geholt werden.
Änderungen von Attributwerten von T gehen nur über Zuweisungen
:new.<Attributname> := Wert bzw. :old.<Attributname> := Wert

ii) Im Trigger-Body dürfen keine DDL-Statements stehen.

iii) Im Trigger-Body darf kein commit oder rollback stehen.

iv) PK-FK-Constraints und DB-Trigger können sich widersprechen.

v) Um einen Trigger auf einer Tabelle T erzeugen zu können, braucht man
a) create trigger Systemrecht
b) alter table-Recht auf T, falls man nicht Besitzer von T ist.

vi) Ein Triggerrumpf kann nicht explizit verändert werden, sondern der Trigger muß komplett ersetzt werden:
create or replace ...

vii) Ein Trigger wird gelöscht mit drop trigger <name>;

viii) Ein Trigger kann deaktiviert oder aktiviert werden ohne ihn zu löschen bzw. neu anzulegen:
alter trigger <name> disable | enable;

Nutzen von DB-Triggern:

a) Überprüfung von Geschäftsregeln, die nicht als Constraints formuliert werden können, durch das DBMS (nicht irgendeine Anwendung).
   Beispiel: Aktualisieren der Beträge in Auftrag und Positionen
b) Zentrale Kontroll- und Protokollirungsmechanismen, die unabhängig von irgendwelchen Anwendungsprogrammen laufen und aktiv werden, sobald auf die Tabelle zugegriffen wird.

Grenzen von DB-Triggern:

- DB-Trigger sind Ereignisabhängig und dann nur bzgl. der Ereignisse update, insert, delete!

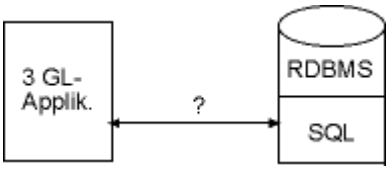
IV.2 Call Level Interface

Wie bereits erwähnt, hat SQL zwei Beschränkungen:
i) SQL hat keine prozeduralen Konstrukte
ii) SQL-Statements können nicht dynamisch in einer Applikation zur Laufzeit erzeugt werden.

Die erste Beschränkung wurde von einigen Herstellern gelöst durch die Definition proprietärer prozeduraler Erweiterungen (PL/SQL bei ORACLE). Das haben wir im vorigen Abschnitt behandelt.
Außerhalb solcher propr. Erweiterungen ist es bis jetzt nicht möglich, SQL-Statements in den prozeduralen Ablauf einer Anwendung einzubinden (egal ob statisch oder dynamisch). Dies war und ist aber eine wünschenswerte Option, um überhaupt Datenbankapplikationen entwickeln zu können, die nicht vom Endanwender umfangreiche SQL-Kenntnisse verlangen!
Historisch hat die erste Beschränkung bereits relativ früh zu Überlegungen geführt, SQL-Statements normalen 3GL's zur Verfügung zu stellen und damit ein prozedurales Umfeld für SQL zu schaffen.Wir behandeln hier den 1995 geschaffenen Standard einer bestimmten Methode des Zusammenspiels zwischen SQL und 3GL's, genannt Call Level Interface (CLI). Dieser Standard löst sowohl Beschränkung i) und ii), wie wir dann sehen werden. Anwenden werden wir diesen CLI-Standard mit der Sprache JAVA, realisiert durch JDBC.

Um das grundsätzliche Problem noch einmal zu verdeutlichen:
Relationale Datenbanken bieten als einziges Tor zur Außenwelt ihre SQL-Schnittstelle an! Einen anderen Zugang gibt es nicht.
Kennengelernt haben wir bisher die Kommunikation mit einem RDBMS über einen Kommandointerpreter, den wir mit SQL- bzw. prozeduralen SQL-Statements füttern konnten. (Direktes SQL)

Die Frage ist also: wie kann man Verbindungen schaffen zwischen einem 3GL-Programm und dem RDBMS?



IV.2.1 Allgemeine Standards und JDBC

Der Mechanismus von CLI ist kurz beschrieben:
Ein in einer üblichen 3GL-Sprache (C, C++, Java,...) geschriebenes Programm benutzt SQL-Operationen, indem es bestimmte vorher implementierte CLI-Routinen aufruft. Diese Routinen werden zuvor zum Programm gelinkt und haben die Funktion, das DBMS über dynamisches SQL mit der Durchführung der SQL-Operationen zu beauftragen. Dabei werden die SQL-statements als strings im 3GL-Programm behandelt und von den Routinen dem DBMS übergeben.
So gesehen bedeutet also CLI die Spracherweiterung üblicher Programmiersprachen um spezielle Funktionen, die das SQL-Handling regeln und damit eben das Interface zwischen einer 3GL und einem DBMS bilden!
Insbesondere bedeutet die Verwendung von Dynamischem SQL, daß die zur Ausführung gedachten SQL-Statements erst zur Laufzeit definiert werden müssen!
Es sind auch keine zusätzlichen Compiler notwendig, der Standard 3GL-Compiler reicht aus.

Weiterhin können mit CLI Programme geschrieben werden, die unabhängig vom zugrunde liegenden DBMS (und damit dem SQL-Sprachumfang) sind! CLI enthält einige Routinen, die dieses Problem handhaben!

CLI ist ein Standard zur Einbindung von Routinen in eine 3GL; deshalb muß auch die Syntax von CLI von 3GL zu 3GL variieren!

Wir werden CLI im Zusammenhang mit der Sprache JAVA behandeln, Beispielcode wird also immer für das folgende JAVA sein!

Ein kleines Beispiel zur Einführung:

```
String stmt = "select * from Auftrag where anr = 5001";
Statement s = conn.createStatement ();
ResultSet rs = s.executeQuery (stmt);
```

- Mit der ersten Zeile wird eine Variable stmt vom Typ String deklariert und dieser ein select-Statement als Stringwert zugewiesen.
  Mit der 2. Zeile wird für die Session, dargestellt durch eine Connection-Instanz mit dem Namen conn, mittels der Methode createStatement ein Statement als Instanz der Klasse Statement und mit Namen s erzeugt.
  Diese Instanz s kann nun über ihre Methode executeQuery (stmt) das als String übergebende SQL-Statement zur Ausführung bringen, d. h. zur weiteren

Bearbeitung zum DBMS parsen.
Rückgabewert der Methode executeQuery () ist eine Variable vom Datentyp ResultSet, d. h. eine Variable, die die potentiell mehreren Datensätze, die das select-Statement liefert, aufnehmen kann. (ein Cursor!)

- Das Programmfragment führt also aus JAVA heraus ein (statisches) SQL-Statement aus. Natürlich hätte man über eine Eingabeerfragung der Variablen stmt auch zur Laufzeit dynamisch ein SQL-Statement zuweisen können!
- Die in der ResultSet-Variablen rs aufgelaufenen Datensätze können nun innerhalb JAVA weiterverarbeitet werden. Dazu später mehr!

Zunächst grundsätzliche Erläuterungen des allgemeinen CLI-Konzepts: Jede CLI-Routine wird in der zu betrachtenden 3GL implementiert als Funktion oder Prozedur (oder Methode im objektorientierten Fall!). Die Prozeduren bzw. Methoden werden aufgerufen, deshalb Call level Interface!

CLI definiert also eine standardisierte Meta-Sprache, bestehend aus Konstrukten, die dann konkret als Spracherweiterungen in einer 3GL umgesetzt werden!

CLI-Routinen können Parameter übergeben. Jeder Parameter hat einen Namen, einen Datentyp und einen Modus (IN, OUT).

Parameter Datentypen sind:

CLI	C	JAVA
Integer	Long	long
smallint	Short	short
character (n)	Char der Länge n	string
any	Float, double	float, double, ...

In der Standardisierung SQL2 (1992) werden 47 CLI-Routinen definiert.
Diese 47 definierten Routinen mit ihren Metanamen behandeln alle vom Standard erfaßten Möglichkeiten der Behandlung von SQL-Code in einer 3GL. Da diese Metanamen im konkreten Fall (JAVA) durch andere ersetzt werden, will ich sie nicht benennen und ihre Funktion beschreiben. Den interessierten Leser verweise ich auf das Buch von Date/Darwen. Statt dessen werden wir die wesentlichen Funktionalitäten von CLI und ihre Umsetzung in JAVA besprechen!

Wesentlich für CLI-Anwendungen sind:

- Aufbau einer Datenbank-Verbindung (connection)
- Preparieren und Ausführen von SQL-Statements

Verbindungsaufbau:

Zum Aufbau einer Verbindung eines 3GL-Programms mit einer (beliebigen) Datenbank wird ein sogenannter Treiber benötigt, der genau das behandelt! Zur Erledigung seiner Aufgabe braucht der Treiber eine Reihe von Informationen: welches Netzprotokoll, welcher Port, welcher Datenbank-Aliasname, usw.
Auf diese technischen Details können wir jetzt noch nicht eingehen. Betrachten wir einen CLI-Treiber einfach als ein Stück Software, das implementiert werden muß (auf dem Client, im Netz oder auf dem Server), das dem eigentlichen Programm bekannt gegeben und aufgerufen (geladen) werden muß. Treiber für das Java-CLI und

gängige (relationale) Datenbanksysteme kann man aus dem Netz runterladen (weblogic.com, intersolv.com, ...)

Preparieren und Ausführen von SQL-Statements:

Es gibt prinzipiell drei Möglichkeiten, ein SQL-Statement abarbeiten zu lassen:
a) Direkte Ausführung (CLI-Routine execDirect())
b) Preparieren und später ausführen (CLI-Routinen prepare() und execute())
c) Aufruf einer stored procedure (CLI-Routine call()...)

zu a): Die Benutzung der CLI-Routine execDirect (stmt) bedeutet, daß das SQL-Statement unmittelbar zum DBMS geparsed wird. Dieses erstellt einen optimalen Ausführungsplan für das Statement (wie es das immer tut, wenn ein User ein SQL-Statement absetzt, egal woher!) und führt es aus. Dann wird die Ergebnismenge (im Falle eines selects) zurückgegeben.

Diese direkte SQL-Ausführung ist im JDBC realisiert durch die Methoden executeQuery() bzw. executeUpdate() der Klasse Statement.
Nachteilig wirkt sich die Benutzung dieser Routine aus, wenn ein SQL-Statement z. B. innerhalb einer Schleife mehrfach ausgeführt wird:

```
for (int i = 0; i < max; i++) {
executeUpdate(SQL-Statement);
...
}
```

Bei jedem Schleifendurchgang wird das SQL-Statement zum DBMS geparsed, dort ein Plan erstellt, und dann ausgeführt. Aber eigentlich bräuchte man den Plan nur einmal zu erstellen, da die Struktur des SQL-Statements immer die gleiche bleibt; lediglich gewisse Parameterwerte werden sich eventuell ändern!

Aus diesem Grunde benutzt man präparierte Statements, die also im vorhinein zum DBMS geschickt, dort vorkompiliert werden (Ausführungsplan) und später durch konkreten Aufruf ausgeführt werden. Es gibt zwei Arten solcher präparierter SQL-Statements: die Fälle b) und c)

zu b): Fall b) ist zweigeteilt:
zunächst wird durch den Aufruf der CLI-Funktion prepare() und der Eingabe eines SQL-Statements dieses ans DBMS geparsed, dort vorkompiliert und in einem bestimmten Bereich des Hauptspeichers zwischengelagert. Durch einen späteren Aufruf der CLI-Funktion execute() wird das vorkompilierte Statement dann ausgeführt.
Dieser Standard ist im JAVA-API JDBC wie folgt adaptiert:
Mit der Methode prepareStatement(String) der Klasse Connection wird eine Instanz der Klasse PreparedStatement erzeugt. Gleichzeitig wird das im String übergebene SQL-Statement ans DBMS geschickt und dort vorkompiliert abgelegt.
Mit den Methoden executeQuery() bzw. executeUpdate() dieser Klasse kann dann das Statement ausgeführt werden. executeQuery() liefert ein ResultSet zurück, executeUpdate() die Anzahl der aktualisierten Datensätze!

zu c): Die dritte Möglichkeit, ein SQL-statement in einer 3GL über CLI auszuführen,

besteht im Aufruf einer stored procedure. Einige Relationale DB's bieten die Möglichkeit, SQL-Code oder prozedural erweiterten SQL-Code in der Datenbank in kompilierter Form (also mit erstelltem Ausführungplan) abzulegen!
In diesem Fall muß also nicht über die CLI-Routine prepare() der Code ans DBMS geschickt werden, sondern die stored procedure kann direkt aufgerufen werden.
In JAVA JDBC geschieht das wie folgt.

Voraussetzung: in der Datenbank ist die stored procedure *select Kunde* abgelegt

Mit der Methode prepareCall("{call selectKunde}") der Klasse Connection wird eine Instanz der Klasse CallableStatement erzeugt. Gleichzeitig wird die stored procedure aufgerufen, d. h. in den Hauptspeicher geladen. CallableStatement ist eine Spezialisierung der Klasse PreparedStatement, d. h. deren Methoden können benutzt werden, um den in der stored procedure kompilierten SQL-Code auszuführen, z. B. mit executeQuery().

Eine Besonderheit im CLI-Standard ist noch die Nutzung von Platzhaltern zur Übergabe von Parameterwerten zwischen dem SQL-Code und dem 3GL-Code. Dabei werden also Parameter des SQL-Code an 3GL-Variable gebunden. Platzhalter werden angezeigt durch '?'.
Dieser Mechanismus der Parameterübergabe ist nur definiert für die Fälle b) und c), d. h. bei direkt ausgeführten SQL-Statements können keine Parameter übergeben werden.
Im JAVA-API JDBC ist dies wie folgt gelöst:
Die Klasse PreparedStatement hat Methoden setInt(), set Float(), usw. Diese Methoden binden die Eingabeparameter des SQL-Statements zu Variablen des JAVA-Programms.

Beispiel:

```
PreparedStatement ps = conn.prepareStatement("select name from
Kunde where kdnr = ? and geschlecht = ?);

ps.setInt(1, 37);
ps.setstring(2, "W");
ResultSet rs = ps.executeQuery();
```

Im Beispiel werden mit den Methoden setInt(1, 37) dem ersten Platzhalter der Wert 37 und mit setString(2,"W") dem zweiten Platzhalter der Wert "W" übergeben. Natürlich können hier auch Variable übergeben werden!

Die drei Fälle noch mal in einem konkreten JDBC-Beispiel:

```
// Erzeugung einer connection-Instanz (Verbindung zur Datenbank):
   Connection con = DriverManager.getConnetion("Treiber-Spezifikation");

// Fall a): direkte SQL-Ausführung:
   String stmt = "update kunde set name = 'Meier' where kdnr = 7;
   Statement s1 = con.createStatement ();
   int x = s1.executeUpdate(stmt);
```

```
// x liefert die Anzahl der aktualisierten Datensätze zurück!
   s1.close;

// s1.close schließt das Statement und gibt Ressourcen frei!

// Fall b): ein vorkompiliertes Statement (das gleiche wie in a) )
   PreparedStatement ps = con.prepareStatement (stmt);
   int y = ps.executeUpdate ();
   ps.close
```

zum Fall c):

Voraussetzung: es existiert eine stored procedure:

```
create stored procedure selectKunden (ikdnr IN OUT number,
                                      oname OUT varchar 2,
                                      ovorname OUT varchar 2)
IS
Begin
select kdnr, name, vorname into ikdnr, oname,ovorname from kunde
where kdnr = ikdnr;
END;
```

Diese stored precedure hat also 1 Eingabeparameter: ikdnr und 3 Ausgabeparameter ikdnr, name, vorname.
Die Eingabeparameter müssen mit den Methoden setxxx () an JAVA-Variable oder Konstrukten gebunden werden, die Ausgabeparametertypen der stored procedure müssen über die Methode registerOutParameter() der Klasse CallableStatement als JAVA-Typen angemeldet werden!

also:

```
// Fall c):Aufruf der stored procedure select kunde (....):
      CallableStatement cs = con.prepareCall("{call selectKunden(?, ?, ?)}"),
      cs.setInt (1, 37);
      cs.registerOutParameter (2, java.sql.Types.VARCHAR);
      cs.registerOutParameter (3, java.sql.Types.VARCHAR);    *
      ResultSet rs = cs.executeQuery();
      string jerg;
      while (rs.next ()) {
            jerg = getInt(1) + "  " + getString(2) + "  " + getString(3);   **
      System.out.println (jerg);
      }
```

* Im Package java.sql existiert eine Klasse Types, die verschiedene Datentypen anbietet.

** Mit den Funktionen getInt(), getString(), wird den Spalten des ResultSet jeweils ein Java-Datentyp zugewiesen.

IV.2.2 Realisierte Call Level Interfaces

A) DB-Herstellerabhängig:
Dies sind eigens entwickelte API's als Spracherweiterung z. B. für C bzw. C++. Die verwendeten Funktionen müssen über bereitgestellte include-files bzw. Lib-Dateien in die C-Entwicklungsumgebung eingebunden werden. Dann können sie als 'normale' Funktionen im Applikations-C-Code verwendet werden.
Der DB-Hersteller ORACLE bietet hier ein API mit dem Namen OCI an (Oracle Call Interface). Dieses gibt es für alle Plattformen, für die es auch RDBMS von Oracle gibt! Also z. B. AIX, Sun, Windows 95,....
Dieses API enthält eine Menge von Funktionen, von speziellen Loginfunktionen (ologin, ...) über Funktionen, die die Ausführung von SQL-Statements veranlassen, usw.

Solche CLI-API's haben allgemein den Vorteil, daß sie eng mit dem jeweiligen RDBMS des Herstellers zusammenarbeiten (native), d. h. insbesondere auf alle Besonderheiten dieses RDBMS eingehen können (Performance, Sprachumfang, ...). Nachteilig ist natürlich, daß mit diesen Schnittstellen ausschließlich auf diese bestimmte RDBMS zugegriffen werden kann. Solche API's sind also unnütz, wenn die gleiche Applikation auf mehrere Datenbanken verschiedene Hersteller zugreifen muß!

Inzwischen bieten eigentlich alle Hersteller von RDBMS proprietäre CLI-API's an.

B) Herstellerunabhängige API's:
Das in der PC-Welt am meisten verbreitete CLI-API, um PC's zu Datenbank-Clients zu machen und Datenbankzugriffe aus PC-Applikationen zu ermöglichen, ist der von Microsoft geschaffene Quasi-Standard ODBC (als Spracherweiterung von C!).

ODBC steht für Open Data Base Connectivity
Die ODBC-Architektur besteht aus (mindestens) vier Komponenten:

- Applikation: ruft ODBC-Funktionen, um SQL-Statements abzuschicken und Ergebnisse anzunehmen
- Treiber Manager: lädt die benötigten Treiber, die von der Applikation eingesetzt werden müssen
- Treiber führen ODBC-Funktionsaufrufe aus, schicken die SQL-Anforderungen zu einer bestimmten Datenquelle und behandeln die Rückgabe von Ergebnissen
- Datenquellen: Bestehen aus den Informationen über das beteiligte Datenbank-System, Netz-Protokolle, Betriebssystemen und dem Datenbankschema, das die Daten enthält

In die Applikationen können dynamisch verschiedene Datenbank-Treiber geladen werden. Datenbank-Treiber sind dynamic link libraries DLL's, die vom DBMS-Hersteller oder anderen Software Herstellern bezogen werden können. Auch Oracle bietet z. B. einen 32-Bit ODBC-Treiber an. ODBC besteht im wesentlichen aus den folgenden Standardisierungen:

- Ein C-API zur Kommunikation mit jeder Datenbank, zu der ein Treiber existiert. Im API enthalten sind standardisierte Methoden für connect und eine Standardmenge von Error-Codes.
- ODBC definiert ein Minimum an Funktionalitäten, die von den meisten DBMS angeboten werden und von allen Treibern unterstützt werden. (Betrifft vor allem Arbeiten mit Metadaten der Datenbank über's Data Dictionary, usw...)
- ODBC benutzt einen bestimmten SQL-Sprachumfang entsprechend der Norm SQL2 (1992).
- ODBC stellt bestimmte Funktionen zur Verfügung um überprüfen zu können, welche Funktionalitäten das im Zugriff befindliche DBMS anbietet. Dazu werden drei Conformance-Level definiert, Core, Level 1, Level 2, die bestimmte Mengen von Funktionalitäten für's API und für SQL enthalten. Informationen erhält man durch die Funktionen SQLGetInfo, SQLGetFuncitions, ..

Für's Management einer ODBC-Umgebung mit verschiedenen Treibern und Datenquellen existiert ein ODBC-Administrator (ODBCAD 32.exe). Diesen kann man benutzen, um verschiedene Treiber zu registrieren und Treibern Datenquellen zuzuordnen.

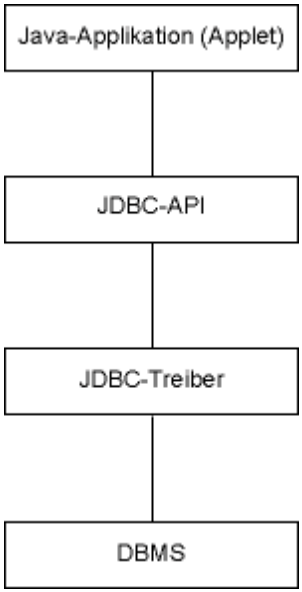
Auf ein Beispiel für eine ODBC-Applikation verzichte ich und gehe jetzt direkt auf einen neuere Entwicklung ein, JDBC.

C) JDBC steht für Java Database Connectivity

JDBC funktioniert sehr ähnlich wie ODBC. Der gravierendste Unterschied ist natürlich, daß JDBC objektorientiert ist, während ODBC rein prozedural orientiert ist. Insbesondere arbeitet JDBC nicht mit Pointern und Handlern, sondern mit Klassen, Objekten, Methoden. Trotzdem kann man natürlich auch JDBC für rein prozedurale Programmabläufe nutzen!

JDBC ermöglicht einer Applikation einen Datenzugriff auf beliebige DBMS über einen JDBC-Treiber, der zur Laufzeit von der Applikation geladen wird. Der (die) Treiber ist in Java geschrieben und damit auf allen Plattformen einsatzfähig. Wie bei ODBC gibt es allgemeine Treiber für beliebige DBMS und JDBC-Treiber, die von Datenbank-Herstellern speziell für ihre DBMS geschrieben wurden.

Kommunikationsschichten für JDBC:



Die Java-Applikation benutzt das JDBC-API, um SQL-Statements in der Applikation behandeln zu können. Das JDBC-API ist unabhängig von der zugrunde liegenden Datenbank!

Der JDBC-Treiber stellt die Verbindung zwischen dem JDBC-API und der Datenbank her. Meist kommt der JDBC-Treiber vom Datenbank-Hersteller. Der JDBC-Treiber definiert eine Menge von Schnittstellen, die von einem Datenbank-Hersteller implementiert sein müssen (Datenbank-seitig!)

Man unterscheidet vier JDBC-Treibertypen:

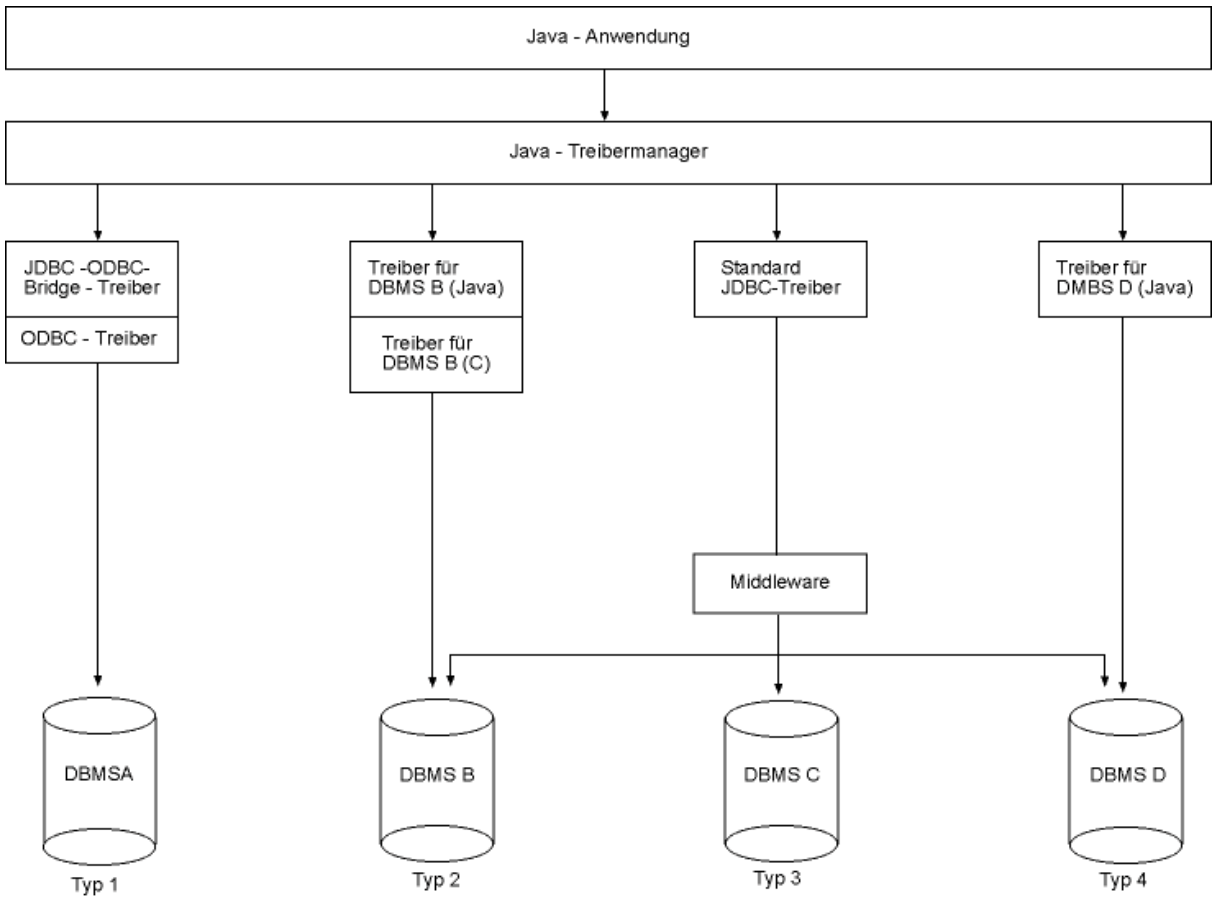
Typ 1: JDBC-ODBC-Bridge
Dieser Treiber greift auf einen bereits vorhandenen ODBC-Treiber und dessen Funktionalitäten zu. Damit kann ein Java Programm auf sämtliche ODBC-Datenbanken zugreifen. Nachteilig ist, daß der Typ 1-Treiber nicht aus reinem Java-Bytecode, sondern auch zum gutem Teil aus dem ODBC-Treiber, also C-Code besteht. Damit ist er plattformabhängig.

Typ 2: Dieser Treiber verlangt einen bereits vorhandenen proprietären Treiber für das jeweilige DBMS und setzt eine JDBC-Schicht darüber. Ein Beispiel ist der JDBC-OCI-Treiber von ORACLE, der auf dem speziell für das ORACLE RDBMS entwickelten CLI-API OCI aufsetzt.

Typ 3: Dieser Treiber ist ein 100%iger Java-Treiber, also plattform-unabhängig. Er kommuniziert nicht direkt mit dem DBMS, sondern über eine sogenannte Middleware.
Diese Middleware unterhält sich nach oben mit dem JDBC-Treiber und unterstützt nach unten verschiedene DBMS. Sie startet einen Datenbankagenten, der dann Kontakt zum DBMS aufnimmt. Dieses Modell, an dem neben Client und Server noch eine dritte Instanz, eben die Middleware z. B. auf einem Netzserver beteiligt ist, nennt man 3-teer-Modell.

Typ 4: Ist auch ein reiner Java-Treiber, der von einem DBMS-Hersteller ausschließlich für sein System entwickelt wurde, also proprietär. ORACLE bietet einen solchen Treiber mit dem Namen JDBC THIN an!

Konfigurationen:



Die Funktion der JDBC-Treiber ist die gleiche, die auch die ODBC-Treiber erfüllen müssen:
parsen der SQL-Anweisungen aus DBMS, Erzeugen und unterhalten einer Verbindung zum DBMS, Handling des Datenaustausches, Aufruf spezieller JDBC-Funktionen!

JDBC-Treiber verschiedenster Art kann man über das Internet z. B. von javasoft.com, weblogik.com, interlink.com, oracle.com, etc. runterladen.

Bevor wir jetzt konkrete Anwendungen von JDBC besprechen, zuerst eine Einführung in die Sprache Java:

Einschub: Java Crashcourse

Java ist eine objektorientierte Programmiersprache. D. h. sie ist selbst in Klassen, Objekte, Methoden, Attribute organisiert, die Programmierer benutzen können, um ihre Programmfunktionalität davon abzuleiten. Die Sprachkonstrukte (Operationen, Datentypen, etc.) sind sehr C-ähnlich, es gibt natürlich auch Unterschiede. Die Gesamtheit aller Sprachelemente ist in verschiedenen sogenannten packages zusammengefaßt: so gibt es z. B. ein Package java.io.∗, das verschiedene Klassen mit diversen Methoden für die Ein-/Ausgabe bereitstellt. Ein anderes package ist z.

B. java.math.*, das mathematische Funktionen etc. zur Verfügung stellt. Insgesamt stellt das Java jdk 1.1.5 ca. 23 packages zur Verfügung, die jeweils verschiedenste Funktionen abdecken.
Zentrales package ist java.lang. Es enthält die Definitionen der grundlegenden Java-Sprachkonstrukte wie Datentypen, usw.
Jedes Java Programm benutzt zumindest dieses package.
Sollen noch andere Funktionalitäten aus anderen packages benutzt werden, müssen diese packages dazugeladen werden. Dies geschieht durch ein import-Statement zu Begin des Programms:
z. B. import java.sql.*

java.sql.* ist das package, das die Spracherweiterung um die CLI-Funktionen realisiert, um aus einem Java-Programm heraus mit einer realationalen Datenbank zu kommunizieren. Dieses package werden wir gleich ausführlich besprechen.

Man kann man zwei Arten von Java-Programmen schreiben:
'normale' Programme (Applikationen), die mit dem java-Interpreter gestartet werden, und sogenannte Applets, die in HTML Seiten eingebunden und von Browsern gestartet werden.

Ein Java-Programm besteht aus einer oder mehreren Klassen, die jeweils in einen eigenen .class-file übersetzt wurden mit dem Java-Compiler javac. Eine dieser Klassen muß eine spezielle Methode main () definieren, den ausführbaren Teil des Programms (wie in C):

```
public static void main (string args[]) {....}
```

Grobe Sprachübersicht:
Die wichtigsten Packages, die den Sprachumfang von Java definieren, sind:

java.applet	enthält die Klasse Applet als Superklasse aller Applets und drei Interfaces
java.awt	(Abstract Window Tookit), enthält drei Kategorien von Klassen: für Graphik, für GuI-Objekte, für Layouts. java.awt hat einige Unter-Packages, die jetzt noch keine Rolle spielen. (event, ...)
java.io	enthält sehr viele Klassen, die mit den verschiedensten I/O-Problemen zu tun haben.
java.lang	enthält die zentralen Sprachkonstrukte
java.math	enthält Klassen zur floating-point-Arithmetik, usw.
java.net	enthält Klassen für Netz-Programmierung
java.text	enthält Klassen für NLS (z. B. Daten, Zahlen, ...)
java.util	enthält nützliche Konstrukte als Klassen (z. B. Vektor)
java.sql	enthält das JDBC-Package

Arbeiten mit JDBC

Das JDBC-Package java.sql als Spracherweiterung-API auf der Basis von CLI enthält die folgenden Klassen:

- Callable Statement: behandelt den Aufruf von stored procedures;
- Connection: die JDBC-Darstellung einer DB-Session; hier können mit speziellen Methoden Statements erzeugt werden
- Database MetaData: ermöglicht über Methoden die Abfrage von Informationen über die aktuell verbundene DB
- Date: Unterklasse von java.util. Date mit speziellen Datumskonstruktoren
- Driver: stellt eine Herstellerspezifische JDBC-Implementierung dar, die über die Methode connect() eine Session zu einer DB öffnen kann
- Driver Manager: hält eine Liste aller registrierten (implementierten) JDBC-Treiber und ermöglicht über Methoden den Aufbau einer Session
- Driver Property Info: hält Informationen, die ein Treiber benötigt
- Prepared Statement: behandelt vorcompilierte Statements
- Result Set: behandelt die Ergebnismengen von select-Statements (wie ein Cursor!)
- Result Set Meta Data: enthält Informationen über Typen und Eigenschaften der Spalten einer Result Set-Instanz (mittels Methoden abrufbar)
- Statement: behandelt direkt auszuführende SQL-Statements
- Time: behandelt den SQL-Time-Datentyp
- Time Stamp: Darstellung der Java-Date-Klasse als Timestamp
- Types: enthält die SQL-Datentypen als Attribute!

Die einzelnen Attribute, Konstrukte, Variablen, Methoden der einzelnen Klassen können aus der Dokumentation des jdk entnommen werden.

Zu JDBC gibt es inzwischen eine ganze Reihe von Büchern; ich empfehle jdem, sich das für ihn geeignete Buch herauszusuchen! Bevor es in die Praxis geht, noch einige Anmerkungen zum Laden und Einbinden von JDBC-Treibern in Java-Programme.

- Prinzipiell gibt es mehrere Möglichkeiten, JDBC-Treiber zu laden und eine Connection-Instanz zu erzeugen.
  Die eleganteste und (aus Sicht des Programms) sicherlich flexibelste ist die Benutzung eines bestimmten Java-Mechanismus um dynamisch Klassen zu laden: Java bietet in der abstrakten Klasse Class die Methode forName() an. Mit dieser Methode werden Klassen, die als Parameterstring übergeben werden, geladen.

Da auch JDBC-Treiber nicht anders als in einer Klasse realisiert sind, können auf diese Weise die Treiber geladen werden:

Class.forName ("<vollständiger Name der Treiberklasse>");
für ORACLE:
Class.forName ("oracle.jdbc.driver.OracleDriver");

Durch Aufruf dieser Methode forName wird also der spezifizierte Treiber geladen, d. h. eine Instanz seiner Klasse erzeugt. Mit diesem Prozeß können natürlich pro Java-Programm mehrere JDBC-Treiber (etwa für verschiedene Datenbanken) geladen und später über verschiedene Connections bzgl. dieser Treiber Zugriffe auf verschiedene Datenbanken gemacht werden.

- Ist ein Treiber geladen (z. B. über Class.forName(...)) kann mittels der Klasse DriverManager bzgl. dieses Treibers eine Verbindung zur Datenbank hergestellt werden in Form einer Connection-Instanz. Dazu wird die Methode getConnenction() der Klasse DriverMangager benutzt.

Wichtig:

Eine Connection-Instanz ist eine Datenbank-Treiberaktion, bestehend aus einem oder mehreren SQL-Statements.
Die Methode getConnection() hat als Parameter den URL (Uniform Ressource Locator) der geladenen Treiber-Instanz als String, gefolgt eventuell von User und Password.
Dieser URL hat bzgl. JDBC die Syntax:

jdbc: <subprotocol> : <Subname>

Für den speziellen OCI-JDBC-Treiber von ORACLE:

jdbc: oracle: oci7: @<Oracle SID>

Ein Beispiel-Verbindungsaufbau wäre also (nachdem eine Treiber-Instanz erzeugt wurde):

```
Connection con = Driver Mangager.getConnection("jdbc:oracle:oci7:
                                                @db_i" kapv, kapv);
```

Natürlich können User und Password auch eingelesen und als Variable übergeben werden!

Eine JDBC-Verbindung in einem Java-Programm wird also in zwei Schritten erzeugt:

1) Erzeugen einer Treiber-Instanz
2) Erzeugen einer Connection-Instanz bzgl. dieser Treiberinstanz

Wichtig:

Wie eben erwähnt, ist eine Connection-Instanz für die Datenbank eine Transaktion, bestehend aus mehreren SQL-Statements. Die Klassen Connection, Statement und Result Set haben close()-Methoden und es ist guter Programmstil, Objekte zu schließen, wenn sie nicht mehr gebraucht werden, um damit Ressourcen freizugeben. Defaultmäßíg sind alle Connection-Objekte im Autocommit-Mode, d. h. jedes Statement wird committed, sobald es ausgeführt wird. Das macht aber nicht immer Sinn und dieser Modus kann mit der Methode setAutoCommit() ausgeschaltet werden. Dann muß von Hand mit den Methoden commit() bzw. rollback() der Klasse Connection gearbeitet werden.
Wird eine Connection geschlossen, die nicht im AutoCommit-Modus ist, sind alle nicht explizit committeten SQL-Statements verloren und werden vom DBMS zurückgerollt!

Zum Abschluß der source-code eines Beispielprogramms zur Kommunikation eines Javaprogramms mit den KAPV-Tabellen in einer Oracle-Datenbank

```
// Klasse Beispiel_JDBC

package java.lehre.dbprog;

import java.sql.*;
import java.io.*;


class Beispiel_JDBC


{

// Funktion, die eine Zeile des Standard-Inputs einliesst
  static String readEntry (String prompt)
  {
    try
    {
      StringBuffer buffer = new StringBuffer ();
      System.out.print (prompt);
      System.out.flush ();
      int c = System.in.read ();
      while (c != '\n' && c != -1)
      {
        buffer.append ((char)c);
        c = System.in.read ();
      }
      return buffer.toString ().trim ();
    }
    catch (IOException e)
    {
      return "";
    }
  }

  public static void main (String args [])
       throws SQLException, ClassNotFoundException, IOException
  {
// Laden des Oracle JDBC driver
    Class.forName ("oracle.jdbc.driver.OracleDriver");

// Aufbau einer Verbindung:
    System.out.println ();
    System.out.println ("Verbindungsaufbau....");
    String user     = readEntry("user: ");;
    String password = readEntry("password: ");;
    String database = readEntry("Datenbank-connect-string: ");;


    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@" + database,
            user, password);

    System.out.println ("....Verbunden");
    System.out.println ();

// Erzeugen eines selects, das alle Produkte aus der Tabelle Produkt
// selektiert und ausgibt:


    String stmt1 = "select * from produkt";
    System.out.println (stmt1);
    System.out.println ();

    Statement s1 = conn.createStatement ();
    ResultSet rs1 = s1.executeQuery (stmt1);

    String jerg1 = new String();
    while (rs1.next ())
      {
      for (int i=1;i <= 4; i++)
        {
        jerg1 = jerg1 + rs1.getString(i) + "  ";
```

```
      }
      System.out.println (jerg1);
      jerg1 = "";
    }

    s1.close();


// Erzeugen eines updates, das in allen Positionen eines Auftrags
// aus der Tabelle Positionen die Stückzahl heraufsetzt:


    int anzahl;
    Statement s2 = conn.createStatement ();
    String stmt2 = "update positionen set stueckzahl = stueckzahl + 20
                    where auftragsnr = 2";
    int x = s2.executeUpdate(stmt2);
    conn.commit();
    int y = s2.getUpdateCount();
    s2.close();

    System.out.println ();
    System.out.println ("Anzahl der geänderten Datensätze: "+y);
    System.out.println ();


// Das update als prepared statement:

    PreparedStatement ps2 = conn.prepareStatement(stmt2);
    int x2 = ps2.executeUpdate();

    System.out.println ();
    System.out.println ("Anzahl der geänderten Datensätze: "+x2);
    System.out.println ();
    ps2.close();



// Aufruf einer stored procedure ins_pos, die zu vorgegebener
// Auftragsnr, produktnr und Stückzahl einen Datensatz in der Tabelle
// Positionen erzeugt:

    int ianr = Integer.parseInt(readEntry("die Auftragsnr: "));
    int ipnr = Integer.parseInt(readEntry("die Produktnr: "));
    int istckz = Integer.parseInt(readEntry("die Stückzahl: "));

    CallableStatement cs = conn.prepareCall("{call ins_pos(?,?,?)}");
    cs.setInt(1,ianr);
    cs.setInt(2,ipnr);
    cs.setInt(3,istckz);

    cs.execute();
    cs.close();

    String stmtps = "select * from positionen where auftragsnr = ?";
    PreparedStatement ps3 = conn.prepareStatement(stmtps);
    ps3.setInt(1,ianr);
    ResultSet rsps3 = ps3.executeQuery();

    System.out.println ("Ergebnis eines einfachen Prozedurenaufrufs: ");
    System.out.println ();

    String jerg3 = new String();
    while (rsps3.next ())
      {
      for (int i=1;i <= 4; i++)
        {
        jerg3 = jerg3 + rsps3.getString(i) + "  ";
      }
      System.out.println (jerg3);
      jerg3 = "";
    }

    ps3.close();

  }
}
```

Kapitel V Arbeiten mit dem Datentyp LOB

V.1 Allgemeine Überlegungen

Zur Verarbeitung großer Datenvolumina, wie sie typischerweise im Bereich von Multimedia-Anwendungen anfallen (Bilder, Videos, Audisequenzen,..), stellen Datenbanksysteme (also insbesondere auch relationale) inzwischen einen neuen Datentyp zur Verfügung, den Datentyp LOB (**L**arge **Ob**ject).
Wir werden die Arbeit mit solchen LOB's am Beispiel des Datenbanksystems der Firma Oracle (version 8.17) darstellen.

Nach einer grundsätzlichen Besprechung der Realisation durch Oracle werden wir den praktischen Umgang mit LOB's in Anwendungen mit Hilfe von PL/SQL und Java/JDBC erörtern.

V.2 Multimediadaten verarbeiten mit Oracle 8.1.7

V.2.1 Der Datentyp LOB

Oracle bietet zur Verarbeitung großer Datenmengen in einem Attribut (Bilder, Video- oder Audiosequenzen) einen eigenen Datentyp an: LOB (Large Object). Dieser Datentyp kann bis zu 4 GB Daten pro Datensatz speichern.

Oracle unterscheidet zwei prinzipiell in der Handhabung verschiedene Arten von LOB's: sogenannte **internal LOB**s und **externel LOB**s, letztere auch **BFILE**s genannt (Binary File). Bfiles sind Verweise auf Betriebsystemdateien im Filesystem.

V.2.1.1 Internal LOBs:

Interne LOB's werden innerhalb des Datenbanksystems gespeichert und fallen somit unter das Transaktionskonzept. Damit ist also auch das recovery von LOB-Daten im Fall von Systemfehlern möglich, und Änderungen an LOB-Daten können commited oder zurückgerollt werden. Oracle differenziert interne LOB's weiter in 3 Typen:

- **BLOB**: ein LOB mit unstrukturierten Binärdaten;
- **CLOB**: ein LOB, dessen Werte aus 1-Byte Charakter-Daten bestehen, entsprechend dem Charakterset, das für die Datenbank definiert wurde;
- **NCLOB**: ein LOB, dessen Daten aus Charakterdaten bestehen entsprechend einem nationalen Charakterset.

Bei inserts oder updates eines LOB-Attributes mit den Werten eines anderen LOB's wird der komplette Wert kopiert, d.h. es wird nicht referenziert, die betreffenden Spalten und Zeilen enthalten jeweils die kompletten LOB-Daten.
CLOB- und NCLOB-Daten werden mit dem 2 Byte Unicode für Charaktersets variabler Länge gespeichert. Beim Laden und Speichern aus und in CLOBs bzw NCLOBs werden also die Daten vom speziellen Charakterset-Code in Unicode übersetzt und umgekehrt.

V.2.1.2 External LOBs (BFILE)

Externe LOBs sind große Binärdatenobjekte, die nicht im Datenbanksystem, sondern im Dateisystem des Betriebssystems des Servers gespeichert sind. BFILEs unterliegen damit nicht dem Transaktionskonzept der Datenbank, insbesondere können also Änderungen der Daten nicht mit den üblichen Mechanismen commit, rollback, recover bearbeitet werden.

BFILEs werden in Tabellenspalten referenziert über einen sogenannten **locator** (der auch für interne LOBs benutzt wird, siehe nächsten Abschnitt). Wird also der BFILE einer Zeile einer Tabelle in einen anderen BFILE kopiert, wird nur dieser locator kopiert, die Daten bleiben im Dateisystem.

V.2.1.3 Der LOB-Locator

Eine Tabelle kann eine beliebige Anzahl von LOB-Attributen besitzen. Im Allgemeinen werden die LOB-Daten auch von internen LOBs **nicht** innerhalb der Tabelle gespeichert, sondern außerhalb, an irgend einer anderen Stelle im Tablespace oder auch in einem anderen Tablespace der Datenbank. Dies wird dadurch ermöglicht, daß in der entsprechenden Tabellenzeile ein sogenannter **locator** im LOB-Attribut gespeichert wird. Dieser locator ist ein Zeiger auf den aktuellen Speicherplatz der eigentlichen LOB-Daten. Jeder Datensatz in der Tabelle bekommt also einen eigenen locator-Wert für seine LOB-Daten. Bei internen LOBs werden dann die Daten innerhalb des Datenbanksystems an anderer Stelle gespeichert, bei externen LOBs als Dateien im Dateisystem des Betriebssystems. Ein dadurch entstehender Unterschied zwischen internen und externen LOBs besteht darin, daß zwei locators einer Tabelle (die zu zwei verschiedenen Datensätzen gehören) im Falle externer LOBs auf die gleiche Datei im Dateisystem verweisen können (also auf identische LOB-Werte), während sie bei internen LOBs immer auf verschiedene LOB-Werte zeigen.

Für interne LOBs gibt es eine eigene storage-Klausel zur Optimierung der Speicherplatz-Allokation (siehe Oracle-Dokumentation). Eine Besonderheit interner LOBs ist noch, daß bei LOB-Daten, die weniger als 4KB umfassen, diese LOB-Daten direkt in der Tabelle gespeichert werden können.

V.2.1.4 Operationen mit dem LOB-Locator

a) Interne LOBs

Bevor eine interne LOB-Spalte einer Tabelle mit Daten gefüllt werden kann, muß sie initialisiert werden, d.h. sie muß einen locator enthalten. Dies geschieht dadurch, daß beim Erzeugen der Tabelle oder beim ersten insert- bzw update-statement für diese Tabelle eine Funktion EMPTY_BLOB() oder EMPTY_CLOB() benutzt wird.
Beispiele:

Beim create table-statement:

```
Create table test (nr INTEGER, daten BLOB default empty_blob() );
```

Beim ersten insert:

```
Create table test (nr INTEGER, daten BLOB);
Insert into test values (1, empty_blob());
```

Hat das BLOB-Attribut daten einen locator, kann auf es zugegriffen werden (etwa über PL/SQL:

```
Declare
        Blobdaten BLOB;
Begin
        Select daten into blobdaten from test where nr = 1;
        /* jetzt kann mit dem locator-Wert in blobdaten der BLOB manipuliert
           werden!  */
End;
/
```

Zur Beachtung: in blobdaten steht der locator, nicht etwa die eigentlichen Daten des BLOB!!
Die Manipulation von LOB-Daten mit verschiedenen Werkzeugen (PL/SQL oder JDBC) wird später besprochen.

b) Externe LOBs

Bevor auf einen externen LOB (BFILE) zugegriffen werden kann, muß das BFILE-Attribut in der Tabelle initialisiert werden (mit einem locator versehen werden). Dazu muß dem Datenbankmanagementsystem zunächst bekannt gemacht werden, wo im Dateisystems des Servers die Datei liegt. Dies geschieht dadurch, daß im Datenbanksystem ein alias erzeugt wird für das Verzeichnis, genannt **directory**:

```
Create directory bilder as ‚c:\bilder';
```

Directories können mit drop directory <name>; wieder gelöscht werden.

Mit Hilfe eines existierenden directory kann nun ein BFILE-Attribut einer Tabelle initialisiert werden unter Benutzung der Funktion BFILENAME():

BFILENAME(<directoryname> IN varchar2, <filename> IN varchar2)

Rückgabewert ist BFILE, also ein locator.
BFILENAME kann in einemm insert- oder update-statement benutzt werden oder zur Initialisierung einer PL/SQL-Variablen.
Beispiel:

Gegeben sei die Tabelle test2 durch das statement:
Create table test2(nr INTEGER, exdaten BFILE);

Initialisierung:
Create directory bilder as ‘c:\bilder’;
Insert into test2 values(1, BFILENAME(‘BILDER’,’bild1.jpg’));

Wichtig ist, daß im Aufruf der Funktion BFILENAME der Name des directory in Großbuchstaben angegeben wird!

Allgemein kann mit einem delete-statement ein locator gelöscht werden:

Delete from test2 where nr = 1;

Hier wird der locator gelöscht, der BFILE existiert natürlich weiter als Datei im Dateisystem des Servers.
Anders bei internen LOBs:

Delete from test where nr = 1;

Dieses statement löscht locator und LOB-Daten aus der Datenbank!

Die Verarbeitung von LOB-Daten kann mit verschiedenen Werkzeugen geschehen. Wir werden zwei Möglichkeiten ansprechen: Verarbeiten mittels Pl/SQL und Verarbeiten mittels Java/JDBC.

V.2.2 LOB-Datenverarbeitung mit PL/SQL

Oracle bietet zur Verarbeitung von LOB-Daten mit seiner SQL-Spracherweiterung PL/SQL ein eigenes package an, das **DBMS_LOB-Package**.

Das Package DBMS_LOB besteht aus einer Reihe von Funktionen zum Lesen und Modifizieren von internen bzw. externen LOBs. Die folgenden Tabellen mit Funktionen und Beschreibung sind der Oracle-Dokumentation entnommen:

Table 3-3 PL/SQL: DBMS_LOB Procedures To Modify BLOB, CLOB, and NCLOB Values

Function/Procedure	Description
APPEND()	Appends the LOB value to another LOB
COPY()	Copies all or part of a LOB to another LOB
ERASE()	Erases part of a LOB, starting at a specified offset
LOADFROMFILE()	Load BFILE data into an internal LOB
TRIM()	Trims the LOB value to the specified shorter length
WRITE()	Writes data to the LOB at a specified offset
WRITEAPPEND()	Writes data to the end of the LOB

Table 3-4 PL/SQL: DBMS_LOB Procedures To Read or Examine Internal and External LOB values

Function/Procedure	Description
COMPARE()	Compares the value of two LOBs
GETCHUNKSIZE()	Gets the chunk size used when reading and writing. This only works on internal LOBs and does not apply to external LOBs (BFILEs).
GETLENGTH()	Gets the length of the LOB value
INSTR()	Returns the matching position of the nth occurrence of the pattern in the LOB
READ()	Reads data from the LOB starting at the specified offset
SUBSTR()	Returns part of the LOB value starting at the specified offset

Es folgt ein Beispiel zum Schreiben eines Text-Dokumentes aus einer externen Datei (BFILE) in eine CLOB-Spalte einer Tabelle unter Benutzung des DBMS_LOB-package:

Annahme, eine Tabelle DOKUMENT sei wie folgt angelegt:

```
Create table dokument( doknr      integer,
                       Autor      varchar2(30),
                       Titel      varchar2(100),
                       Jahr       date,
                       Filename   varchar2(50),
                       Filetype   char(3),
                       Groesse    integer,
                       Text   CLOB default empty_clob()  );
```

Alter table dokument add constraint pk_dokument primary key (doknr);

Create sequence dok_seq; (zur Erzeugung von PK-Werten)

Jetzt muß noch ein directory erzeugt werden, das auf den Ordner im Dateisystem verweist, in dem die Textdokumente sind:

Create directory dokumente as ‚c:\dokumente';

Die folgende stored procedure ladeDokument() benutzt das DBMS_LOB-package, um Textdokumente in die Tabelle dokument zu laden:

```
create or replace procedure ladeDokument(directory IN varchar2,
                                         datei IN varchar2 )
is
        dok_datei    BFILE:=BFILENAME(directory, datei);
        lob_loc      CLOB;
        loblength    integer;
        dnr          integer;
Begin
        Insert into dokument(doknr) values(dok_seq.nextval) returning doknr
        into dnr;
        select text into lob_loc from dokument where doknr = dnr for update;
        DBMS_LOB.OPEN(dok_datei, DBMS_LOB.LOB_READONLY);
        loblength := DBMS_LOB.GETLENGTH(dok_datei);
        DBMS_LOB.LOADFROMFILE(lob_loc, dok_datei, loblength);
        DBMS_LOB.CLOSE(dok_datei);
        Update dokument   set     filename = datei,
                                  groesse = loblength,
                                  filetype = substr(datei, length(datei)-2,
                                             length(datei))
                          where doknr = dnr;
        commit;
End;
/
```

Erläuterungen zur stored procedure:

- Die Variable dok_datei wird bei der Deklaration schon initialisiert mit dem locator-Wert der tatsächlichen Textdatei;
- Die Variable lob_loc ist der locator für die Aufnahme des locators der Tabelle dokument; dies geschieht durch das erste select-statement im Begin-End-Block;
- Das Öffnen, das Bestimmen der Länge des Textes, das Laden in die Spalte text und das Schließen des BFILEs geschieht jeweils mit der entsprechenden DBMS_LOB-Package-Funktion.
- Anschließend werden noch die restlichen Informationen über die Textdatei, die dann vorliegen, durch das update-statement in die Tabelle eingefügt.
- Wichtig ist noch, daß der Übergabeparamter directory der prozedur ladeDokument in Großbuchstaben übergeben wird!!

In ähnlicher Weise können natürlich auch beliebige Binärdaten (etwa Bilder, o. ä.) vom Dateisystem in eine Tabelle eingefügt werden. Dann muß mit dem Datentyp BLOB statt mit CLOB gearbeitet werden.

Natürlich bietet das DBMS_LOB-Package auch die Möglichkeit, LOB-Daten aus einem LOB-Attribut einer Tabelle auszulesen: die Funktion READ().
DBMS_LOB.READ() hat 4 Parameter:

Lob_loc	IN	BLOB oder CLOB oder BFILE
Amount	IN OUT	BINARY INTEGER
Offset	IN	INTEGER
Buffer	OUT	RAW,

Die den Locator, die Anzahl der Bytes(BLOB) bzw Charakter (CLOB), die gelesen werden sollen, das offset in Bytes (BLOB) bzw Charakter (CLOB), von dem aus gelesen werden soll, und den Outpu Buffer.
Auch hier ein Beispiel:

```
create or replace function holeText (did integer) return varchar2
is
        btext clob;
        laenge integer;
        fliesstext varchar2(4000);

Begin
        select text into btext from dokument where doknr = did;
        laenge := DBMS_LOB.getlength(btext);
        DBMS_LOB.read(btext, laenge, 1, fliesstext);
        return(fliesstext);
End;
/
```

Die Funktion holeText() liest aus dem CLOB-Attribut der Tabelle dokument den Fließtext und gibt ihn als varchar2 zurück. Der Text kann nun z.B. von einer Applikation weiterverarbeitet werden. Das Öffnen des LOBs mittels der Funktion DBMS_LOB.OPEN() ist hier optional und wird im Beispiel nicht gemacht.

V.2.3 LOB-Datenverarbeitung mit Java/JDBC (jdk 1.2 bzw 1.3)

Oracle bietet für seine JDBC-Treiber die folgenden oracle.sql.* Klassen zur Ünterstützung der LOB-Verarbeitung an:

- oracle.sql.BLOB
- oracle.sql.CLOB
- oracle.sql.BFILE

Bevor mit einem LOB gearbeitet werden kann, muß erst der locator aus der Tabelle gelesen werden. Dann können LOB-Daten gelesen, geschrieben oder manipuliert werden.

Die Klassen **oracle.sql.BLOB** und **oracle.sql.CLOB** implementieren die **java.sql.Blob** bzw. **java.sql.Clob** Interfaces. Instanzen dieser Klassen enthalten nur den locator, nicht die eigentlichen LOB-Daten.

BLOB oder CLOB locator bekommt man mit den **getBlob()** und **getCLOB()** Methoden. Bei einem Standard JDBC resultset (java.sql.Resultset) oder callable statement (java.sql.CallableStatement) geben getBlob() und getClob() jeweils java.sql.Blob- bzw java.sql.Clob-Objekte zurück. Benutzt man das resultset eines OracleResultSet oder OracleCallableStatement, bekommt man mit getBlob() und getClob() oracle.sql.BLOB- bzw. oracle.sql.CLOB-Objekte zurück. Benutzt man die Methoden getObject() bzw getOracleObject(), muss man casten:

```
ResultSet rs = stmt.executeQuery(„select blob_col from lob_table);
While (rs.next() )
{
        java.sql.Blob blob = (java.sql.Blob) rs.getObject(1);
        oracle.sql.BLOB blob1 = (oracle.sql.BLOB) rs.getOracleObject(1);
}
```

Analog zu den get-Methoden gibt es auch setBLOB usw. –Methoden, um locator in Tabellen zu setzen.

Hat man einen locator erhalten kann man auf die eigentlichen LOB-Daten zugreifen. Diese werden entweder als **array oder als bytestream** gelesen bzw geschrieben.

Beispiel:

Das folgende Codefragment ist Teil einer Methode zum Lesen eines Bildes aus einem BLOB-Attribut ‚bilddaten' einer Tabelle:

```
ResultSet rs= stmt.executeQuery(stmt_text);

if( rs.next() ) {
    oracle.sql.BLOB bild = ( (OracleResultSet) rs).getBLOB("bilddaten");
    BufferedInputStream bis = new BufferedInputStream(new OracleBlobInputStream(bild));
    byte[] data = new byte[(int)bild.length()];
    int val = 0;
    for ( int i=0; (val=bis.read())!= -1; i++) {
      data[i] = (byte)val;
    }
    return tk.createImage(data);

}
```

Das folgende Beispiel ist Teil einer Methode und schreibt ein Bild aus einem Java-Programm in ein BLOB-Attribut einer Tabelle:

```
oracle.sql.BLOB bild = null;
Statement stmt2= conn.createStatement();
ResultSet rs = stmt2.executeQuery("select bilddaten from bild where bildnr=" +
                                                  String.valueOf(bnr) + " for update");
if ( rs.next()) {
    bild = (oracle.sql.BLOB) ( (OracleResultSet) rs).getBlob(1);
}
else return 0;
// Schreiben des Bilds in die Datenbank:
```

```
ImageWriter iw = new ImageWriter(new FileInputStream(this.file), bild.getBinaryOutputStream());
iw.run();
```

In diesem Beispiel wird wesentlich benutzt die Methode **getBinaryOutputStream()** des oracle.sql.BLOB-Objekts bild, um die BLOB-Daten zu schreiben. Voraussetzung ist, dass bereits ein Bild in Form eines FileInputStreams im Java-Programm geladen ist.
Die Klasse ImageWriter dient lediglich dazu, in einem eigenen Thread einen inputstream in einen outputstream zu kopieren.

Fazit:
Mit den Methoden der Klassen ResultSet und BLOB bzw CLOB können beliebige LOB-Daten aus Tabellen gelesen und in Tabellen geschrieben werden. Dabei erweitern die von Oracle bereitgestellten Klassen oracle.sql.BLOB bzw oracle.sql.CLOB die Interfaces java.sql.Blob bzw java.sql.Clob um viele Methoden und Funktionalitäten zur Verarbeitung von LOB-Daten. Für weitergehende Informationen verweise ich auf die Dokumentationen von Java (jdk1.3) und Oracle (jdbc) und die einschlägige Literatur.

Literatur:

J. J. Date: An introduction to database systems
Addison-Wesley 1995

C. J. Date, H. Darwen: A guide to the SQL standard
Library of Congress Cataloging-in-Publication Data 1997

A. Heuer, G. Saake: Datenbanken: Konzepte und Sprachen
MITP GmbH Bonn 2000

G. Saake, A. Heuer: Datenbanken: Implementierungstechniken
MITP GmbH Bonn 1999

G. Vossen: Datenbankmodelle, Datenbanksprachen und Datenbank-Management-Systeme
Addison-Wesley 1995

Diverse Handbücher des Datenbankherstellers ORACLE