3. Relationale Datenbanken

3.1 Das relationale Datenmodell

Das relationale Datenmodell ist heute das bei weitem wichtigste unter den verschiedenen Datenmodellen, die von DBMS unterstützt werden. Es ist gekennzeichnet durch außerordentliche Einfachheit seiner Strukturen und es hat, wie kein anderes praktisch bedeutsames Modell, eine komplette formale Basis. Die grundlegenden Arbeiten zum relationalen Datenmodell stammen von Codd /COD70/. Die zum relationalen Datenmodell entwickelte Datenmanipulationssprache SQL hat sich als Standard durchgesetzt.

Im relationalen Modell werden die Daten der realen Welt als eine Sammlung von *Relationen* dargestellt. Anschaulich kann man sich eine Relation als Tabelle vorstellen. Sämtliche Informationen, Entities wie Beziehungen zwischen diesen, werden über solche „Tabellen“ modelliert. Entsprechend sind die Objekte für die Datenmanipulation Tabellen.

Erinnern wir uns an die mathematische Definition einer Relation:

Sind $D_1, D_2, ..., D_n$ Mengen von Werten, so ist
$R \subseteq D_1 \times D_2 \times ... \times D_n$ eine n-stellige *Relation* über den Mengen (*domains*) und
n ist der Grad (*degree*) der Relation.

Relation, domain

($D_i \times D_j$ steht für das kartesische Produkt.)

Ein Element $r = (d_1, d_2, ..., d_n) \in R$ $(d_i \in D_i, i = 1, ..., n)$ ist ein *Tupel* der Relation R (n-Tupel). d_i ist die i-te *Komponente* des Tupels.

Tupel

Beispiel 3.1:

D_1 sei die Menge {rot, blau, grün}, D_2 die Menge {0,1}. Dann ist das kartesische Produkt $D_1 \times D_2$ = {(rot,0) , (rot,1) , (blau,0) , (blau,1), (grün,0), (grün,1)}. Jede Teilmenge dieser Menge ist eine zweistellige Relation, etwa R = {(rot,0),(rot,1), (grün,1)}, aber auch die leere Menge.

Ein Tupel einer Relation beschreibt offensichtlich ein Objekt unseres Interesses, ein Entity, über die Kombination seiner Werte. Ein solches Tupel ist beispielsweise Name, Beruf, Wohnort, Geburtsjahr des Angestellten Walter. Die Relation entspricht dann der Menge aller Angestellten (Wir werden jedoch später sehen, dass die Gleichsetzung von Tupel und Entity nicht immer sinnvoll ist.).

Entity - Tupel

In einer Datenbank verändert sich natürlich die Zahl der Tupel in einer Relation über der Zeit, etwa durch Neueinstellung von Angestellten, ebenso wie sich die Werte innerhalb einzelner Tupel verändern können, etwa durch die Erhöhung eines Gehaltes. Ähnlich wie bei der Beschreibung der Struktur von Entities über Entity-Typen beschreiben wir deshalb die Eigenschaften einer Relation (gleichartige Menge von Tupeln!) durch das Relationenschema.

Relation und Relationenschema

Ein *Relationenschema* $R(A_1, ..., A_n)$ spezifiziert eine Relation mit Namen R und mit den paarweise verschiedenen Attributen $A_1, ..., A_n$ (mit den Attributnamen $A_1, ..., A_n$). Jedem Attribut A_i ist ein Wertebereich $dom(A_i)$ zugeordnet. Die Wertmengen verschiedener Attribute können natürlich identisch sein.

Die zu $R(A_1, ..., A_n)$ gehörigen Relationen sind also sämtlich Relationen des Typs $R \subseteq dom(A_1) \times dom(A_2) \times ... \times dom(A_n)$.

In einer Datenbank existiert zu jedem Zeitpunkt genau eine Relation R zum Relationenschema $R(A_1, ..., A_n)$, nämlich diejenige mit den gerade gültigen Werten (entsprechend den zu diesem Zeitpunkt existierenden Entities). Man sagt auch R sei eine Ausprägung von $R(A_1, ..., A_n)$.

Ein Relationenschema zum Beispiel 3.1 wäre FARBMENGE(FARBE, MENGE), wobei FARBE der Wertebereich D_1 und MENGE der Wertebereich D_2 zugeordnet ist.

Oft wird der Unterschied zwischen Relationenschema und Relation sprachlich nicht sauber eingehalten, vielmehr spricht man - wo keine Unklarheiten auftreten - durchgängig einfach von Relation. Im Beispiel sprechen wir also von der Relation FARBMENGE.

Relation als Tabelle

Seine große Anschaulichkeit erhält das relationale Datenmodell dadurch, dass man Relationen als Tabellen darstellen kann - und jedermann weiß mit Tabellen umzugehen. Jede Zeile der Tabelle entspricht einem Tupel der Relation. Beachten Sie, dass wegen der Definition einer Relation als *Menge* in der Tabelle keine gleichen Zeilen auftreten können. Die Spalten der Tabelle werden entsprechend den Attributnamen benannt. Man kann damit die Spalten unabhängig von ihrer Reihenfolge direkt ansprechen und in beliebiger Reihenfolge aufschreiben.

Beispiel 3.2:

Die folgende Tabelle zeigt eine Relation mit Daten zu den Angestellten eines Unternehmens mit dem Relationenschema

ANGEST(ANGNR, NAME, WOHNORT, BERUF, GEHALT, ABTNR).

(198, 'SCHMIDT', 'KARLSRUHE', 'KAUFMANN', 7500, 4)

ist ein Tupel der Relation.

ANGEST	ANGNR	NAME	WOHNORT	BERUF	GEHALT	ABTNR
	112	MÜLLER	KARLSRUHE	PROGRAMMIER	4500	3
	205	WINTER	HAGEN	ANALYTIKER	7800	3
	117	SEELER	MARBURG	INGENIEUR	6000	5
	198	SCHMIDT	KARLSRUHE	KAUFMANN	7500	4
	⋮	⋮	⋮	⋮	⋮	⋮

Die Wertebereiche für NAME, WOHNORT und BERUF definieren wir als *String* (Zeichenreihe), den Wertebereich für ANGNR, GEHALT und

ABTNR etwa als dom(ANGNR) = dom(GEHALT) = dom(ABTNR) = Integer, also die Menge der natürlichen Zahlen.

elementares Attribut

Jedes Attribut einer Relation muss *elementar* sein, d. h. der Wert eines Attributes ist unteilbar, er setzt sich nicht aus mehreren anderen Werten zusammen. Zusammengesetzte oder mehrwertige Werte sind nicht erlaubt.

1. Normalform

Eine Relation bei der jedes Attribut elementar ist, ist in der **ersten Normalform (1NF)**. Sämtliche theoretischen Arbeiten über relationale Datenbanken gehen von Relationen in 1. Normalform aus.

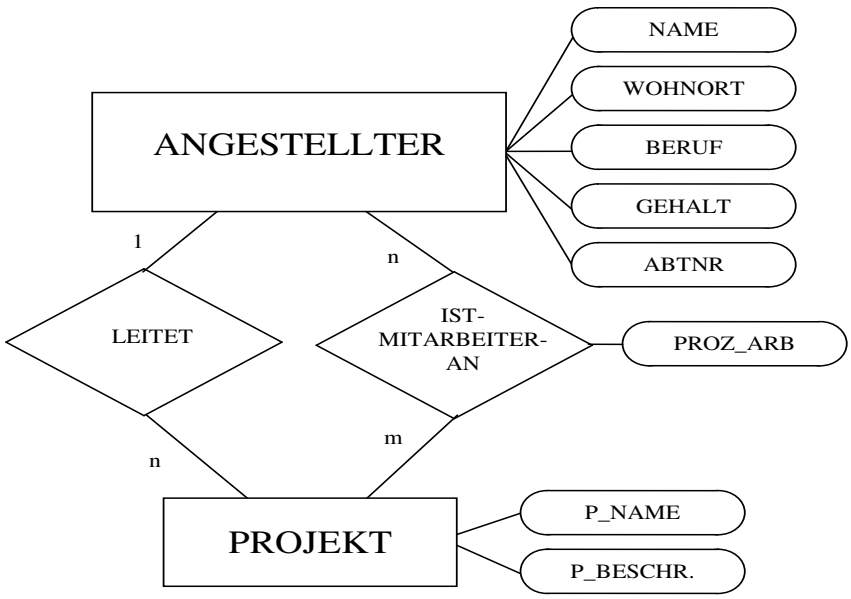
In einigen Anwendungen ist die erste Normalform sehr hinderlich. Aus diesem Grunde entstanden Überlegungen zu relationalen Systemen, die diese Restriktion teilweise lockern, etwa indem listen- oder mengenwertige Attribute zugelassen werden. Einen kurzen Blick auf diese Entwicklung zu *non-first normal form relations* (NF^2) werfen wir am Ende des Kurses.

Schema der relationalen Datenbank

Die Gesamtheit der Relationenschemata einer Datenbank heißt *Schema der (relationalen) Datenbank*. Zum Schema der Datenbank gehört darüber hinaus die Menge der *Integritätsbedingungen*, die Aussagen über die Korrektheit von Werten in der Datenbank machen. Auf diesen Aspekt gehen wir später ein.

Modellierung im relationalen Datenmodell

Die hervorstechende Eigenschaft des relationalen Datenmodells ist es, dass ein einziges Konzept zur Modellierung von Daten benützt wird: das Relationenschema. Es gibt keinen Unterschied zwischen Entities und Beziehungen zwischen Entities: beide werden durch Relationen ausgedrückt. Wir wollen dies am Beispiel der Entity-Typen ANGESTELLTER und PROJEKT und der Beziehungstypen IST-MITARBEITER-AN und LEITET diskutieren.



Die Entity-Typen ANGESTELLTER und PROJEKT können wir durch die Relationenschemata

ANGEST (ANGNR, NAME, WOHNORT, BERUF, GEHALT, ABTNR)

und

PROJEKT (PNAME, PNR, P_BESCHR, P_LEITER)

darstellen. Die m:n-Beziehung zwischen beiden stellen wir durch ein weiteres Relationenschema ANG_PRO dar:

ANG_PRO (PNR, ANGNR, PROZ_ARB).

Dabei nehmen wir an, dass Entities vom Typ ANGESTELLTER durch das Attribut ANGNR, Entities vom Typ PROJEKT durch PNR identifiziert werden. Die Relation ANG_PRO enthält für jede Beziehung zwischen einem Angestellten und einem Projekt genau ein Tupel.[1]

Sie sehen, dass Beziehungen mit Attributen - im Beispiel PROZ_ARB - auf sehr naheliegende Weise dargestellt werden können.

Ebenfalls sieht man, dass die Art der Beziehung (1:1, 1:n, n:m) nicht von Bedeutung ist: jede Beziehung zwischen beliebig vielen Entity-Typen E_1, E_2, ..., E_n kann unmittelbar durch ein Relationenschema

$R(e_1, ..., e_n)$

ausgedrückt werden, wobei e_i identifizierende Attribute der Entity-Typen sind. Bei entsprechender Komplexität der Beziehung B (1:n, bzw. 1:1 Beziehung) zwischen E und E' (Relationen R und R') muss keine eigene Relation für B geschaffen werden; es genügt z.B. die Hinzunahme eines Schlüssels von R' zu den Attributen von R. Im Beispiel wird die 1:n-Beziehung LEITET durch die Hinzunahme von ANGNR aus ANGEST zur Relation PROJEKT als P_LEITER abgebildet. Genauer gehen wir auf die Abbildung von ER-Diagrammen in relationale Schemata in Abschnitt 3.6.5 ein.

Schlüssel

Schlüssel

Minimalitätseigenschaft von Schlüsseln

Da eine Relation eine Menge ist, sind die Tupel unterscheidbar. Es gibt also für jede Relation R eine Menge von Attributen – im Extremfall alle Attribute von R, deren Werte für die Tupel identifizierend sind. Solche Attributmengen nennen wir *Schlüssel*, wenn sie *minimal* sind. D.h.: Ist X ein Schlüssel von R, so ist keine Teilmenge von X ebenfalls Schlüssel von R.

Beim Schemaentwurf spezifiziert man mit dem Relationenschema, welche Attributmenge einen Schlüssel bilden soll. Es dürfen dann beim Arbeiten mit der Datenbank nur solche Tupel eingefügt werden, die die Schlüsseleigenschaft der gewählten Attributmenge nicht verletzen.

---

1 Solche identifizierenden Attribute können, müssen aber nicht notwendigerweise bereits im ER-Schema enthalten sein. Im Relationenmodell hingegen können Beziehungen ohne diese Attribute nicht modelliert werden.

Beispiel 3.3:

Relationenschema ANG_PRO (ANGNR, PNR, PROZ_ARB):
{ANGNR, PNR} ist Schlüssel,
{ANGNR} ist kein Schlüssel,
{PNR} ist kein Schlüssel,
{ANGNR, PNR, PROZ_ARB} ist kein Schlüssel.

mehrere Schlüssel

Eine Relation kann mehrere Schlüssel haben. Beispielsweise kann ein Angestellter identifiziert werden über seine Angestelltennummer, aber auch über Name, Geburtstag und Adresse. In diesem Falle wird meist ein Schlüssel ausgezeichnet und zum *Primärschlüssel* erklärt. Die anderen Schlüssel werden oft Schlüsselkandidaten oder *candidate keys* genannt; sie spielen bei einigen Entwurfsverfahren für relationale Datenbanken eine Rolle.

Primärschlüssel
candidate key

Wo der Primärschlüssel von Bedeutung ist, werden wir im Relationenschema die zugehörigen Attribute unterstreichen.

Arbeiten mit relationalen Datenbanken

Abgeschlossenheit

Ebenso wie alle Daten in der Datenbank als Relationen dargestellt werden, ist auch das Ergebnis einer Abfrage (Query) eine Relation. Insofern ist das relationale Datenmodell abgeschlossen. Ein relationales Datenbanksystem muss mächtige Operationen zur Verfügung stellen, um aus den vorhandenen Relationen die gewünschte Relation ableiten zu können. Diese Operationen müssen es ermöglichen, Mengen von Tupeln mit gewissen Eigenschaften aus einer Relation herauszufinden, aber auch durch Streichen von Spalten Relationen kleineren Grades oder durch Kombination von Relationen solche größeren Grades zu erzeugen. Das prinzipielle Arbeiten mit einer relationalen Datenbank wird vielleicht am anschaulichsten, wenn man sich das übliche Arbeiten mit Tabellen vor Augen hält.

Wir wollen dies an einem einfachen Beispiel zeigen.

Beispiel 3.4:

Betrachten wir die Relationschemata

ANGEST (ANGNR, NAME, WOHNORT, BERUF, GEHALT, ABTNR),
PROJEKT (PNAME, PNR, P_BESCHR, P_LEITER),
ANG_PRO (PNR, ANGNR, PROZ_ARB) wie o.a. und zusätzlich
KUNDE (KDNR, NAME, WOHNORT, TÄTIG_ALS).

Diese Relationenschemata werden wir im Folgenden öfter für Beispiele heranziehen.

Eine Tabelle zur Relation ANGEST ist in Beispiel 3.2 angegeben. Für die Relationen PROJEKT, ANG_PRO und KUNDE finden Sie Tabellen im Folgenden[2]:

PROJEKT	P_NAME	PNR	P_BESCHR	P_LEITER
	DATAWAREHOUSE	12	...	205
	INTRANET	18	...	117
	PROJEKT 2000	17	...	198
	VU	33	...	198
	. . .	. . .	. . .	. . .

ANG_PRO	PNR	ANGNR	PROZ_ARB
	12	205	100
	18	117	20
	33	117	80
	17	198	30
	18	198	70
	17	112	100
	. . .	. . .	. . .

KUNDE	KDNR	NAME	WOHNORT	TÄTIG_ALS
	743	KLÖTERJAHN	HAMBURG	KAUFMANN
	801	LEANDER	DAVOS	ARZT
	324	OSTERLOH	DAVOS	VERWALTUNG
	227	SCHMIDT	KARLSRUHE	KAUFMANN
	. . .	. . .	. . .	. . .

Eine typische Abfrage ist nun die folgende:

FINDE DIE WOHNORTE ALLER ANGESTELLTEN,
DEREN BERUF PROGRAMMIERER IST.

Die Abfrage kann durch Durchsuchen aller Tupel der Relation beantwortet werden: für jedes Tupel der Relation ANGEST, in dem

BERUF = 'PROGRAMMIERER',

wird der Wert der Komponente WOHNORT ausgedruckt.

---

2 Damit Sie während der Bearbeitung der Kurseinheit nicht immer wieder auf dieser Seite nachschlagen müssen, finden Sie diese Tabellen nochmals auf einer Seite im Anhang, die Sie aus der Kurseinheit heraustrennen können.

Eine komplexere Abfrage ist die folgende:

FINDE DIE NAMEN ALLER ANGESTELLTEN, DIE AM PROJEKT MIT DER NUMMER 17 MITARBEITEN.

Diese Abfrage kann nicht durch Durchsuchen einer einzigen Relation beantwortet werden. Man könnte folgendermaßen vorgehen: ermittle in der Relation ANG_PRO die Nummern aller Angestellten, die an Projekt 17 mitarbeiten. Für jede dieser Angestellten-Nummern finde man in Relation ANGEST den Namen des Angestellten. Ein zweiter Weg zur Beantwortung dieser Abfrage besteht darin, aus ANGEST und ANG_PRO eine größere Relation A-P-A zu machen: die beiden Tabellen werden bezüglich der Spalte ANGNR zusammengesetzt - Zeilen aus ANGEST und ANG_PRO, die denselben Wert für ANGNR haben, werden zu einer Zeile in A-P-A zusammengefasst:

A_P_A	ANGNR	NAME	PNR	PROZ_ARB
	112	MUELLER	17	100
	205	WINTER	12	100
	117	SEELER	18	20
	117	SEELER	33	80
	198	SCHMIDT	17	30
	198	SCHMIDT	18	70
	. . .	. . .	. . .	. . .

Aus der so entstandenen Tabelle sucht man nun wieder diejenigen Tupel heraus, für die PNR = 17, und gibt die entsprechenden Werte von NAME aus.

Diese Beispiele mögen ausreichen, um eine Idee vom Arbeiten mit relationalen Datenbanken zu vermitteln. Natürlich müssen Sprachen für relationale Datenbanken neben der Abfrage auch Änderungen ermöglichen - wir müssen Tupel einfügen, löschen und verändern können. Einige der Sprachen, die für relationale Datenbanken entwickelt und auch implementiert wurden, werden später diskutiert werden. Die heute bei weitem wichtigste Sprache ist SQL.

3.2 Grundlegende Sprachen für relationale Datenbanken

3.2.1 Einführung

Im relationalen Datenmodell denken wir in Relationen oder Mengen. Sprachen für dieses Modell müssen es deshalb erlauben, gewünschte Relationen, die aus den vorhandenen Relationen zu erzeugen sind, zu spezifizieren. Ferner müssen Möglichkeiten zur Veränderung von Relationen und zur Einführung neuer und zum Löschen vorhandener Relationen gegeben sein.

Es gibt eine ganze Reihe von Vorschlägen für relationale Datenmanipulationssprachen. Alle diese Sprachen lassen sich jedoch auf einen der beiden folgenden grundlegenden Ansätze zurückführen, oder sind Mischformen hiervon:

1. **Relationenalgebra**
   Spezifikation von gewünschten Relationen durch Angabe einer Folge von Operationen, mit der die Relationen aufgebaut werden sollen. Der Benutzer wendet spezielle Operationen auf Relationen an, um seine gewünschte Relation zu konstruieren.

2. **Relationenkalkül**
   Spezifikation von gewünschten Relationen in *deskriptiver* Weise, d. h. ohne Angabe, welche Operationen zum Aufbau der Relation verwendet werden sollen. Mit Hilfe des *Prädikatenkalküls* wird die Menge der gewünschten Tupel beschrieben: es wird ein Prädikat (eine Bedingung) angegeben, das die Tupel erfüllen müssen.

Retrieval

Kern jeder Datenmanipulationssprache sind diejenigen Komponenten, die die Datenauswahl, d.h. das Lesen der gewünschten Daten, betreffen *(Retrieval)*. Sind Daten zu verändern, so ist der schwierigste Schritt, die zu verändernden Daten erst einmal aufzufinden; die Veränderung der gefundenen Daten ist im relationalen Datenmodell dann sehr einfach.

Anmerkung: Gelegentlich wird begrifflich getrennt zwischen Abfragesprachen (Query Languages) und Datenmanipulationssprachen: Abfragesprachen erlauben ausschließlich das Lesen von Daten. Die Begriffe werden jedoch nicht einheitlich verwendet. Wir machen keinen Unterschied zwischen Abfragesprachen und Datenmanipulationssprachen.

3.2.2 Relationenalgebra

In der Mathematik bezeichnet man ein System, das aus einer nichtleeren Menge und einer Familie von Operationen (häufig auch „Operatoren“) auf dieser Menge besteht, als Algebra. Eine Relationenalgebra definiert also eine Menge von Operationen auf Relationen, mit deren Hilfe neue Relationen erzeugt werden können.

Wir haben im vorigen Kapitel bereits gesehen, welcher Art diese Operationen sein müssen: wir müssen beispielsweise Spalten und Zeilen aus Tabellen entfernen oder Tabellen zu neuen Tabellen zusammenfassen können. Wir betrachten zunächst einen minimalen Satz von Operationen zur Definition der relationalen Algebra, und diskutieren anschließend einige weitere nützliche Operationen. Minimal soll heißen, dass keine dieser Operationen durch die anderen ausgedrückt werden kann und somit weggelassen werden könnte; gleichzeitig ist dieser Satz von Operationen relational vollständig: nur wenn diese Operationen verfügbar sind, kann man mit der relationalen Algebra dieselben Abfragen ausdrücken wie mit dem Relationenkalkül, das gewissermaßen als Maßstab für die Mächtigkeit einer relationalen Sprache dient (siehe auch Seite 24).

Grundoperationen der relationalen Algebra

Im Folgenden wird ein grundlegender Satz von Operationen für die Relationenalgebra vorgestellt. Im Anschluss wird die Anwendung dieser Operationen in Beispiel 3.5 noch einmal zusammenfassend verdeutlicht.

Seien im folgenden $R(R_1, ..., R_n)$ und $S(S_1, ..., S_m)$ Relationen.

Wir beginnen mit zwei mengenorientierten Operationen – der Vereinigung und der Differenz:

Vereinigung:
Dies ist die Vereinigung der Tupelmengen aus R und S. Als Voraussetzung müssen R und S *vereinigungsverträglich* sein, d.h. sie müssen die gleichen Attribute (bzgl. Namen und Wertebereichen) haben.

R und S heißen *vereinigungsverträglich*, falls gilt:
$\{R_1, ..., R_n\} = \{S_1, ..., S_m\}$, d.h.:
n = m und
es gibt eine Permutation $\tau:\{1, ..., n\}\rightarrow\{1, ..., n\}$,
so dass für alle $i \in \{1, ..., n\}$
Name und Wertebereich von R_i und $S_{\tau(i)}$ gleich sind.

Seien R und S vereinigungsverträglich.

Dann heißt $V = R \cup S$ die *Vereinigung von R und S* und ist folgendermaßen definiert:
V hat das Schema $(R_{i1}, ..., R_{in})$ mit
$\{R_{i1}, ... R_{in}\} = \{R_1, ..., R_n\}$ und $(R_{i1}, ..., R_{in})$ alphabetisch geordnet[3]
und sei ferner $(R_{j1}, ..., R_{jn}) = (S_1, ..., S_n)$.
$v = (a_{i1}, ..., a_{in}) \in V \Leftrightarrow (a_1, ..., a_n) \in R$ oder $(a_{j1}, ..., a_{jn}) \in S$.

Differenz:
Dies ist die Menge der Tupel in R, die nicht auch in S enthalten sind. R und S müssen auch hier vereinigungsverträglich sein.

Seien R und S vereinigungsverträglich.

Dann heißt $D = R - S$ die *(Mengen-)Differenz von R und S* und ist folgendermaßen definiert:
D hat das Schema $(R_{i1}, ..., R_{in})$ mit
$\{R_{i1}, ..., R_{in}\} = \{R_1, ..., R_n\}$ und $(R_{i1}, ..., R_{in})$ alphabetisch geordnet
und sei ferner $(R_{j1}, ..., R_{jn}) = (S_1, ..., S_n)$.
$d = (a_{i1}, ..., a_{in}) \in D \Leftrightarrow (a_1, ..., a_n) \in R$ und $(a_{j1}, ..., a_{jn}) \notin S$.

Zwei weitere Operationen verändern im Wesentlichen die Struktur von Relationen – das kartesische Produkt erweitert sie und die Projektion schränkt sie ein:

---

3 Durch diese Ordnung auf den Attributen wird die Kommutativität der „mathematischen Vereinigung“ auch für die „Relationenvereinigung“ (also mit Schemata!) erhalten. Jede andere von den Ausgangsrelationen unabhängige Ordnung leistet das ebenfalls.

Kartesisches Produkt:
Dies ist die Relation, die alle möglichen Kombinationen von Tupeln aus R einerseits und S andererseits enthält. Das kartesische Produkt hat alle Attribute von R und S. Voraussetzung ist, dass alle Attribute von R und S verschieden sind.

Sei $\{R_1, ..., R_n\} \cap \{S_1, ..., S_m\}$ die leere Menge.

Dann heißt $K = R \times S$ das *kartesische Produkt von R und S* und ist folgendermaßen definiert:

K hat das Schema $(K_1, ..., K_{n+m})$ mit
$\{K_1, ..., K_{n+m}\} = \{R_1, ..., R_n\} \cup \{S_1, ..., S_m\}$ und
$(K_1, ..., K_{n+m})$ alphabetisch geordnet
und ferner sei $(K_{i1}, ..., K_{in}) = (R_1, ..., R_n)$ und
$(K_{j1}, ..., K_{jm}) = (S_1, ..., S_m)$
$k = (a_1, ... a_{n+m}) \in K \Leftrightarrow (a_{i1}, ..., a_{in}) \in R$ und $(a_{j1}, ..., a_{jm}) \in S$.

Vergleiche auch Beispiel 3.1.

Projektion:
Die Projektion dient dazu, Attribute aus Relationen zu entfernen. Sind für eine Auswertung nur die Attribute $R_{i1}, ..., R_{ik}$ interessant, dann werden einfach alle anderen Attribute der Relation R weggelassen. Die so entstehende *Projektion von R auf $R_{i1}, ..., R_{ik}$* wird als $P = \pi_{Ri1, ..., Rik}(R)$ geschrieben. Formal kann die Projektion so definiert werden:

Sei $\{R_{i1}, ..., R_{ik}\} \cup \{R_{j1}, ..., R_{j\,n-k}\} = \{R_1, ..., R_n\}$ mit
$\{R_{i1}, ..., R_{ik}\} \cap \{R_{j1}, ..., R_{j\,n-k}\}$ leer.

Dann heißt $P = \pi_{Ri1, ..., Rik}(R)$ die *Projektion von R auf $R_{i1}, ..., R_{ik}$* und ist folgendermaßen definiert:

P hat das Schema $(R_{i1}, ..., R_{ik})$.
$p = (a_{i1}, ..., a_{ik}) \in P$,
falls $\exists \{a_{j1}, ..., a_{j\,n-k}\} \in dom(R_{j1}) \times ... \times dom(R_{j\,n-k})$, so dass $(a_1, ..., a_n) \in R$.

P enthält keine Duplikate: Mehrere Tupel mit gleichen Attributwerten für $R_{i1}, ..., R_{ik}$ fallen in P zu einem Tupel zusammen, da Relationen als Mengen definiert sind.

Mit der Selektionsoperation kann der Inhalt einer Relation untersucht werden:

Selektion:
Mit der Selektion werden aus einer Relation alle Tupel ausgewählt, die eine gegebene Bedingung erfüllen. Die Bedingung muss für jedes Tupel für sich nachprüfbar sein.

Sei B eine Bedingung, wobei nur Attribute von R als freie Variablen vorkommen. Für ein Tupel $t \in R$ sei B(t) der (Wahrheits-)Wert von B, wenn für die Attribute in B die entsprechenden Attributwerte aus t eingesetzt werden.

Dann heißt $S = \sigma_B(R)$ die *Selektion auf R bzgl. der Selektionsbedingung B* und ist folgendermaßen definiert:

S hat das Schema $(R_1, ..., R_n)$.

$s = (a_1, ..., a_n) \in S$, falls $s \in R$ und B(s) gilt.

Mit Hilfe der Operationen der Relationenalgebra können nun beliebig komplexe Ausdrücke gebildet werden. Die Abarbeitung erfolgt jeweils von links nach rechts; andere Reihenfolgen können wie üblich durch Klammerung erzwungen werden.

Alternative Definitionen der Relationenalgebra behandeln die Schemata nicht als integrierten Bestandteil der Relationen, sondern orientieren sich eher an dem „rein“ mathematischen Begriff der Relation. Die Mengenoperationen werden dann in der Definition einfacher, die Arbeit mit den Operatoren wird jedoch umständlicher, da neben den Attributen auch deren Stellung im Tupel berücksichtigt werden muss.

Obwohl die Umbenennung von Attributen üblicherweise nicht zur relationalen Algebra gezählt wird, wird diese Operation (auch in alternativen Definitionen) benötigt, um komplexe Ausdrücke ohne Namenskonflikte oder Mehrdeutigkeiten erstellen zu können. Deshalb definieren wir im Folgenden die Umbenennung formal wie eine relationale Operation:

Umbenennung:
Durch die Umbenennung erhalten die Attribute einer Relation neue Bezeichnungen (bzw. Namen). Die Wertebereiche der Attribute bleiben unverändert.

Seien $(R_1', ..., R_n')$ n Attribute mit $dom(R_i') = dom(R_i)$ für alle $i \in \{1, ..., n\}$.

Dann heißt $U = \rho_{R1', ..., Rn'}(R)$ die *Umbenennung der Attribute von R nach $R_1', ..., R_n'$ in R* und ist folgendermaßen definiert:

U hat das Schema $(R_1', ..., R_n')$.

$u \in U$, falls $u \in R$.

Falls nur eine kleine Zahl von Attributen $\{R_{i1}, ..., R_{ik}\}$ umbenannt wird, dann schreiben wir vereinfachend auch: $U = \rho_{Ri1' \leftarrow Ri1, ..., Rik' \leftarrow Rik}(R)$.

Beispiel 3.5:

Erinnern wir uns an die letzte Abfrage aus Beispiel 3.4

> FINDE DIE NAMEN ALLER ANGESTELLTEN, DIE AM PROJEKT MIT DER NUMMER 17 MITARBEITEN.

Diese kann nun mittels Relationenalgebra folgendermaßen formalisiert werden:

$$\pi_{\text{NAME}} (\sigma_{\text{ANGNR = ANG\_PRO.ANGNR}} (\rho_{\text{ANG\_PRO.ANGNR} \leftarrow \text{ANGNR}} (\sigma_{\text{PNR = 17}} (\text{ANG\_PRO})) \times \text{ANGEST}))$$

Um das Verständnis zu erleichtern, gehen wir diesen Ausdruck nun Operation für Operation durch. Dabei gehen wir im Wesentlichen von innen nach außen vor. Zunächst werden aus der Relation ANG_PRO diejenigen Tupel herausgesucht, die Projekt Nummer 17 betreffen:

Selektion: $\sigma_{\text{PNR = 17}} (\text{ANG\_PRO})$

Wenn wir das Kreuzprodukt aus der durch die Selektion reduzierten Relation ANG_PRO mit der Relation ANGEST bilden, würde es zu einem Namenskonflikt kommen: Sowohl ANGEST als auch ANG_PRO enthalten das Attribut ANGNR[4]. Um das zu vermeiden, wird das Attribut ANGNR in ANG_PRO nach ANG_PRO.ANGNR umbenannt:

$$\rho_{\text{ANG\_PRO.ANGNR} \leftarrow \text{ANGNR}} (\sigma_{\text{PNR = 17}} (\text{ANG\_PRO}))$$

Nun kann das Kreuzprodukt gebildet werden …

Kreuzprodukt: ...×ANGEST

... und anschließend werden die zueinander passenden Kombinationen ausgewählt ...

Selektion: $\sigma_{\text{ANGNR = ANG\_PRO.ANGNR}} (\ldots)$

... und von diesem Zwischenergebnis schließlich nur die Namen ausgegeben:

Projektion: $\pi_{\text{NAME}} (\ldots)$

Das Ergebnis dieser Abfrage enthält die Namen 'MÜLLER' und 'SCHMIDT'.

In Beispiel 3.4 wird eine weitere Möglichkeit beschrieben, um das gleiche Ergebnis zu berechnen. Diese lautet als Ausdruck der Relationenalgebra folgendermaßen:

$$\pi_{\text{NAME}} (\sigma_{\text{PNR = 17}} (\sigma_{\text{ANGNR = ANG\_PRO.ANGNR}} (\rho_{\text{ANG\_PRO.ANGNR} \leftarrow \text{ANGNR}} (\text{ANG\_PRO})) \times \text{ANGEST}))$$

---

4 Wird - so wie in diesem Beispiel - der Name der Relation vor das Attribut gestellt, dann wird gesagt, dass das Attribut durch den Relationsnamen *qualifiziert* wird. Die meisten Namenskonflikte können durch Qualifikation vermieden werden. (Deshalb wird im Zusammenhang mit relationalen Abfragen und -sprachen oft die Qualifikation als automatisch vorausgesetzt.)

Im Vergleich zum ersten Ausdruck ist lediglich die Selektion bzgl. der Projektnummer weiter nach vorne gerückt, also in der Ausführungsreihenfolge weiter nach hinten, nämlich hinter die Ausführung des kartesischen Produktes. Solche Verschiebungen können z.B. bei der Optimierung von Abfragen sinnvoll sein, weil einige Operatoren der relationalen Algebra über verschiedene, nützliche algebraische Eigenschaften verfügen (siehe Abschnitt 3.4).

Ein Beispiel für die Vereinigungsoperation ist die Zusammenfassung der Angestellten und Kunden zu einer allgemeineren „Klasse“ von Personen. Dazu müssen zunächst die speziellen Attribute der Relationen ANGEST und KUNDE durch Projektion entfernt und die verschiedenen Bezeichnungen für die Arbeit der Angestellten und Kunden angeglichen werden. Dann können beide Tabellen mit der **Vereinigung**soperation zusammengefasst werden:

$$\pi_{\text{NAME, WOHNORT, BERUF}}(\text{ANGEST}) \cup$$
$$\pi_{\text{NAME, WOHNORT, BERUF}}(\rho_{\text{BERUF}\leftarrow\text{TÄTIG\_ALS}}(\text{KUNDE}))$$

Das Ergebnis enthält die Angestellten Müller, Winter und Seeler sowie die Kunden Klöterjahn, Leander und Osterloh. Aufgrund der Definition von Relationen als Mengen von Tupeln ist die Person mit dem Namen 'SCHMIDT' nur einmal vorhanden, obwohl sie in beiden Ausgangsrelationen, ANGEST und KUNDE, vorhanden ist.

Zusätzliche Operationen

Zusätzlich zu den vorgestellten Grundoperationen wurden verschiedene spezielle Operationen eingeführt, die zwar keine neuen Möglichkeiten bieten, aber die Formulierung vieler Abfragen sehr vereinfachen. Sie können alle aus den Grundoperationen abgeleitet werden. Wir werden hier den Verbund und den natürlichen Verbund sowie in Übung 3.3 einen weiteren Mengenoperator, die Schnittmenge, vorstellen:

Es ist sehr häufig notwendig, zwei Relationen bezüglich zweier Attribute mit gleichen Wertebereichen und gleicher oder ähnlicher Bedeutung miteinander zu verbinden. Wir hatten ein Beispiel kennengelernt, wo diese Verbindung so geschah: In Beispiel 3.5 wurden die Tupel der Relation ANGEST und ANG_PRO als kartesisches Produkt zusammengefasst, wobei schließlich nur die Kombinationen interessant waren, für die der Wert für ANGNR in beiden Tupeln derselbe war. Diese Art der Verknüpfung wird als Verbund bezeichnet.

Verbund/Join:

Verbund

Mit dem Verbund werden diejenigen Tupel aus zwei Relationen kombiniert, die bzgl. jeweils eines Feldes aus beiden Relationen einem vorgegebenen Vergleich genügen.

Seien $R(R_1, ..., R_n)$ und $S(S_1, ..., S_m)$ Relationen.

Sei $\{R_1, ..., R_n\} \cap \{S_1, ..., S_m\}$ die leere Menge,
$i \in \{1, ..., n\}$, $j \in \{1, ..., m\}$ mit $\text{dom}(R_i) = \text{dom}(S_j)$ und
θ ein zweistelliger Operator auf $\text{dom}(R_i)$, der einen Wahrheitswert liefert.

Dann heißt $V = R \bowtie_{Ri\,\theta\,Sj} S$ der *Verbund (join) von R und S bzgl. der Verbundbedingung* $R_i\ \theta\ S_j$ und ist folgendermaßen definiert:

$$V = \sigma_{Ri\,\theta\,Sj}(R \times S)$$

equi-join

Der Verbund mit θ = '=' (Gleichheit von Werten) wird häufig als *equi-join* bezeichnet.

Beispiel 3.6:

R	A	B	C
	1	2	3
	6	5	6
	8	8	9
	1	1	7

S	D	E
	9	3
	5	7

$R \bowtie_{A>E} S$	A	B	C	D	E
	6	5	6	9	3
	8	8	9	9	3
	8	8	9	5	7

$R \bowtie_{C=E} S$	A	B	C	D	E
	1	2	3	9	3
	1	1	7	5	7

Beispiel 3.7:

Die Abfrage aus Beispiel 3.5 kann mit der Verbundoperation so formuliert werden:

$$\pi_{\text{NAME}} (\rho_{\text{ANG\_PRO.ANGNR} \leftarrow \text{ANGNR}} (\sigma_{\text{PNR} = 17} (\text{ANG\_PRO})) \bowtie_{\text{ANG\_PRO.ANGNR} = \text{ANGNR}} \text{ANGEST})$$

Übung 3.1:

Definieren Sie den Verbund ohne Verwendung der Grundoperationen.

(Hinweis: Orientieren Sie sich an der Definition des kartesischen Produktes und erweitern Sie diese um die Verbundbedingung, wie sie in der Definition des Verbunds beschrieben ist.)

Augenfällig ist der Zweck des equi-joins bzgl. Attributen mit der gleichen Bedeutung, die dann sinnvollerweise oft auch die gleichen Namen tragen. Inhaltlich zusammenhängende Tupel werden so miteinander kombiniert. Diese Operation bzgl. aller gleichnamigen Attribute zweier Tabellen wird als natürlicher Verbund bezeichnet. Wie wir später im Zusammenhang mit dem Schemaentwurf bei relationalen Systemen sehen werden, spielt der natürliche Verbund eine besondere Rolle.

Natürlicher Verbund/Natural Join:

Natürlicher Verbund

Mit dem natürlichen Verbund werden diejenigen Tupel aus zwei Relationen so kombiniert, dass jeweils die Werte der Attribute gleichen Namens aus beiden Relationen übereinstimmen. Im Ergebnis sind diese Attribute nur einmal vorhanden.

Seien $R(R_1, ..., R_n)$ und $S(S_1, ..., S_m)$ Relationen.

Sei $(R_{i1}, ..., R_{ik}) = (S_{j1}, ..., S_{jk})$ $(k > 0)$ die gleichnamigen Attribute von R und S,
$\{S_{l1}, ..., S_{l\,m-k}\} = \{S_1, ..., S_m\} \setminus \{ S_{j1}, ..., S_{jk} \}$ die exklusiven Attribute von S.

Dann heißt $V = R \bowtie S$ der *natürliche Verbund (natural join) von R und S* und ist folgendermaßen definiert:

$$V = \pi_{R1, ..., Rn, Sl1, ..., Sl\,m-k} (\sigma_{Ri1=S.Sj1 \wedge ... \wedge Rik=S.Sjk} (\rho_{S.Sj1 \leftarrow Sj1, ..., S.Sjk \leftarrow Sjk} (S) \times R))$$

Beispiel 3.8: (natürlicher Verbund)

Relation R

A	B	C
a	b	c
b	b	c
c	b	d
d	a	b

Relation S

B	C	D
b	c	x
b	c	y
a	b	z

$R \bowtie S$

A	B	C	D
a	b	c	x
a	b	c	y
b	b	c	x
b	b	c	y
d	a	b	z

Die Relationen R und S haben die Attributnamen B und C gemeinsam. Für jedes Tupel von R wird geprüft, welche Tupel von S in den Attributen B und C mit ihm übereinstimmen. Beispielsweise stimmt (a, b, c) in diesem Sinne überein mit (b, c, x) und (b, c, y), wir erhalten die Tupel (a, b, c, x) und (a, b, c, y) für $R \bowtie S$, (c, b, d) stimmt zwar in B, nicht aber zugleich in C mit Tupeln von S überein, so dass also kein Tupel entsteht, das mit (c, b, d) beginnt.

Beispiel 3.9:

Die Abfrage aus Beispiel 3.5 kann mit dem natürlichen Verbund erheblich einfacher formuliert werden:

$$\pi_{NAME} (\sigma_{PNR = 17} (ANG\_PRO) \bowtie ANGEST)$$

Die zweite Variante, bei der die Selektion bzgl. der Projektnummer erst nach dem Join ausgeführt wird, lautet entsprechend:

$$\pi_{NAME} (\sigma_{PNR = 17} (ANG\_PRO \bowtie ANGEST))$$

Übung 3.2:

Definieren Sie den natürlichen Verbund ohne Verwendung der Grundoperationen.

(Hinweis: Ersetzen Sie in der o.a. Definition des natürlichen Verbundes nur die letzten beiden Zeilen durch tupel- bzw. mengenorientierte Ausdrücke. Orientieren Sie sich dabei an der Lösung zur Übung 3.1)

Übung 3.3:

Die Schnittmenge zweier Relationen enthält alle Tupel, die in beiden Relationen enthalten sind.

Definieren Sie den relationalen Operator zur Bildung der Schnittmenge zweier Relationen mit Hilfe der Grundoperationen.

Hinweis: Die Schnittmenge lässt sich durch geschickten Einsatz (2×) der Differenz darstellen.

Natürlich kann die formal anmutende Notation durch eine „benutzerfreundlichere“ Notation ersetzt werden, etwa indem man erklärende Schlüsselworte wie JOIN, PROJECT usw. verwendet. Ferner kann man eine Abfrage auch in eine Folge von Einzelschritten zerlegen und auf diese Weise Schachtelung vermeiden. Diese Möglichkeit kommt dem ungeübten Benutzer entgegen.

Beispiel 3.10:

So könnte man beispielsweise die zweite Variante der Abfrage aus Beispiel 3.9 wie folgt schreiben:

```
NATJOIN ANGEST, ANG.PRO BY ANGNR GIVING R1
SELECT R1 WHERE PNR = 17 GIVING R2
PROJECT R2 OVER NAME GIVING RESULT.
```

Übungen !

Übung 3.4:

Gegeben seien die Relationen ANGEST, PROJECT und ANG_PRO wie in Beispiel 3.4. Versuchen Sie, für die folgenden Ausdrücke in Relationenalgebra äquivalente umgangssprachliche Formulierungen zu finden. D.h., welche Ergebnisse sollen diese Abfragen liefern?

a) $\pi_{\text{NAME, PNR}}$ (ANG_PRO $\bowtie$ ANGEST)

b) $\pi_{\text{NAME, BERUF}}$ (PROJEKT $\bowtie_{\text{P\_LEITER = ANGNR}}$ ANGEST)

c) $\pi_{\text{NAME, P\_NAME, PROZ\_ARB}}$
($\sigma_{\text{PNR = ANG\_PRO.PNR}}$ (PROJEKT
$\bowtie_{\text{P\_LEITER = ANGNR}}$ ANGEST
$\bowtie \rho_{\text{ANG\_PRO.PNR'} \leftarrow \text{PNR}}$ (ANG_PRO))

d) Eine etwas kompliziertere Abfrage:

$$
\begin{aligned}
&\pi_{\text{NAME}} \\
&(\quad \sigma_{\text{P1.PNR} <> \text{P2.PNR}} \\
&\quad (\quad \rho_{\text{P1.PNR} \leftarrow \text{PNR}} (\pi_{\text{PNR, P\_LEITER}} (\text{PROJEKT})) \\
&\qquad\qquad \bowtie \\
&\qquad\quad \rho_{\text{P2.PNR} \leftarrow \text{PNR}} (\pi_{\text{PNR, P\_LEITER}} (\text{PROJEKT})) \\
&\quad) \\
&\quad \bowtie_{\text{P\_LEITER=ANGNR}} \text{ANGEST} \\
&)
\end{aligned}
$$

Übung 3.5:

Formulieren Sie, wiederum bezogen auf die Relationen ANGEST, PROJEKT und ANG_PRO, folgende Abfragen in Relationenalgebra:

a) Gib die Nummern aller Abteilungen aus, die an Projekt 23 beteiligt sind.

b) Eine etwas kompliziertere Abfrage:
Gib die Namen und Angestelltennummern aller Angestellten aus, die an einem der Projekte mitarbeiten, an denen auch Frau Kaufmann (der Name 'KAUFMANN' sei eindeutig) mitarbeitet.

3.2.3 Relationenkalkül

Bei Sprachen auf der Basis des Relationenkalküls gibt der Benutzer die *Definition einer Relation* an, die aus den vorhandenen Relationen seines externen Schemas abgeleitet werden soll. Er sagt nichts über die anzuwendenden Operationen aus. Der Relationenkalkül lässt sich unterteilen in den Werte-orientierten Relationenkalkül und den Tupel-orientierten Relationenkalkül. Der Unterschied besteht hauptsächlich darin, dass in dem letzteren Variablen ganze Tupel bezeichnen, während in dem Werte-orientierten Relationenkalkül Variablen einzelne Komponenten eines Tupels bezeichnen. In diesem Abschnitt werden wir näher auf den Tupel-orientierten Relationenkalkül eingehen; weitere Informationen zum Werte-orientierten Relationenkalkül, das auch als Bereichskalkül bezeichnet wird, finden sich z.B. in /ULL89/.

Der Relationenkalkül ist nichts anderes als eine formale Sprache zur Definition einer neuen Relation in den Begriffen schon gegebener Relationen. Beispielsweise formulieren wir die Abfrage

FINDE DIE WOHNORTE ALLER ANGESTELLTEN,
DIE PROGRAMMIERER SIND

so:

{(ANGEST.WOHNORT) | ANGEST.BERUF = 'PROGRAMMIERER'}

Die geschweiften Klammern geben an, dass wir eine Menge von Tupeln (also eine Relation) suchen. Vor dem Trennstrich '|' steht das Relationenschema der

gewünschten Relation; nach dem Trennstrich steht eine Bedingung, das *Prädikat*, das festlegt, welche Tupel die gewünschte Relation umfassen soll.

Ausdruck im Relationenkalkül

Ein Ausdruck im Relationenkalkül hat also die Form {t | q}, wobei t eine Liste von Attributnamen (Schema der gewünschten Relation) und q ein Prädikat (Qualifikationsteil, der die gewünschten Tupel für t spezifiziert) ist. Wo notwendig, müssen Attributnamen mit den Relationennamen qualifiziert werden.

Prädikat

Das *Prädikat* q ist ein logischer Ausdruck von beliebiger Komplexität, der in üblicher Weise aufgebaut ist aus Attributnamen, Konstanten, Vergleichsoperatoren ('=', '≠', '>', ...), Booleschen Operatoren ('∧', '∨' bzw. AND, OR, NOT), Existenzquantoren '∃' („es existiert“), Allquantoren '∀' („für alle“) und Tupelvariablen. Eine *Tupelvariable* ist eine Variable, die ein Tupel einer Relation bezeichnet. Eine Tupelvariable ist in einem Ausdruck *gebunden*, wenn sie durch einen '∃' oder '∀' Quantor eingeführt wird, ansonsten ist sie *frei*. Ist q ein logischer Ausdruck, in dem x vorkommt und nicht gebunden ist, so sind (∃x)q und(∀x)q Ausdrücke, in denen x gebunden ist. (∃x)q besagt, dass ein Wert von x existiert, so dass q durch Einsetzen dieses Wertes an Stelle von x wahr wird. (∀x)q besagt, dass für jeden Wert von x der Ausdruck q wahr wird.

Tupelvariable

freie und gebundene Variable

Man kann sich die Begriffe der freien und gebundenen Variablen etwas veranschaulichen mit folgendem Vergleich: freie Variablen entsprechen globalen Variablen, die in einem Programm außerhalb der betrachteten Prozedur deklariert sind; gebundene Variablen entsprechen lokalen Variablen. Die Quantifizierung entspricht der Variablendeklaration.

Beispiel 3.11:

Gegeben sei die Abfrage:

FINDE DEN (ODER DIE) PROGRAMMIERER
MIT DEM HÖCHSTEN GEHALT.

Wir formulieren diese Abfrage im Relationenkalkül wie folgt:

```
RANGE ANGEST X
{(ANGEST.NAME) | ANGEST.BERUF = 'PROGRAMMIERER' ∧
                    ¬ ∃ X (X.BERUF = 'PROGRAMMIERER' ∧
                          X.GEHALT > ANGEST.GEHALT)}
```

Die Abarbeitung der Abfrage kann man sich so vorstellen: wir betrachten nacheinander jedes Tupel in ANGEST mit Beruf = 'Programmierer' und prüfen, ob es irgendein anderes Tupel mit diesem Beruf in ANGEST gibt, das ein größeres Gehalt ausweist. Wenn nicht, ist eines der gesuchten Tupel gefunden. Wir drücken dies kürzer aus, indem wir sagen, es darf kein anderes Tupel mit Wert 'PROGRAMMIERER' mit größerem Gehalt existieren.

Während durch die Nennung von ANGEST im linken Teil der Mengenklammer implizit eine Tupelvariable für diese Relation definiert ist, benöti-

gen wir zusätzlich eine weitere Tupelvariable X vom Typ ANGEST und definieren sie mit RANGE ANGEST X.

¬ ∃ X (X.GEHALT > ANGEST.GEHALT) bedeutet dann gerade, was wir wollen: es existiert nicht ein Tupel X, dessen Gehalt größer ist als der Wert ANGEST.GEHALT des gerade betrachteten Tupels.

In (X.GEHALT > ANGEST.GEHALT) ist X frei, in ∃ X(...) ist X gebunden.

Ist { t | q } der Ausdruck des Relationenkalküls, so muss Folgendes gelten:
Für jeden Relationennamen, der in q aber nicht in t vorkommt, muss eine Tupelvariable explizit definiert sein, die durch ∃ oder ∀ gebunden ist. Mit anderen Worten: q darf nur solche freien Variablen enthalten, die auch in t vorkommen.

Dies ist leicht einzusehen: würde eine Komponente einer Relation T in q frei auftreten, d.h. ohne dass eine Komponente von T auch in t auftritt, so wäre völlig unklar, was dies zu bedeuten hätte.

Beispiel 3.12:

{R.X | T.Y = R.Z} lässt unklar, ob alle Werte Y der Relation T oder irgendein Wert Y der Relation T gleich R.Z sein soll.

{(R.X, T.V) | T.Y = R.Z} ist hingegen eindeutig: wir wollen (R.X, T.V) für diejenigen Tupelkombinationen R und T, für die T.Y = R.Z (In der Relationenalgebra kann diese Abfrage mit einem Verbund ausgedrückt werden: $\pi_{X, V}(R \bowtie_{Y=Z} T)$.

Wir wollen uns das Arbeiten mit einer Kalkül-orientierten Sprache anhand einiger Beispiele klarmachen.

Beispiel 3.13: (Einfaches Retrieval)

FINDE DIE NUMMERN ALLER ANGESTELLTEN DER
ABTEILUNG 6, DIE IN DORTMUND WOHNEN

{(ANGEST.ANGNR) | ANGEST.ABTNR = 6 ∧
ANGEST.WOHNORT = 'DORTMUND'}

Beispiel 3.14: (Retrieval mit Existenz-Quantor)

(vgl. hierzu auch das ausführlich diskutierte Beispiel 3.11)

FINDE DIE NAMEN DER ANGESTELLTEN, DIE AN EINEM PROJEKT MITARBEITEN

RANGE ANG_PRO X
{(ANGEST.NAME) | ∃ X(X.ANGNR = ANGEST.ANGNR)}

Im Unterschied zu Beispiel 3.11 wird hier auf zwei Relationen Bezug genommen: ANGEST und ANG_PRO. Der Qualifikationsteil besagt, dass ein Tupel in der Relation ANG_PRO existieren muss, das dieselbe Angestelltennummer aufweist wie das gerade betrachtete Tupel in ANGEST.

Beispiel 3.15: (Retrieval mit mehreren Existenzquantoren I)

FINDE DIE NAMEN DER ANGESTELLTEN, DIE AN EINEM PROJEKT MITARBEITEN, AN DEM AUCH DER ANGESTELLTE MIT DER NUMMER 10 MITARBEITET

RANGE ANG_PRO X
RANGE ANG_PRO Y
$\{(\text{ANGEST.NAME}) \mid \exists X (X.\text{ANGNR} = \text{ANGEST.ANGNR} \wedge \exists Y(Y.\text{PNR} = X.\text{PNR} \wedge Y.\text{ANGNR} = 10))\}$

Diese Formulierung ist auf Anhieb nicht leicht zu verstehen. Gehen wir schrittweise vor:

$\exists X (...$: Zunächst muss jeder der gesuchten Angestellten ja an einem Projekt mitarbeiten, wofür ein Tupel in ANG_PRO existieren muss.

(X.ANGNR = ANGEST.ANGNR): Dies wird mit dem entsprechenden Tupel aus ANGEST in Verbindung gebracht, um im Ergebnis den Namen zu erhalten.

$... \wedge \exists Y(Y.\text{PNR} = X.\text{PNR} \wedge Y.\text{ANGNR} = 10)$:
Dann muss in ANG_PRO ein Tupel existieren, das dieselbe Projektnummer aufweist und dessen Angestelltennummer 10 ist.

Übung 3.6:

Überlegen Sie, ob mit dieser Formulierung der Abfrage der Name des Angestellten Nr. 10 mit ausgegeben wird oder nicht. Wie müsste man die Alternative formulieren?

Beispiel 3.16: (Retrieval mit mehreren Existenzquantoren II)

FINDE DIE NAMEN DER ANGESTELLTEN, DIE AN EINEM PROJEKT UNTER DER LEITUNG DES ANGESTELLTEN MIT DER NR. 10 MITARBEITEN.

RANGE PROJEKT X
RANGE ANG_PRO Y
$\{(\text{ANGEST.NAME}) \mid \exists Y(Y.\text{ANGNR} = \text{ANGEST.ANGNR} \wedge \exists X(X.\text{PNR} = Y.\text{PNR} \wedge X.\text{P\_LEITER} = 10))\}$

Für die gesuchten Tupel aus ANGEST muss gelten, dass ein Tupel Y in ANG_PRO existiert, das die gleiche ANGNR aufweist und für das außerdem ein Tupel X in PROJEKT mit derselben Projektnummer existieren muss, in dem P_LEITER = 10 gilt.

In einer realen Abfragesprache könnte man diese Abfrage natürlich auch in einfachen überschaubaren Schritten formulieren:

W: {(PROJEKT.PNR) | PROJEKT.P_LEITER = 10}
RANGE W X
W1: {(ANG_PRO.ANGNR) | ∃X(X.PNR = ANG_PRO.PNR)}
RANGE W1 Y
{(ANGEST.NAME) | ∃Y(Y.ANGNR = ANGEST.ANGNR)}

W ist ein Name für die im ersten Schritt erzeugte Relation, auf die im zweiten Schritt Bezug genommen wird (X ist Tupelvariable auf W). Die Abfrage beginnt also jetzt damit, die Projekte des Angestellten Nr. 10 zu bestimmen, hierzu dann die Mitarbeiter-Nummern aus ANG_PRO, hierzu dann die Namen aus ANGEST.

Durch diese Zergliederung in Einzelschritte erhält eine Abfrage im Relationenkalkül jedoch zusätzliche Struktur: Die Ausführungsreihenfolge wird teilweise festgelegt – der deklarative Charakter des Relationenkalküls, ein wesentlicher Unterschied zur Relationenalgebra, geht dabei verloren.

Beispiel 3.17: (Retrieval mit Allquantor)

FINDE DIE NAMEN DER ANGESTELLTEN, DIE AN KEINEM PROJEKT MITARBEITEN.

RANGE ANG_PRO X
{(ANGEST.NAME) | ∀ X(X.ANGNR ≠ ANGEST.ANGNR)}

Zur Veranschaulichung: Wir betrachten jedes Tupel in ANGEST. Ein Tupel ist dann eines der gesuchten Tupel, wenn in ANG_PRO alle Tupel Angestellten-Nummern aufweisen, die von ANGEST.ANGNR verschieden sind (Dann nämlich arbeitet der Angestellte an keinem Projekt mit.).

Übung 3.7:

Formulieren Sie die Abfrage aus Beispiel 3.17 mit Hilfe von Existenzquantoren.

Beispiel 3.18:

Bei der Festlegung des Ergebnisschemas im linken Teil der Mengenklammer eines Ausdrucks im Relationenkalkül müssen die Attributnamen nicht immer mit den Namen bestehender Relationen qualifiziert werden. So wird die Vereinigung der Daten von Angestellten und Kunden – wie am Ende von Beispiel 3.5 für die Relationenalgebra dargestellt – folgendermaßen ausgedrückt:

```
RANGE ANGEST X
RANGE KUNDE Y
{ (NAME, ORT, BERUF) |
        ∃ X (X.NAME=NAME ∧ X.WOHNORT=ORT ∧
                                        X.BERUF=BERUF) ∨
        ∃ Y (Y.NAME=NAME ∧ Y.WOHNORT=ORT ∧
                                        Y.TÄTIG_ALS=BERUF)}
```

Im Qualifikationsteil muss dann sichergestellt werden, dass die Attribute für jedes Ergebnistupel einen wohldefinierten Wert annehmen, i.d.R. einen Wert aus einer bestehenden Relation.

Übung 3.8:

Formulieren Sie die in Übung 3.5 angegebenen Abfragen im Relationenkalkül.

Vollständigkeit einer Abfragesprache

relational vollständig

Der Relationenkalkül wird als Maßstab für die Beurteilung der Mächtigkeit relationaler Abfragesprachen angesehen. Eine Sprache, die nicht mindestens die Ausdruckskraft des Relationenkalküls besitzt, wird als unvollständig erachtet. Man bezeichnet eine Abfragesprache als *relational vollständig* (complete), wenn mit ihr alles ausgedrückt werden kann, was im Relationenkalkül ausgedrückt werden kann.

Relationenkalkül (sowohl Tupel-orientiert, als auch Werte-orientiert) und Relationenalgebra besitzen dieselbe Sprachmächtigkeit, d.h. es gibt für jeden Ausdruck im Relationenkalkül einen Ausdruck in der Relationenalgebra, der dieselbe Relation erzeugt, und umgekehrt.

Erweiterungen

Aggregierungsfunktionen

Für die praktische Anwendung sind Sprachmöglichkeiten wichtig, die im „reinen" Kalkül und in der „reinen" Algebra nicht vorhanden sind. Hierzu gehören Möglichkeiten zur sortierten Ausgabe, arithmetische Operationen etwa bei Vergleichen ($A < B + 10$) und vor allem Aggregierungsfunktionen wie Zahl der Tupel einer Relation, Durchschnitt, Summe, Maximalwert, Minimalwert, usw. Diese Funktionen sind wichtig, um Abfragen der Art

> WIEVIELE ANGESTELLTE MIT EINER BESTIMMTEN EIGENSCHAFT GIBT ES?

formulieren zu können. Derartige Erweiterungen werden wir im nächsten Abschnitt im Zusammenhang mit SQL kennenlernen.

Update

Neben dem Retrieval muss eine Abfrage-, oder deutlicher: eine Datenmanipulationssprache natürlich Möglichkeiten zur Veränderung (Update) von Relationen

bieten. Dies kann so geschehen, dass die zu verändernden Tupel mittels der Abfragesprache gelesen werden, die Änderungen dann mittels der *Wirtssprache* (COBOL, PL/I, C, etc.) durchgeführt werden und schließlich die veränderten Tupel mittels eines Befehls der Abfragesprache zurückgeschrieben werden.

Eine andere Möglichkeit besteht darin, dass die Abfragesprache selbst die Sprachkonstrukte bietet, mit denen Änderungen möglich sind. In diesem Fall wird also keine Wirtssprache benötigt (vgl. Beispiel 3.32 bis Beispiel 3.34).

Auf Updates gehen wir erst im nächsten Abschnitt für die konkrete Sprache SQL ein, da weder Algebra noch Kalkül Update-Konzepte enthalten oder in der Praxis eine Rolle spielen.

3.3 SQL – eine relationale Abfragesprache

In diesem Abschnitt werden wir die Abfragesprache mit der weitesten Verbreitung vorstellen: SQL.

3.3.1 Grundlegende Sprachelemente von SQL

Nachdem E.F. Codd 1970 den bahnbrechenden Artikel „A Relational Model of Data for Large Shared Data Banks“ /COD70/ veröffentlicht hatte, begann man im IBM San Jose Research Laboratory über eine Sprache zu forschen, die dazu geeignet wäre dieses Modell zu implementieren. Die Sprache wurde SEQUEL genannt, ein Akronym für **S**tructured **E**nglish **QUE**ry **L**anguage.

Innerhalb eines Forschungsprojektes mit Namen System R entwickelte man ein Datenbanksystem desselben Namens zu Forschungszwecken. SEQUEL war das Programmierinterface zu diesem Datenbanksystem. Zwischen 1974 und 1975 wurde diese Sprache in einem Prototypen implementiert, der wiederum den Namen SEQUEL erhielt. In den nächsten Jahren wurde eine überarbeitete Version SEQUEL/2 entwickelt Diese Version der Sprache wurde in SQL umbenannt. [5]

Das Projekt System R lief in der Zeit von 1971 bis 1979 und mündete in einem verteilten Datenbankprojekt mit dem Namen System R* (englisch ausgesprochen „are star“).

In den späten 70er Jahren wurde allgemein bekannt, dass IBM dabei war, ein kommerzielles DBMS zu entwickeln, das SQL als Sprache benutzen sollte. Als Konsequenz daraus wurde die Konkurrenz aktiv. Tatsächlich schlug eine kleine Firma mit dem Namen „Relational Software Inc.“ die große IBM in dem Rennen um die erste Markteinführung eines solchen Systems. Diese Firma heißt übrigens inzwischen ORACLE und ist seitdem geringfügig gewachsen. IBM brachte 1981 sein erstes relationales DBMS auf den Markt, damals SQL/DS genannt. Die erste Version des bekannten DB2 erschien 1983.

---

5 Aus dieser Umbenennung rührt der Glaube her, SQL stünde für 'Structured Query Language' und würde "sequel" ausgesprochen. Laut Standard ist SQL jedoch kein Akronym.

Einen rechtlichen Status als Standard erhielt SQL 1986 durch die Veröffentlichung eines ANSI Standards der Sprache. Bekannt wurde dieser Standard unter dem Kürzel SQL-86. Eine Überarbeitung erschien 1989 (SQL-89). Größere Änderungen erfolgten 1992 (SQL-92). Viele Schwachstellen von SQL-86 und SQL-89 wurden in diesem Standard beseitigt. SQL:1999, die neueste Revision geht weit über die bisherigen Standards hinaus und ist eine Annäherung an die Möglichkeiten, die neue Datenbanksysteme bieten (können) und die Bedürfnissen ihrer Benutzer.

In der Zeit von 1992 bis 1999 wurden einzelne Aspekte des zukünftigen Standards SQL:1999 gesondert veröffentlicht. Die erste dieser Veröffentlichungen war der Standard zu SQL/CLI (Call Level Interface), bekannt als CLI-95. Wahrscheinlich die bekannteste Implementierung dieses Standards ist ODBC, Open Database Connectivity.

1996 wurde SQL/PSM (Persistent Stored Modules) veröffentlicht. PSM-96, wie der Standard genannt wurde, legt die Syntax der prozeduralen Logik fest, die auf Serverseite in DBMS implementiert wird. Dieser Standard war nötig geworden, weil viele kommerzielle DBMS „Stored Procedures“ anboten, deren Syntax allerdings voneinander abwich.

1998 erfolgte die Veröffentlichung von SQL/OLB (Object Language Bindings), der beschreibt, wie SQL Statements in JAVA eingebettet werden können. Er basiert auf dem JDBC Modell und wurde bekannt unter dem Namen OLB-98.

1999 wurde der neue Sprachstandard für SQL angenommen und als SQL:1999 veröffentlicht.

Der Bindestrich ist verschwunden und die Jahreszahl wurde um das Jahrhundert und Jahrtausend erweitert. Die Verwendung der vierstelligen Jahreszahl entspringt der Y2K-Umstellung, die Änderung des Bindestriches in einen Doppelpunkt ist von weitreichenderer Bedeutung: Sie bezieht sich auf die Normen der ISO, die die Namen der Standards von den Jahreszahlen der Veröffentlichung durch einen Doppelpunkt trennt, im Gegensatz zur ANSI, die hierzu eben den Bindestrich benutzt. Dies deutet auf die zunehmende Internationalisierung des Standardisierungsprozesses hin und die Marginalisierung der ANSI Norm außerhalb der USA durch das Aufgehen ihrer Normung in der ISO.[6]

SQL basiert auf dem relationalen Modell, ist aber keineswegs eine perfekte Abbildung desselben auf eine Sprache. SQL weicht in einigen Aspekten vom relationalen Modell und relational vollständigen Sprachen, wie in den vorangegangenen Abschnitten erklärt, ab. Die meisten Unterschiede stellen Erweiterungen dar, auf die wir später noch eingehen. Der wichtigste Unterschied betrifft allerdings

6 Eine genaue Darstellung der Geschichte der SQL-Normung (die für Entwickler durchaus interessant sein mag, wenn Sie gezwungen sind mit verschiedenen Implementierungen parallel zu arbeiten) finden Sie in /MELT02/ im Anhang F. Die Darstellung in diesem Kapitel lehnt sich an die in /MELT02/ an.

das grundlegende Objekt relationaler Datenbanken und des relationalen Modells. SQL arbeitet mit Tabellen, die Duplikate von Zeilen enthalten können. Dies entspräche Tupeln, die in allen Attributen übereinstimmen, was in Relationen nicht möglich ist.

Um den Unterschieden zwischen der Sprachregelung in SQL und dem relationalen Modell Rechnung zu tragen, werden wir in diesem Kapitel folgendermaßen unterscheiden:

Relationales Modell	**SQL**	**(SQL englisch)**
Relation	Tabelle	(Table)
Tupel	Zeile	(Row)
Attribut	Spalte	(Column)

An den Stellen, an denen wir uns explizit auf das relationale Modell als Grundlage für SQL beziehen, werden wir die Sprachregelung des Modells anwenden und dort, wo es um die Umsetzung in SQL geht, diejenige, die im SQL:1999 Standard benutzt wird.

Neben den Aggregierungsfunktionen, die wir später in diesem Kapitel behandeln werden, bietet SQL noch eine wichtige Ergänzung gegenüber dem vorgestellten Relationenmodell und „bloß“ relational vollständigen Sprachen an: sogenannte NULL-Werte. Prinzipiell kann jedes Attribut für ein Tupel den Wert NULL annehmen, was (ohne weitere Vereinbarungen) bedeutet, dass dieser Wert zwar vorgesehen, aber (noch) nicht festgelegt ist. Die Definition verschiedener Operatoren wurde für NULL-Werte verändert. Es ergibt sich eine ternäre Logik mit den Werte WAHR (TRUE), FALSCH (FALSE) und UNBEKANNT (UNKNOWN) bzw. NULL.

Wir werden mit den folgenden Beispielen einen Einblick in die umfassenden Möglichkeiten der SELECT-Anweisung geben und daran anschließend einige weitere DML- und DDL-Anweisungen in SQL vorstellen.[7]

Die SELECT-Anweisung

Eine SQL-Abfrage (*Query*) hat folgende Grundstruktur:

SELECT A_1, ..., A_n
FROM R_1, ..., R_n
WHERE *Prädikat*(R_1, ..., R_n).

Die Bedeutung einer solchen Query kann man sich so veranschaulichen: selektiere aus dem Kreuzprodukt der Relationen R_1, ..., R_n alle Tupel, die die Bedingung

7 Traditionell wird SQL häufig in DML (Data Manipulation Language) und DDL (Data Definition Language) aufgeteilt. Diese Aufteilung findet sich so im Standard nicht wieder. Laut Standard müsste man eher von SDL (Schema Definition Language), SML (Schema Manipulation Language) und DML sprechen.

Prädikat(R_1, ..., R_n) erfüllen, und projiziere die so entstehende Relation auf die Attribute A_1, ..., A_n.

Dabei ist A_i ein Attribut einer der Relationen R_1, ..., R_n. *Prädikat(R_1, ..., R_n)* ist eine Bedingung, die Attributwerte mit Konstanten oder zwei Attributwerte eines Tupels miteinander vergleicht oder andere Operatoren auf die Attributwerte anwendet; solche Teilbedingungen können durch die Booleschen Operatoren $\wedge$ (AND) und $\vee$ (OR) miteinander verknüpft und mit $\neg$ (NOT) negiert werden. Als Operanden können in den Prädikaten auch Mengen auftreten, insbesondere Ergebnismengen, die über andere SQL-Abfragen konstruiert werden.

Beispiel 3.19: (Gerüst der SELECT-Anweisung)

FINDE DIE BERUFE DER ANGESTELLTEN IN ABTEILUNG 3.

```
SELECT BERUF
FROM ANGEST
WHERE ABTNR = 3
```

Hier werden in der Tabelle ANGEST die Tupel mit ABTNR = 3 ermittelt und von diesen dann die Werte des Attributs BERUF ausgegeben.

Duplikate in SQL

Die Ergebnismenge des obigen Beispiels kann doppelte Elemente enthalten. Zum Aussortieren doppelter Zeilen muss in der SELECT-Klausel das Schlüsselwort DISTINCT angegeben werden. Der Grund hierfür liegt darin, dass es zusätzlichen Rechenaufwand kostet, aus jeder Ergebnistabelle Duplikate zu eliminieren, während viele praktische Anwendungen (insbesondere aber auch die Berechnung von Zwischenergebnissen) diese Anforderung gar nicht stellen.

Beispiel 3.20: (Aussortieren doppelter Tupel)

Die Abfrage:

FINDE ALLE WOHNORTE DER ANGESTELLTEN.

... lautet in SQL:

```
SELECT DISTINCT WOHNORT
FROM ANGEST
```

Die WHERE-Klausel kann hier entfallen, da lediglich die Projektion auf die Spalte WOHNORT durchgeführt wird und dazu keine Bedingung an die einzelnen Zeilen der Tabelle ANGEST gestellt werden muss.

Es soll jedoch in diesem Beispiel um den Einsatz des Schlüsselwortes DISTINCT gehen: Das Ergebnis dieser Abfrage enthält erwartungsgemäß drei Zeilen. Ohne die Angabe von DISTINCT jedoch wäre die Zeile (Karlsruhe) zweimal enthalten gewesen, also insgesamt vier Zeilen im Ergebnis (vgl. Beispieldaten aus Beispiel 3.4):

mit DISTINCT
Hagen
Karlsruhe
Marburg

ohne DISTINCT
Karlsruhe
Hagen
Marburg
Karlsruhe

Das zweifache Auftauchen von „Karlsruhe“ widerspricht der Frage nach allen Wohnorten, die Query ohne DISTINCT entspricht also auch semantisch nicht der Anforderung.

Um Beziehungen zwischen mehreren Relationen zu betrachten, können in der FROM-Klausel der Abfrage mehrere Tabellen angegeben werden. Über diese Möglichkeit werden in SQL JOINS realisiert. In der WHERE-Klausel wird deren Verknüpfung untereinander bestimmt.

Beispiel 3.21: (Einfache SELECT-Anweisung I)

SELECT ...
FROM ...
WHERE ...

FINDE FÜR JEDES PROJEKT DIE PROJEKTNUMMER UND DEN NAMEN DES PROJEKTLEITERS

```
SELECT PNR, NAME
FROM PROJEKT, ANGEST
WHERE PROJEKT.P_LEITER = ANGEST.ANGNR
```

In diesem Beispiel werden Werte aus verschiedenen Tabellen verlangt. Die angegebene Anweisung wird folgendermaßen ausgeführt:

- Erst Bildung eines Kreuzproduktes der Tabellen PROJEKT und ANGEST,
- Auswertung der WHERE Bedingung und Selektion der Zeilen, die die Bedingung erfüllen,
- dann Projektion auf die in der SELECT-Klausel angegebenen Spalten.

Man beachte bei diesem Beispiel, dass man die Spalten nicht mit den Namen der Tabellen qualifizieren muss, weil die Namen der Spalten auch bezogen auf zwei Tabellen eindeutig sind. Es ist jedoch sinnvoll, die Namen der Tabellen immer mitzuführen.

Beispiel 3.22: (Einfache SELECT-Anweisung II)

Erinnern wir uns an die letzte Abfrage aus **Beispiel 3.4**:
(siehe auch **Beispiel 3.9**)

FINDE DIE NAMEN ALLER ANGESTELLTEN, DIE AM PROJEKT MIT DER NUMMER 17 MITARBEITEN

Als SQL-Abfrage lautet sie:

```
SELECT ANGEST.NAME
FROM ANG_PRO, ANGEST
WHERE  ANG_PRO.PNR = 17 AND
          ANG_PRO.ANGNR = ANGEST.ANGNR
```

In der Tabelle, die sich nach Bildung des JOINS aus ANGEST und ANG_PRO ergibt, kommt die Spalte ANGNR zweimal vor, weil sie sowohl in ANGEST als auch in ANG_PRO enthalten ist. In dem WHERE-Teil der Abfrage muss daher jeweils der Tabellenname vorangestellt werden, um das Attribut eindeutig zu qualifizieren.

Tupelvariablen in SQL

In komplizierteren Abfragen wird es in SQL unter Umständen notwendig, *Tupelvariablen* (im Standard: 'Correlation Names') einzuführen. Tupelvariablen sind vor allem dann nötig, wenn in einer Abfrage mehrere verschiedene Tupel derselben Relation gleichzeitig betrachtet werden sollen. Oder anders ausgedrückt: Tupelvariablen sind immer dann erforderlich, wenn der JOIN einer Tabelle mit sich selbst gebildet werden soll. Einen solchen Fall demonstriert das folgende Beispiel:

Beispiel 3.23: (Tupelvariablen in SQL)

Die Abfrage (siehe auch **Übung 3.4**) ...

GIB DIE NAMEN ALLER PERSONEN AUS, DIE MEHRERE PROJEKTE LEITEN.

... kann in SQL folgendermaßen dargestellt werden:

```
SELECT DISTINCT ANGEST.NAME
FROM ANGEST, PROJEKT AS P1, PROJEKT AS P2
WHERE  ANGEST.ANGNR = P1.P_LEITER AND
       ANGEST.ANGNR = P2.P_LEITER AND
       P1.PNR != P2.PNR[8]
```

In dieser Abfrage werden durch FROM ... PROJEKT AS P1, PROJEKT AS P2 zwei Tupelvariablen für die Tabelle PROJEKT definiert[9]. Gemäß der WHERE-Klausel wird ein Angestellter mit zwei Zeilen (P1 und P2) verbunden, die verschiedene Projekte repräsentieren (P1.PNR != P2.PNR), wobei der Angestellte aber in beiden Projektleiter ist (ANGEST.ANGNR= P1.P_LEITER und ANGEST.ANGNR= P2.P1_LEITER).

Übung 3.9:

Formulieren Sie die Abfrage aus Beispiel 3.23 mit nur einer Tupelvariable.

Übung 3.10:

*Formulieren Sie die Abfragen a) bis c) aus **Übung 3.4** als SQL-Anweisungen.*

Übung 3.11:

Geben Sie an, ob die SQL-SELECT-Anweisung eher Ausdrücken der Relationenalgebra oder des Relationenkalküls entspricht und begründen Sie dies.

Der SQL-Standard sieht vor, dass JOIN-Operatoren in der FROM-Klausel angegeben werden. U.a. kann dadurch die Lesbarkeit einer SELECT-Anweisung verbessert werden, weil sich im Idealfall die Angaben in der WHERE-Klausel auf die gewünschten Eigenschaften der einzelnen Entitäten (repräsentiert durch die Tupel der Relationen) konzentrieren.

---

8 '!=' steht in SQL für den mathematischen Vergleichoperator '≠'(„ungleich“) und ist synonym zu '<>'.

9 Das Schlüsselwort AS ist optional.

Beispiel 3.24: (JOIN-Operator)

Die Abfrage aus Beispiel 3.21 lautet mit SQL-JOIN-Operator:

expliziter JOIN in SQL

```
SELECT PNR, NAME
FROM PROJEKT JOIN ANGEST
ON P_LEITER = ANGNR
```

Durch diese Schreibweise wird klar, dass nur der JOIN von PROJEKT und ANGEST gefragt ist. Bedingungen an die einzelnen Tabellen (etwa: BERUF = 'PROGRAMMIERER') werden nicht gestellt.

Die Abfrage aus Beispiel 3.22 lautet mit SQL-JOIN-Operator:

```
SELECT NAME
FROM ANGEST NATURAL JOIN ANG_PRO
WHERE PNR = 17
```

Hier wird deutlicher: Alle Angestellten mit Arbeitsanteilen (NATURAL JOIN) am Projekt Nummer 17 (WHERE-Klausel) gesucht. Eine weitere Möglichkeit einen NATURAL JOIN in SQL zu schreiben ist:

```
SELECT NAME
FROM ANGEST JOIN ANG_PRO
USING ANGNR
WHERE PNR = 17
```

Das Schlüsselwort USING kann benutzt werden, um ein Attribut für den NATURAL JOIN explizit anzugeben z.B. wenn mehrere gleichnamige Spalten vorhanden sind. Die ON bzw. USING Klausel wurde eingeführt, um die JOIN Bedingungen syntaktisch von den Selektionsbedingungen in der WHERE Klausel zu trennen[10].

Die Abfrage aus Beispiel 3.23 lautet mit JOIN...ON:

```
SELECT DISTINCT NAME
FROM ANGEST JOIN PROJEKT AS P1 JOIN PROJEKT AS P2
ON ANGNR = P1.P_LEITER AND
ANGNR = P2.P_LEITER AND
P1.PNR != P2.PNR
```

Wiederum geht es offensichtlich nur um verschiedene JOIN-Operationen (siehe auch Lösung zu **Übung 3.4**). Die WHERE-Klausel kann entfallen.[11]

Beispiel 3.24a: (OUTER JOINS)
Wir erweitern für dieses Beispiel die Tabelle PROJEKT um eine Zeile:

10 Wenn Sie ON oder USING benutzen, müssen Sie das Schlüsselwort JOIN zum verbinden der Tabellennamen verwenden.

11 Bei manchen DBMS müssen Sie die einzelnen JOINs verschachteln
...FROM (JOIN PROJEKT AS P1 ON ANGNR =P1.LEITER) JOIN PROJEKT AS P2....

PROJEKT	P_NAME	PNR	P_BESCHR	P_LEITER
	DATAWAREHOUSE	12	...	205
	INTRANET	18	...	117
	PROJEKT 2000	17	...	198
	VU	33	...	198
	SQL KURS	34	...	NULL

Die Firma in unserem Beispiel hat demnach ein neues Projekt mit dem Namen 'SQL_KURS' initiiert. Das Projekt hat momentan noch keinen Leiter, außerdem sind ihm noch keine Mitarbeiter zugeordnet. Folgende Anfrage kann mit den bislang betrachteten JOINS nicht realisiert werden:

FINDE ALLE PROJEKTNAMEN UND DIE NUMMERN DER ZUGEORDNETEN ANGESTELLTEN

Die QUERY

```
SELECT PROJEKT.P_NAME, ANG_PRO.ANGNR
FROM PROJEKT NATURAL JOIN ANG_PRO
```

liefert das Ergebnis

DATAWAREHOUSE	205
INTRANET	117
INTRANET	198
PROJEKT 2000	112
PROJEKT 2000	198
VU	117

Somit geht die Information über das Projekt SQL KURS nicht in das Ergebnis der Abfrage ein. Um ein korrektes Ergebnis zu erhalten, müssen die Zeilen der Tabelle Projekt, die nicht der JOIN Bedingung entsprechen erhalten bleiben. Hierzu dienen die OUTER JOINS[12].

- LEFT OUTER JOIN: Es bleiben die nicht der JOIN-Bedingung entsprechenden Zeilen der Tabelle erhalten, die links (vom Schlüsselbegriff OUTER JOIN) steht.
- RIGHT OUTER JOIN: Es bleiben die nicht der JOIN-Bedingung entsprechenden Zeilen der Tabelle erhalten, die rechts steht.
- FULL OUTER JOIN: Es bleiben die nicht der JOIN-Bedingung entsprechenden Zeilen der Tabellen erhalten, die links und rechts stehen.

12 Im Gegensatz dazu werden die bislang behandelten JOINS als 'INNER JOIN' bezeichnet. Es ist möglich INNER JOIN explizit in der Query anzugeben (FROM TABELLE1 INNER JOIN TABELLE2).

Zur Verdeutlichung benutzen wir folgende Tabellen[13]:

EINS	
a	b
a1	b1
a2	b2
a3	NULL

ZWEI	
b	c
b1	c1
b2	c2
b3	NULL

DREI	
c	d
c1	d1
c2	d2
c3	NULL

Ein INNER JOIN über die Tabellen EINS und ZWEI

```
SELECT *
FROM EINS JOIN ZWEI
ON EINS.B = ZWEI.B
```

liefert das Ergebnis:

a1	b1	b1	c1
a2	b2	b2	c2

Ein LEFT OUTER JOIN über die Tabellen EINS und ZWEI

```
SELECT *
FROM EINS LEFT OUTER JOIN ZWEI
ON EINS.B = ZWEI.B
```

liefert das Ergebnis

a1	b1	b1	c1
a2	b2	b2	c2
a3	NULL	NULL	NULL

Entsprechend RIGHT OUTER JOIN und FULL OUTER JOIN:

RIGHT OUTER JOIN

a1	b1	b1	c1
a2	b2	b2	c2
NULL	NULL	b3	NULL

FULL OUTER JOIN

a1	b1	b1	c1
a2	b2	b2	c2
a3	NULL	NULL	NULL
NULL	NULL	b3	NULL

OUTER JOINS über mehrere Tabellen muss man in Abweichung von der INNER JOIN Syntax verschachteln und klammern:

```
SELECT *
FROM (EINS LEFT OUTER JOIN ZWEI ON EINS.B = ZWEI.B)
LEFT OUTER JOIN DREI
ON ZWEI.C = DREI.C
```

13 Die Namen der Tabellen und Spalten sind grau, die eigentlichen Inhalte weiß hinterlegt.

Das Ergebnis der Query ist wie folgt:

a1	b1	b1	c1	c1	d1
a2	b2	b2	c2	c2	d2
a3	NULL	NULL	NULL	NULL	NULL

Eine Query, die das richtige Ergebnis zur Frage nach den Projekten und den Angestellten liefert, ist demnach:

```
SELECT PROJEKT.P_NAME, ANG_PRO.ANGNR
FROM PROJEKT LEFT OUTER JOIN ANG_PRO
ON PROJEKT.PNR = ANG_PRO.PNR
```

Sie liefert folgendes Ergebnis:

DATAWAREHOUSE	205
INTRANET	117
INTRANET	198
PROJEKT 2000	112
PROJEKT 2000	198
VU	117
SQL KURS	NULL

Übung 3.11a:
Formulieren Sie eine Query, die die Namen der Projekte und die Namen und Nummern von evtl. zugeordneten Mitarbeitern ausgibt.

Die Grundidee hinter dem Outer Join besteht darin, Informationen zu erhalten, die bei einem Inner Join verloren gehen. Dieser Informationsverlust kommt dadurch zustande, dass durch die Join-Bedingung Zeilen herausgefiltert werden. Dies passiert weil in einer der beiden Tabellen Zeilen vorhanden sind, die nach der Join-Bedingung keiner Zeile in der anderen Tabelle entsprechen. Ein Outer Join bezieht diese Zeilen mit ein und setzt in die Zeilenpositionen einen NULL-Wert, in der Werte einer passenden Zeile der anderen Tabelle auftauchen würden – wenn es eine passende Zeile gäbe. Ein Outer Join beinhaltet Zeilen, die ein Inner Join herausfiltert. Sie sollten Outer Joins benutzen, wenn Sie meinen, dass Informationen sonst verloren gehen könnten.

Sie sollten allerdings auch bei der Verwendung von Outer Joins vorsichtig vorgehen, da sie auch entscheidende Nachteile haben.

- **Syntax:** Die Implementierung der Outer Join Syntax ist in vielen Systemen nicht standardkonform.
- **Null-Werte (**mehr zu Nullwerten in Abschnitt 3.3.2)**:**
  - **NOT NULL:** Outer Joins setzen sich über NOT NULL-Constraints hinweg, d.h. ein Attribut, das Sie als NOT NULL definiert haben kann in einem Outer Join durchaus NULL werden.
  - **NULL:** Es ist nicht möglich zu erkennen, ob ein NULL-Wert in der ursprünglichen Tabelle schon vorhanden war, oder ob er durch den Outer Join eingefügt wurde.

- **Performance:** Outer Joins sind immer langsamer als Inner Joins.
- **Verkettung und kognitive Überlastung:** Eine Query, in der ein TABELLE1 LEFT OUTER JOIN TABELLE2 RIGHT OUTER JOIN TABELLE3 FULL OUTER JOIN TABELLE4 vorkommt, ist schwierig zu interpretieren. Da dies nicht nur für Studierende und Kursautoren, sondern auch für Hersteller von DBMS gilt, ist es wahrscheinlich, dass Sie bei solchen Konstruktionen nicht das erwartete Ergebnis erhalten.

Beispiele und nähere Erläuterungen finden Sie in /BECK04/.

Neben dem „Standard-Satz“ an logischen und Vergleichsoperatoren (AND, OR, NOT, '=', '!=', '<', '>',...) können die Bedingungen in der WHERE-Klausel weitere Operatoren enthalten, die in der Regel besondere Eigenschaften von einzelnen Werten ermitteln (SUBSTRING, LIKE, BETWEEN, ...) oder arithmetische Ausdrücke bilden ('+', '–', '*', '/', ...). Von besonderer Bedeutung ist aber, dass innerhalb der WHERE-Klausel und seit SQL:1999 /ANSI99/ auch in der FROM-Klausel Mengen an Werten bzw. Relationen betrachtet werden können.

Nehmen Sie an, Sie wollten den Namen des Angestellten ermitteln, der im PROJEKT 2000 mitarbeitet und das höchste Gehalt hat. Die folgende Query:

```
SELECT ANGEST.NAME
FROM ANGEST JOIN ANG_PRO JOIN PROJEKT
ON ANGEST.ANGNR = ANG_PRO.ANGNR
AND ANG_PRO.PNR = PROJEKT.PNR
WHERE ANGEST.GEHALT = MAX(ANGEST.GEHALT)[14]
```

führt nicht zum gewünschten Erfolg. Stattdessen erhält man eine Fehlermeldung, die besagt, dass eine Gruppenfunktion hier nicht zulässig ist. Abhilfe schafft folgende Query.

Beispiel 3.25: (geschachtelte Abfragen I)

```
SELECT DISTINCT ANGEST.NAME
FROM ANGEST JOIN ANG_PRO JOIN PROJEKT
ON ANGEST.ANGNR = ANG_PRO.ANGNR
AND ANG_PRO.PNR = PROJEKT.PNR
WHERE ANGEST.GEHALT =
   (SELECT MAX(ANGEST.GEHALT)
   FROM ANGEST)
```

IN-Operator

Die Abfrage aus Beispiel 3.22 kann analog zum obigen Beispiel mit dem IN-Prädikat folgendermaßen formuliert werden:

14 Aggregierende Funktionen wie MAX werden wir im weiteren Verlauf des Kapitels noch genauer besprechen.

```
SELECT NAME
FROM ANGEST
WHERE ANGNR IN
        (SELECT ANGNR
         FROM ANG_PRO
         WHERE PNR = 17)
```

Mit dem zweiten SELECT wird eine Tabelle erzeugt, in der die Nummern aller Angestellten enthalten sind, die am Projekt 17 mitarbeiten. Die erste SELECT-Anweisung untersucht nun für jede Zeile in ANGEST mittels des IN-Prädikats, ob ANGEST.ANGNR im Ergebnis der inneren SELECT Anweisung enthalten ist.

geschachtelte Abfragen

Wenn – wie im letzten Beispiel – innerhalb einer SQL-Query eine weitere SQL-Query auftaucht, so wird das als *Schachtelung* von Abfragen bezeichnet. Die in ihrer WHERE-Klausel eine weitere Abfrage enthaltende SELECT-Anweisung wird als *äußere Abfrage* bezeichnet – die in der WHERE-Klausel stehende SELECT-Anweisung entsprechend als *innere Abfrage* oder auch als *SUBQUERY* oder *SUBSELECT*.

In einer WHERE-Klausel können mehrere Abfragen enthalten sein. Die Schachtelungstiefe ist (theoretisch) beliebig: eine innere Abfrage kann also in ihrer WHERE-Klausel wiederum Abfragen enthalten. Ferner sind in inneren Abfragen die in der äußeren Abfrage betrachteten Tabellen und Spalten sichtbar. Falls von dieser Möglichkeit Gebrauch gemacht wird, spricht man von korrelierten Subqueries.

Beispiel 3.26: (geschachtelte Abfragen II)

Die Abfrage aus Beispiel 3.23:...

> GIB DIE NAMEN ALLER PERSONEN AUS, DIE MEHRERE PROJEKTE LEITEN.

...lässt sich auch als geschachtelte Abfrage formulieren:

EXISTS-Operator

```
SELECT DISTINCT NAME
FROM ANGEST, PROJEKT AS P1
WHERE  ANGNR = P1.P_LEITER AND
          EXISTS ( SELECT *
                   FROM PROJEKT AS P2
                   WHERE  ANGNR = P2.P_LEITER AND
                          P1.PNR != P2.PNR)
```

Man kann sich die Abarbeitung hier folgendermaßen vorstellen: In der äußeren Abfrage wird jeder Angestellte betrachtet und in P1 ein Projekt gesucht, dessen Leiter er ist (ANGNR = P1.P_LEITER). Mit Hilfe der inneren Abfrage wird jeweils festgestellt, ob (EXISTS) es noch andere Projekte (P1.PNR != P2.PNR) mit eben diesem Projektleiter (ANGNR = P2.P_LEITER) gibt.

Das Prädikat EXISTS (siehe letztes Beispiel) testet lediglich, ob das Ergebnis überhaupt einen Wert enthält. Deshalb ist die Struktur des Ergebnisses unwichtig

und es wird traditionell '*' angegeben. Hier bedeutet * nicht die Abfrage aller Attribute/Spalten, sondern eher 'Irgendwas/Irgendein Wert'. Sie sollten sich an diese inoffizielle Konvention halten. Die Prädikate IN (siehe Beispiel 3.25), ALL und ANY oder synonym SOME (siehe folgendes Beispiel) erwarten als Ergebnis einfache Listen, dürfen also nur Werte einer Spalte liefern. Entsprechend akzeptieren diese Prädikate auch explizit aufgeführte Listen. Statt...

```
WHERE ANGNR IN (SELECT ...)
```

...kann z.B. ...

```
WHERE ANGNR IN (101, 115, 121, 127, ...)
```

... angegeben werden. Der Standard /ANSI99/ unterscheidet skalare Subqueries, die nur einen skalaren Wert liefern, Zeilen-Subqueries, die genau eine Zeile zurückgeben und Tabellen-Subqueries, die Tabellen zurückgeben. Steht eine Subquery z.B. anstelle eines einfachen Attributs, muss es eine skalare Subquery sein.

Beispiel 3.27: (Ergebnisse von SUBQUERIES)

Wir erweitern die Firma für die folgenden Beispiele um einen weiteren Mitarbeiter, Herrn Schulze, der an den Projekten 33 und 18 mit je 30% beteiligt ist, am Projekt 17 mit 35% und am Projekt 12 mit 5%.
Zwei Abfragen mit ALL, ANY und ein SUBSELECT:

a) FINDE ALLE PERSONEN, DIE IN KEINEM IHRER PROJEKTE EINEN ARBEITSZEIT-SCHWERPUNKT HABEN. (Gemeint sind Personen, die in keinem Projekt mit mehr als 40% ihrer eigenen Arbeitszeit beteiligt sind.)

ALL-Operator

```
SELECT NAME
FROM ANGEST
WHERE 40 > ALL ( SELECT PROZ_ARB
                 FROM ANG_PRO
                 WHERE ANG_PRO.ANGNR=ANGEST.ANGNR)
```

b) FINDE ALLE PERSONEN, DIE IN IRGENDEINEM PROJEKT NUR SEHR WENIG TÄTIG SIND. (Gemeint sind Personen, die an einem bestimmten Projekt mit 5% oder weniger ihrer eigenen Arbeitszeit beteiligt sind.)

ANY-Operator

```
SELECT NAME
FROM ANGEST
WHERE 5 >= ANY ( SELECT PROZ_ARB
                 FROM ANG_PRO
                 WHERE ANGNR = ANGEST.ANGNR)
```

c) FINDE DIE PERSONEN, DIE MEHR IN EINEM PROJEKT ARBEITEN ALS DER PROJEKTLEITER[15].

Abfrage anstelle eines Wertes

```
SELECT DISTINCT NAME
FROM ANGEST, ANG_PRO, PROJEKT
WHERE  ANGEST.ANGNR = ANG_PRO.ANGNR AND
       ANG_PRO.PNR = PROJEKT.PNR AND
       PROZ_ARB >( SELECT PROZ_ARB
                   FROM ANG_PRO AP
                   WHERE AP.ANGNR=
                                PROJEKT.P_LEITER AND
                         AP.PNR= PROJEKT.PNR)
```

Zwei weitere Prädikate, die SQL für die Auswertung von Subselects zur Verfügung stellt sind UNIQUE, ein Test auf doppelte Zeilen und MATCH. Zum Umgehen von schwer zu verstehenden NOT EXISTS Konstruktionen, wie zum Beispiel „doppelte NOT EXISTS Konstruktionen“ als Ersatz für FOR ALL, hat SQL:1999 die quantifizierten Prädikate FOR ALL, FOR ANY und FOR SOME eingeführt.[16]

Für die mengenorientierten Operationen Differenz, Durchschnitt und Vereinigung enthält SQL die Operatoren EXCEPT, INTERSECT und UNION, von denen jedoch in vielen DBMS nur der letztgenannte unterstützt wird[17].

Beispiel 3.28: (SQL-Mengen-Operatoren)

Die folgende SQL-Anweisung bildet die Differenz der Namen von Angestellten und Kunden:

```
   SELECT NAME
   FROM ANGEST
EXCEPT
   SELECT NAME
   FROM KUNDE
```

Um Durchschnitt oder Vereinigung zu bilden, ist anstatt EXCEPT lediglich INTERSECT oder UNION einzusetzen. Entsprechend der Beispieldaten aus **Beispiel 3.2** und **Beispiel 3.4** fällt in der Differenz der Name 'SCHMIDT' heraus; im Durchschnitt ist nur dieser Name vorhanden.

Die Angabe von DISTINCT in den SELECT-Klauseln ist beim Einsatz der mengenorientierten SQL-Operationen überflüssig, da EXCEPT, INTERSECT und UNION doppelte Tupel entfernen (Sind doppelte Tupel erwünscht, muss hinter den Operatoren das Schlüsselwort ALL angegeben werden.).

---

15 Gemeint ist die Zuordnung in Prozenten der Arbeitszeit, sonst wäre die Antwort wahrscheinlich 'alle'.

16 Zur genauen Syntax und zur Verwendung verweisen wir auf /GULU99/ S. 598 ff

17 Bei ORACLE heißt der EXCEPT Operator MINUS.

Übung 3.12:

Geben Sie die Differenz und den Durchschnitt der Relationen ANGEST und KUNDE wie in Beispiel 3.28 als SELECT-Anweisung ohne EXCEPT und INTERSECT an.

In der SELECT-Klausel einer Abfrage können neben der bereits eingeführten Duplikateliminierung (mit DISTINCT) und der einfachen Angabe aller Attribute (mit '*') noch die in der Abfrage ermittelten Werte manipuliert werden. Zum einen können Attributwerte mit arithmetischen Operatoren oder Operatoren zur Manipulation von Zeichenketten zusammengesetzt werden. Zum anderen können die Attribute der Ergebnisrelation neu bezeichnet oder auch umbenannt werden. So liefert die Abfrage...

Bezeichnung von Attributen mit AS

```
SELECT GEHALT*PROZ_ARB/100 AS PGEHALT ...
```

... eine Tabelle mit einer Spalte mit dem Namen PGEHALT, die den Anteil des Gehalts darstellt, den ein Angestellter in einem Projekt verdient.

Die Zeilen der Ergebnistabelle einer Abfrage können mit der ORDER BY-Klausel sortiert werden. Nach ORDER BY werden die Attribute, nach welchen das Ergebnis sortiert sein soll, aufgezählt. Für jedes Attribut kann im Anschluss festgelegt werden, ob die Sortierung in aufsteigender (ASC) oder absteigender (DESC) Reihenfolge durchgeführt wird. Als Sortierkriterien kommen nicht nur diejenigen Attribute in Frage, die selbst Teil des Ergebnisses sind, sondern alle Attribute die über einen JOIN mit dem Ergebnis verknüpft sind. Dies ist allerdings erst seit SQL:1999 so, in SQL-92 ist das Ordnen nach Attributen, die nicht in der SELECT Liste auftauchen, nicht erlaubt.

Beispiel 3.29: (Sortierung)

Die folgende Abfrage liefert eine Aufstellung aller Projekte und ihrer Mitarbeiter. Die Projekte sind nach Projektnamen sortiert und je Projekt sind die Mitarbeiter nach der Arbeitszeit sortiert, welche sie für das Projekt aufwenden:

```
SELECT NAME, P_NAME
FROM ANGEST NATURAL JOIN ANG_PRO NATURAL JOIN
PROJEKT
ORDER BY P_NAME, PROZ_ARB
```

Als wichtige Erweiterung über die relationale Vollständigkeit hinaus bietet SQL eine Reihe von aggregierenden *Funktionen* (im Standard set functions genannt)

aggregierende Funktionen

- SUM (Summe)
- AVG (Durchschnitt)
- COUNT (Zahl von Werten in einer Spalte)
- MAX (größter Wert in einer Spalte)
- MIN (kleinster Wert in einer Spalte)

- EVERY (nur anwendbar auf Spalten BOOLESCHEn Typs) gibt FALSE zurück, wenn mindestens einer der zu aggregierenden Werte FALSE ist, sonst TRUE. Diese Funktion ist neu im SQL:1999 Standard /ANSI99/.
- ANY oder synonym SOME (nur anwendbar auf Spalten BOOLSCHEn Typs) gibt TRUE zurück, wenn mindestens einer der zu aggregierenden Werte TRUE ist, sonst FALSE. Diese Funktion ist neu im SQL:1999 Standard /ANSI99/.

Jede dieser Funktionen, arbeitet auf der Menge der Werte einer Spalte innerhalb einer Tabelle. Um die Anzahl von Zeilen in einer Tabelle zu bestimmen, wird meist COUNT(*) benutzt. Alle aggregierenden Funktionen ignorieren NULL-Werte, auch COUNT.

Beispiel 3.30: (Aggregierungsfunktionen)

a) GIB DIE SUMME DER GEHÄLTER ALLER ANGESTELLTEN AN.

SUM

```
SELECT SUM(GEHALT)
FROM ANGEST
```

b) GIB DIE ZAHL ALLER ZEILEN DER TABELLE ANGEST AN.

COUNT

```
SELECT COUNT(*)
FROM ANGEST
```

c) Gib die Zahl der Projektleiter an.

```
SELECT COUNT(DISTINCT P_LEITER)
FROM PROJEKT
```

d) Gib die Zahl der geleiteten Projekte an

```
SELECT COUNT(P_LEITER)
FROM PROJEKT
```

Zur Verdeutlichung der neuen Aggregierungsfunktionen EVERY, ANY und SOME betrachten wir folgende Tabellen:

WAHR1
SPALTE
TRUE
TRUE
FALSE
FALSE
UNKNOWN

WAHR2
SPALTE
TRUE
TRUE
UNKNOWN

WAHR3
SPALTE
FALSE
FALSE
UNKNOWN

e) EVERY (oder EVERY(ALL… oder EVERY(DISTINCT…)

Das folgende Statement gibt FALSE zurück:

```
SELECT EVERY(SPALTE) FROM WAHR1
```

Das folgende Statement gibt TRUE zurück:

```
SELECT EVERY(SPALTE) FROM WAHR2
```

f) ANY oder synonym SOME oder in Kombination mit ALL oder DISTICT

Das folgende Statement gibt TRUE zurück:

```
SELECT ANY(SPALTE) FROM WAHR1
```

Das folgende Statement gibt FALSE zurück:

```
SELECT ANY(SPALTE) FROM WAHR3
```

Übung 3.13:

Geben Sie für die folgende Abfrage eine SQL-Abfrage mit der MAX-Funktion und eine SQL-Abfrage ohne Aggregierungsfunktion an:

GIB DAS HÖCHSTE GEHALT EINES PROGRAMMIERERS AN.

Zusatzaufgabe:
Geben Sie außerdem einen entsprechenden Ausdruck der relationalen Algebra an.

(Siehe auch **Beispiel 3.11**)

Häufig sollen verschiedene Teilmengen einer Tabelle miteinander verglichen werden, indem zu jeder Teilmenge ein zusammenfassendes Datum betrachtet wird, wie in der Abfrage im folgenden Beispiel. Um solche Abfragen zu beantworten, müssen jeweils alle interessanten Teilmengen der Relation zusammengestellt und dann muss jeweils die gewählte Aggregierungsfunktion angewendet werden. Diesen Vorgang, die Tabelle bezüglich eines oder mehrerer Attribute in disjunkte Teilmengen zu zerlegen, nennt man *Gruppierung*. SQL bietet dazu die GROUP BY-Klausel an, die an die SELECT-Anweisung angehängt wird.

Beispiel 3.31: (Gruppierung)

Ausgegangen wird wiederum von den in **Beispiel 3.4** angegebenen Relationen.

GIB FÜR JEDE ABTEILUNG DEREN NUMMER, DIE SUMME ALLER GEHÄLTER UND DAS DURCHSCHNITTSGEHALT DER ANGESTELLTEN DIESER ABTEILUNG AN.

```
SELECT ABTNR, SUM(GEHALT), AVG(GEHALT)
FROM ANGEST
GROUP BY ABTNR
```

GROUP BY

Durch GROUP BY werden die Zeilen der Tabelle ANGEST nach den Werten von ABTNR zu „Gruppen“ zusammengefasst (Partition). Auf jede dieser Teilmengen wird die Funktion AVG bzw. SUM angewandt, d.h. es werden der Durchschnittswert und die Summe aller Gehälter in dieser Teilmenge ermittelt. Sollen nur Abteilungen erfasst werden, deren Gehaltssumme größer als 6000 ist, so kann diese Bedingung in der HAVING-Klausel angegeben werden:

```
SELECT ABTNR, SUM(GEHALT), AVG(GEHALT)
FROM ANGEST
GROUP BY ABTNR
HAVING SUM(GEHALT) > 6000;
```

HAVING

Während die GROUP BY-Klausel darüber entscheidet, wie Gruppierungen gebildet werden, bestimmt die HAVING-Klausel, welche Gruppen im Ergebnis aufgenommen werden. Die HAVING-Klausel erlaubt dabei - anders als die WHERE-Klausel - Gruppen als Ganzes zu untersuchen, indem Bedingungen bzgl. aggregierter Werte gestellt werden können.

Übung 3.14:

Geben Sie eine SELECT-Anweisung für folgende Abfrage an:

GIB JEWEILS DIE SUMME DER ARBEITSZEIT ALLER MITARBEITER (Z.B.: 1150 PROZENT) FÜR DIE PROJEKTE 10 BIS 19 AN, SOWEIT MEHR ALS ZEHN PERSONEN DARAN BETEILIGT SIND.

In den SQL:1999 Standard wurden neue Möglichkeiten zur Gruppierung aufgenommen, wie GROUP BY CUBE, GROUP BY ROLLUP und GROUP BY GROUPING SETS. Sie können sich in /GULU99/ ab S. 637 über die genaue Syntax informieren. Beispiele und nähere Erläuterungen finden Sie in /BECK04/.

Erweiterungen in kommerziellen DBMS

In verschiedenen DBMS wurden weitere Spracherweiterungen implementiert, die z.B. rekursive Abfragen (zur Berechnung von Hierarchien oder transitiven Hüllen etc.) erlauben[18], die mit relational vollständigen Sprachen nicht möglich sind. Dies führte zur Implementierung dieser und weiterer Erweiterungen in den SQL:1999 Standard, der z.B. rekursive Abfragen mit WITH RECURSIVE ermöglicht[19]. Andere Ansätze beschäftigen sich mit der Modellierung und Abfrage zeitlicher Abläufe und oder räumlicher Daten oder objektrelationalen Erweiterungen. Diese Konzepte haben in SQL:1999 in hohem Maße Eingang gefunden.[20] Eine eingehende Darstellung finden Sie in /MELT03/.

INSERT, UPDATE- und DELETE

Neben dem Retrieval erlaubt SQL natürlich auch das Einfügen, Verändern und Löschen von Zeilen in der Datenbank. Dazu stehen die Anweisungen INSERT, UPDATE und DELETE zur Verfügung.

Bei UPDATE und INSERT werden die Daten, die eingefügt oder verändert werden sollen, als Zeilen oder als Zeilenabfrage angegeben. Bei UPDATE und DELETE kann in einer WHERE Klausel (gegebenenfalls auch geschachtelt) spezifiziert werden, welche Sätze geändert werden sollen.

---

18 z.B. realisiert in ORACLE mit CONNECT BY…PRIOR

19 siehe /GULU99/ S. 629 f.

20 Es gibt verschiedene Versuche, durch SQL nahegelegte und auch andere Erweiterungen „mathematisch“ in das Relationenmodell einzuarbeiten. Eine veränderte Auffassung von den Grundlagen des relationalen Datenmodells – wie sie zu Beginn des Kapitels mit Algebra und Kalkül vorgestellt wurden – hat sich dadurch bisher nicht ergeben.

Wir geben nun für jede der drei Anweisungen ein Beispiel:

Beispiel 3.32: (INSERT)

FÜGE EINEN NEUEN ANGESTELLTEN ´MEIER´ EIN

... lautet in SQL:

INSERT

```
INSERT INTO ANGEST
VALUES ( 128, ´MEIER´, ´HAGEN´, ´INGENIEUR´, 5400 , 3 )
```

Bei dieser Art des Einfügens einer neuen Zeile müssen die Werte in der gleichen Reihenfolge angegeben werden, wie bei der Definition der Tabelle ANGEST. Einfach ist einzusehen, dass ein Anwendungsprogramm, das diese Notation verwendet, bei jeder Änderung des Schemas – und sei es nur das Hinzufügen eines Attributes – geändert werden muss. Eine gute Lösung, um Programme und konzeptuelles Schema zu entkoppeln, stellen externe Sichten dar, die wir später einführen werden. Doch auch die INSERT-Anweisung lässt sich (in geringem Maße) im Sinne der Datenunabhängigkeit besser darstellen:

```
INSERT INTO ANGEST
(ANGNR, NAME, WOHNORT, BERUF, GEHALT, ABTNR)
VALUES ( 128, ´MEIER´, ´HAGEN´, ´INGENIEUR´, 5400 , 3 )
```

Durch die explizite Deklaration der einzufügenden Werte ist die Anweisung unabhängig von der Reihenfolge und muss auch bei verschiedenen Veränderungen der Relation ANGEST nicht angepasst werden. [21]

Ferner kann mit einer INSERT-Anweisung auch eine Reihe von Zeilen eingefügt werden, indem entweder die konkreten Werte explizit (durch VALUES (Wert1, Wert2), (Wert3, Wert4),...) angegeben werden, oder stattdessen eine SELECT-Anweisung die Menge der einzufügenden Daten berechnet.

Beispiel 3.33: (UPDATE)

ERHÖHE DAS GEHALT DER ANGESTELLTEN DER ABTEILUNG NUMMER 30 UM 10 %

... lautet in SQL:

UPDATE

```
UPDATE ANGEST
SET GEHALT = GEHALT * 1.1
WHERE ABTNR = 30
```

Wie die INSERT-Anweisung, so kann auch die UPDATE-Anweisung die Änderungswerte per SELECT-Anweisung aus der Datenbank bestimmen:

ERHÖHE DAS GEHALT DER MITARBEITER AM PROJEKT NUMMER 17 ABHÄNGIG VON DER BETEILIGUNG UM 10 %

... lautet in SQL:

---

21 Vereinfacht gesagt, funktioniert diese Anweisung, solange alle darin nicht aufgeführten (zukünftigen) Felder der Tabelle ANGEST einen sogenannten DEFAULT-Wert haben oder automatisch mit NULL-Werten belegt werden.

```
UPDATE ANGEST
SET GEHALT = GEHALT + GEHALT * 0.1 *
                (SELECT PROZ_ARB
                 FROM ANG_PRO
                 WHERE ANG_PRO.ANGNR=ANGEST.ANGNR
                        AND ANG_PRO.PROJEKT = 17) /100
WHERE ANGNR IN (SELECT ANGNR
                    FROM ANGPRO
                    WHERE PNR = 17)
```

Beispiel 3.34: (DELETE)

DER ANGESTELLTE 'MÜLLER' ARBEITET NICHT MEHR AN DEM PROJEKT NUMMER 10 MIT

... lautet in SQL:

DELETE

```
DELETE FROM ANG_PRO
WHERE  PNR=10 AND
          ANGNR =  (SELECT ANGNR
                    FROM ANGEST
                    WHERE NAME = 'MÜLLER')
```

Schemamanipulation

Während bisher die SQL-DML beschrieben wurde, kommen wir nun zur Schemamanipulation (Datendefinition).

Wie die verschiedenen Begriffe aus dem Relationenmodell bei der Definition einer SQL-Tabelle abgebildet werden können, beschreiben wir im folgenden Beispiel:

Beispiel 3.35: (CREATE TABLE)

Die Definition der Relation ANG_PRO wie in **Beispiel 3.4** beschrieben:

Erzeugen einer Tabelle:
CREATE TABLE

```
CREATE TABLE ANG_PRO
(PNR          INTEGER,
ANGNR         INTEGER,
PROZ_ARB      INTEGER,
CONSTRAINT ANG_PRO_PK PRIMARY KEY (PNR, ANGNR),
CONSTRAINT ANG_PRO_ANGNR_FK FOREIGN KEY (ANGNR)
REFERENCES ANGEST(ANGNR),
CONSTRAINT ANG_PRO_PROJEKT_FK FOREIGN KEY (PNR)
REFERENCES PROJEKT(PNR))
```

Diese Anweisung erzeugt die - zunächst leere - Relation bzw. Tabelle ANG_PRO. Diese Tabelle enthält drei Spalten bzw. Attribute: PNR, ANGNR und PROZ_ARB. Der Wertebereich dieser Attribute ist jeweils die Menge der ganzen Zahlen (INTEGER). Die Tabelle darf Zeilen enthalten, die

für das Attribut PROZ_ARB einen undefinierten Wert (einen NULL-Wert) besitzen.

Mit CONSTRAINT ANG_PRO_PK PRIMARY KEY (PNR, ANGNR) wird festgelegt, dass die Kombination der Zeilen (PNR, ANGNR) den Primärschlüssel der Relation bildet, dieses Constraint erhält den Namen ANG_PRO_PK. Schlüsselattribute müssen immer einen von NULL verschiedenen Wert besitzen und eindeutig sein. Das Constraint PRIMARY KEY ist somit eine Kombination aus den Constraints NOT NULL und UNIQUE.

Schlüssel

CONSTRAINT ANG_PRO_ANGNR_FK FOREIGN KEY (ANGNR) REFERENCES ANGEST legt fest, dass ANGNR ein *Fremdschlüssel* der Relation ANGEST sein soll. Dies bedeutet, dass für ANGNR nur ein Wert eines Schlüssels einer Zeile der Relation ANGEST zulässig ist; ein anderer Wert für ANGNR wird nicht akzeptiert. Wird ein anderer Schlüssel als der Primärschlüssel ausgewählt, kann die Angabe in Klammern hinter dem Namen der referenzierten Tabelle stehen. Der Primärschlüssel der Relation ANGEST ist das Attribut ANGNR, die Angabe ANGEST(ANGNR) könnte also weggelassen werden. Das Constraint erhält den Namen ANG_PRO_ANGNR_FK.

Fremdschlüssel

Mit dieser Konstruktion wird *referentielle Integrität* erreicht. Es kann in ANG_PRO kein Angestellter aufgeführt werden, der nicht tatsächlich (in ANGEST also) existiert.

referentielle Integrität

Das Schlüsselwort CONSTRAINT ist optional. Wird kein Name für ein Constraint angelegt, so vergibt das DBMS einen vom System generierten. In Anbetracht der Tatsache, dass Constraints Datenbankobjekte sind, die erschaffen, gesucht und gelöscht werden können, bietet es sich an, für Tabellenconstraints (wie die obigen) Namen zu vergeben. Bei Spaltenconstraints (Constraints, die auf Spaltenebene vergeben, allerdings nach der Definition der Tabelle logischerweise zu Tabellenconstraints werden) wie die folgenden, mag dies nicht so wichtig sein.

```
CREATE TABLE TESTTABELLE
(TESTSPALTE1      DATE          NOT NULL,
TESTSPALTE2       VARCHAR(20)   UNIQUE)
```

Bei den meisten DBMS können mit der CREATE TABLE-Anweisung außerdem auch verschiedene Parameter für das interne Repräsentationsschema festgelegt werden. So können z.B. physische Zugriffspfade oder Hilfsstrukturen für den effizienteren Zugriff bei häufigen Abfragen definiert werden. Im SQL-Standard ist dies nicht vorgesehen.

Die Befehle zum Löschen von Tabellen und Ändern der Struktur einer Tabelle sind im SQL-Standard /ANSI99/ zwar vorgesehen und werden i.d.R. von relationalen DBMS angeboten. Abgesehen von „Nachbesserungen“, wie dem Hinzufügen einzelner Attribute, gibt es für die Veränderung von Schemata über der Zeit, die sog. Schema-Evolution, jedoch noch keinen in der Praxis bewährten Ansatz.

Beispiel 3.36: (DROP TABLE)

Löschen einer Tabelle

Ausgehend vom vorangegangenen Beispiel entfernt die folgende SQL-Anweisung die Relation ANG_PRO wieder aus der Datenbank:

```
DROP TABLE ANG_PRO
```

Sichten (Views)

Sichten dienen dazu, dem Benutzer (fast immer ein Anwendungsprogramm) einen Ausschnitt der Datenbank als seine Arbeitsumgebung einzurichten und ihm dabei eine spezielle Darstellung der Daten anzubieten.

Basistabelle

Der Standard nennt die im konzeptuellen Schema definierten und damit im System physisch realisierten Tabellen *Basetables (Basistabellen)*. SQL bietet die Möglichkeit, alternative Sichten auf die im System verwalteten Basisrelationen zu definieren. Der Benutzer kann auf einer solchen Sicht mit Einschränkungen die gleichen Operationen anwenden wie auf den Basisrelationen; die Einschränkungen beziehen sich auf Änderungen. Die Sichten in SQL stellen für den Anwender wiederum Tabellen dar. Eine Sicht wird als Ergebnis einer Abfrage formuliert.

Beispiel 3.37: (CREATE VIEW)

Wir wollen für einen bestimmten Benutzerkreis eine Sicht definieren, die für alle Abteilungen die Zahl der Angestellten angibt.

CREATE VIEW

```
CREATE VIEW ZAHL_DER_ANGEST (ABTNR, ANZAHL) AS
SELECT ABTNR, COUNT (ANGNR)
FROM ANGEST
GROUP BY ABTNR
```

Durch die SQL-Abfrage SELECT ... wird eine Tabelle beschrieben, deren Zeilen gerade aus Abteilungsnummer und Zahl der Angestellten dieser Abteilung bestehen. Durch GROUP BY werden die Zeilen der Tabelle ANGEST nach den Werten von ABTNR gruppiert; auf jede dieser Gruppen wird die Funktion COUNT angewandt, d.h. es wird gezählt, wie viele unterschiedliche Werte für ANGNR vorhanden sind.

Die so definierte Relation erhält den Namen ZAHL_DER_ANGEST. Die Spalten heißen ABTNR und ANZAHL. Nach dieser Definition von ZAHL_DER_ANGEST kann diese View in gleicher Weise in Abfragen einbezogen werden wie eine Basistabelle.

abgeleitete Tabelle

Sichten stellen *abgeleitete Tabellen* (im Standard *derived Tables*) dar, sie unterscheiden sich von Basistabellen dadurch, dass ihr Inhalt bzgl. des konzeptuellen Schemas redundant ist und sie nicht physisch gespeichert sind, sondern zum Zeitpunkt des aktuellen Zugriffs aus den Basistabellen ermittelt werden. Alle Änderungen von Basistabellen werden also sofort in allen darauf definierten Sichten sichtbar.

Sichten dürfen auch auf Sichten aufgebaut werden.
Hierfür ein Beispiel:

Beispiel 3.38: (geschachtelte VIEW-Definition)

Aufbauend auf der Sicht ZAHL_DER_ANGEST aus dem letzten Beispiel zeigt diese Sicht Abteilungsnummer und Anzahl der Angestellten für diejenigen Abteilungen, die mehr als 100 Angestellte haben:

Sichten auf Sichten

```
CREATE VIEW ZDA_100 (ABTNR, ANZAHL) AS
        SELECT ABT_NR, ANZAHL
        FROM ZAHL_DER_ANGEST
        WHERE ANZAHL > 100
```

Sichten werden dynamisch erzeugt, sie existieren nur durch ihre Definition. Damit sind Änderungen über Sichten nur dann möglich, wenn eindeutig klar ist, welche Änderungen auf den zugehörigen Basisrelationen gemeint sind. Das Problem der '*updatable views*' ist komplex, wir verweisen auf den Standard /ANSI99/ und /GULU99/.

Beispiel 3. 39: (VIEW-UPDATES)

Update auf Sichten

Auf einer Sicht ANGEST_SPEZIAL (ANGNR, WOHNORT) können alle Änderungsoperationen erlaubt werden, da das zugrundeliegende Tupel der Basisrelation ANGEST eindeutig definiert ist. Nicht so im Falle einer Sicht

LEITUNG (PNR, P_LEITERNAME),

die auf den Basisrelationen PROJEKT und ANGEST beruht. Würde nämlich z.B. P_LEITERNAME in einem Tupel des Views geändert, so wäre unklar, was dies bedeutet: hat das Projekt einen neuen Leiter (Änderung von PNR in PROJEKT) oder hat der Leiter z.B. wegen Heirat seinen Namen geändert (Änderung von NAME in ANGEST)?

Übung 3.15:

Gegeben seien die Tabellen

ARTIKEL(ARTNR, ARTBEZ, EINKAUFSPREIS, VERKAUFSPREIS)

und

PROD_GRUPPE(GRUPPENNR, ARTNR).

Formulieren Sie mit Hilfe von SQL folgende Abfragen:

a) FINDE ARTIKELNUMMER UND -BEZEICHNUNG FÜR ALLE ARTIKEL, DEREN VERKAUFSPREIS < 100 IST.

b) FINDE ARTIKELNUMMER UND -BEZEICHNUNG FÜR ALLE ARTIKEL, DIE ZU PRODUKTGRUPPE 93 GEHÖREN.

c) ERMITTLE DEN DURCHSCHNITTLICHEN EINKAUFSPREIS JE PRODUKTGRUPPE.

d) Definieren Sie in SQL eine Sicht mit denselben Attributen wie ARTIKEL, in der jedoch nur Tupel vorkommen, die zur Produktgruppe 93 oder 97 gehören.

3.3.2 Konsistenz, Transaktionen und Recovery

Auf einer Datenbank arbeiten meist viele Benutzer oder Anwendungsprogramme gleichzeitig. Es ist deshalb eine wesentliche Aufgabe von DBMS, dafür zu sorgen, dass sich diese parallelen Vorgänge nicht wechselseitig stören oder zu Fehlern in der Datenbank führen. Das zentrale Konzept zur Sicherung der Korrektheit dieses parallelen Geschehens auf einer Datenbank ist das *Transaktionskonzept*. Eine andere zentrale Aufgabe des DBMS ist es, dafür zu sorgen, dass die Datenbank nach dem Auftreten von Fehlern, nach Softwareabstürzen oder nach Speicherfehlern wieder in einen konsistenten Zustand versetzt wird. Dieser Komplex wird mit *Recovery* (Wiederherstellung) umschrieben.

Dieses Kapitel gibt eine kurze Einführung in diese Themenbereiche, insbesondere aus Anwendungssicht. Auf Realisierungskonzepte oder gar Implementierungsfragen innerhalb des DBMS gehen wir nicht ein. Eine umfassende Behandlung finden Sie im Kurs Datenbanken 2 und im Kurs 1665.

Konsistenz und Fehlerursachen

Unter der Konsistenz einer Datenbank verstehen wir die Korrektheit und die Vollständigkeit der darin enthaltenen Informationen.

Beispiele von Inkonsistenz:

a) Referenz ins Leere (jemand löscht ein Tupel, ohne dass die darauf verweisenden Referenzen korrigiert werden)
b) Eine Buchung X ist vom Konto A abgebucht, aber nicht dem Zielkonto B gutgeschrieben
c) Ein Summenfeld gibt nicht die korrekte Summe der Einzelelemente (z.B. Gehaltssumme)
d) Ein negatives Alter
e) Bei Mehrfachspeicherung derselben Daten in verschiedenen Sätzen: Änderung ist an einer Stelle erfolgt, nicht an der anderen

Konsistenzverletzungen können verschiedene Ursachen haben, beispielsweise:

1. Falsche Eingabedaten
2. Fehler im Anwendungsprogramm (falsche Programmierung, etwa im Fall b oder d)
3. Falsche Behandlung paralleler Anwendungsabläufe
4. Abstürze des Betriebssystems oder des DBMS
5. Ausfälle von Massenspeichern
6. Falsche Schemadefinitionen (z.B. ein den Daten nicht angemessener Datentyp)

Natürlich kann das DBMS nur in sehr begrenztem Umfang falsche Eingabedaten oder Inkonsistenzen, die sich aus Fehlern im Anwendungsprogramm ergeben, erkennen. Es gibt jedoch die Möglichkeit, **Integritätsbedingungen** zu definieren, anhand derer das DBMS Korrektheitsprüfungen vornehmen kann. Beispielsweise könnte man dem DBMS im Schema der Datenbank mitteilen, dass das Alter einer Person nie negativ sein kann; oder: dass das Summenfeld „Gehaltssumme“ die Summe aller Gehälter des Unternehmens sein muss; oder dass ein Tupel nur dann gelöscht werden darf, wenn kein anderes Tupel darauf verweist. Hier bieten die verschiedenen kommerziellen Datenbanksysteme unterschiedliche Möglichkeiten an. Etwas genauer wird diese Frage wieder in Datenbanken 2 oder im Kurs 1665 behandelt.

Anders sieht die Situation bezüglich der Punkte 3, 4 und 5 aus. Da der einzelne Anwendungsprogrammierer oder Nutzer der Datenbank von den parallelen Aktivitäten nichts weiß und auf diese auch keinen Einfluss hat, erwarten wir vom DBMS,

- dass es die parallelen Abläufe so organisiert, dass sie nicht zu Inkonsistenzen führen können. Dies bedeutet auch, dass jedes Anwendungsprogramm immer eine konsistente Sicht auf die Datenbank erhält, dass also die von ihm gelesenen Daten konsistent und aktuell sind. (***Transaktionsmanagement***)

- dass das System in der Lage ist, nach dem Absturz von Systemsoftware oder nach einem Plattenfehler wieder einen konsistenten Zustand der Datenbank herzustellen. (***Recovery***)

Beispiel 3.40:

Ein einfaches Beispiel, wie Parallelarbeit zu Inkonsistenzen führen kann, ist das folgende:

Programm 1 möchte die Gehälter von Personen A (aktuell € 10.000) und B (aktuell € 10.000) jeweils um 5% erhöhen.
Programm 2 möchte jeweils € 100 hinzufügen.

Programm 1 greift auf A zu, multipliziert das Gehalt mit 1.05.
Programm 2 greift auf B zu, addiert 100.
Programm 1 greift auf B zu, multipliziert mit 1.05
Ergebnis: Gehalt B = (altes gehalt + 100)*1.05 = € 10605
Programm 2 greift auf A zu, addiert 100
Ergebnis: Gehalt A = (altes gehalt *1.05)+100 = € 10600

Die Datenbank ist inkonsistent geworden, obwohl beide Programme korrekt arbeiten! Außerdem stimmt das Gehaltsgefüge in der Firma nicht mehr. Diese Fehler werden nicht bemerkt, allenfalls per Zufall später, wenn tatsächlich jemand die Gehälter vergleicht. Da aber bis dahin viele weitere Aktionen auf diesen Daten erfolgen können, kann sich dieser Fehler verheerend auswirken, wie Sie sich leicht vorstellen können.

Es ist nicht Aufgabe der Datenbank zu bestimmen, ob das Gehalt 10605€ oder 10600€ sein soll, das ist im Entscheidungsbereich der Applikationen. Aber eine

sehr wichtige Aufgabe der Datenbank ist es, die Änderungen konsistent durchzuführen. Das DBMS muss alle Datenmanipulationen einheitlich durchführen.

Beispiel 3.41:

Dieses Beispiel macht deutlich, dass Abhängigkeiten zwischen Daten zu vorübergehenden Inkonsistenzen führen können, die erst durch spätere Befehle aufgehoben werden: Speichert ein Programm beispielsweise sowohl das Gewicht einzelner Postpakete, als auch die Summe der Gewichte aller Postpakete, so führt das Eintragen eines neuen Paketes so lange zu einer inkonsistenten DB, bis die Summe aller Gewichte korrigiert wird. Würde direkt nach der Eintragung eines neuen Paketes die Konsistenz der DB überprüft, würde das Gesamtgewicht nicht stimmen.[22]

Transaktion

Die obigen Beispiele zeigen etwas sehr wesentliches: Häufig führt nur eine Zusammenfassung von einzelnen, inhaltlich zusammengehörenden Befehlen zu einem konsistenten Datenbankzustand – und diese Folge von Befehlen muss als Ganzes, als *eine* Einheit kontrolliert werden, also dem DBMS bekannt sein. Eine solche Folge von Befehlen, die nur als Ganzes Sinn machen, nennen wir eine *Transaktion*.

Im Beispiel der Pakete müssen die Befehle für das Eintragen eines neuen Paketes und für die damit verbundene Neuberechnung des Gesamtgewichts zu einer Transaktion zusammengefasst werden. Damit weiß das DBMS, dass nur die Ausführung aller Befehle innerhalb der Transaktion einen konsistenten Zustand der Datenbank herstellt. Am Beispiel der Gehaltsberechnungen kann das DBMS verhindern, dass die Programme die Objekte A und B in unterschiedlichen Zuständen lesen, also eine inkonsistente Sicht auf die Datenbank erhalten – es kann dies aber nur dann, wenn wir die Befehlsfolge explizit als Transaktion deklarieren.

Weil Zwischenschritte inkonsistent sein können, dürfen sie für andere Transaktionen nicht sichtbar sein. Deshalb gilt, dass entweder alle Befehle einer Transaktion ausgeführt werden oder keine.

Eine ***Transaktion*** ist also eine Folge von Befehlen, die entweder vollständig und korrekt ausgeführt wird oder überhaupt nicht.

Falls eine Transaktion nicht vollständig ausgeführt werden kann, so bedeutet dies praktisch, dass alle schon ausgeführten Operationen rückgängig gemacht werden müssen. Im Ergebnis dürfen weder andere Transaktionen noch die Datenbank selbst in irgendeiner Weise von dieser abgebrochenen Transaktion beeinflusst worden sein, sie hat nicht existiert. Umgekehrt muss das DBMS natürlich gewährleisten, dass Ergebnisse von Transaktionen, die erfolgreich ausgeführt wurden, unter keinen Umständen verloren gehen.

---

22 Hier zeigt sich auch eine unsaubere Schemadefinition. Es ist immer problematisch, abgeleitete Daten und die Daten selbst gleichzeitig in einer Datenbank zu speichern.

Die Komponente des DBMS, die dafür zuständig ist, dass im Falle von Hardware- oder Softwarefehlern die Datenbank wieder in einen konsistenten Zustand versetzt wird, heißt ***Recovery-Manager***. Der Recovery-Manager legt Sicherungskopien von Datenbeständen an und protokolliert Transaktionen in sog. *Logfiles* mit, so dass

- Ergebnisse abgeschlossener Transaktionen nach Fehlern wieder in die Datenbank eingespielt werden können,
- Änderungen in der Datenbank, die durch nicht abgeschlossene Transaktionen bewirkt wurden, rückgängig gemacht werden können,
- Im Falle von Speicherfehlern (Platten) der jüngste konsistente Zustand der Datenbank wiederhergestellt werden kann.

Die Eigenschaften einer Transaktion kann man mit dem ACID-Prinzip beschreiben. ACID steht für **a**tomicity, **c**onsistency, **i**solation, **d**urability:

Unteilbarkeit (atomicity)
: Eine Transaktion ist eine unteilbare Verarbeitungseinheit; sie wird entweder ganz oder überhaupt nicht ausgeführt.

Konsistenz (consistency)
: Eine korrekte Ausführung der Transaktion führt die DB von einem konsistenten zu einem konsistenten Zustand.

Isolation (isolation)
: Eine Transaktion muss so ablaufen, als sei sie die einzige im System. Zwischenzustände (die ja inkonsistent sein können) dürfen für andere Transaktionen nicht sichtbar sein.

Dauerhaftigkeit (durability)
: Ergebnisse einer erfolgreich beendeten Transaktion sind dauerhaft, d.h. überleben jeden nachfolgenden Fehler.

Es ist Aufgabe des ***Transaktionsmanagement,*** für die Unteilbarkeit und Isolation ablaufender Transaktionen zu sorgen. In der Regel bedient sich das Transaktionsmanagement hierfür verschiedener Typen von *Sperren*, mittels derer Objekte für andere Transaktionen vorübergehend nicht zugreifbar gemacht werden. Es übernimmt von sich aus das Setzen und Freigeben der Sperren, der Anwendungsprogrammierer muss dies nicht programmieren. Auf diese *Synchronisationsverfahren* gehen wir hier jedoch nicht weiter ein. Das Transaktionsmanagement muss natürlich mit dem Recovery-Manager zusammenspielen.

Beginn und Ende von Transaktionen

Im SQL-92 Standard gab es kein explizites Statement für den Beginn einer Transaktion. Transaktionen werden laut Standard implizit begonnen bei Befehlen, die eine Transaktion begründen, wie SELECT und INSERT, jedoch nicht bei Statements, die keine Transaktion begründen wie CONNECT, das nur eine Verbindung zu einer Datenbank herstellt.

Wenn sie nicht schon begonnen hat, starten folgende Statements eine Transaktion:

- Jedes SQL-Schema Statement ALTER, CREATE, DROP sowie GRANT UND REVOKE

- Die SQL-Daten Statements OPEN, CLOSE, FETCH, INSERT, UPDATE, DELETE, FREE LOCATOR, HOLD LOCATOR
- Eines der neuen SQL:1999 Transaktions-Statements: START TRANSACTION, COMMIT AND CHAIN, ROLLBACK AND CHAIN
- Das SQL-Kontroll Statement RETURN, falls es die Abarbeitung einer Subquery bewirkt und keine Transaktion aktiv ist.

Der SQL:1999 Befehl START TRANSACTION kann benutzt werden, um eine Transaktion explizit zu beginnen und dem System den Zugriffsmodus, den ISOLATION LEVEL und die Diagnostics Area mitzuteilen. Der Befehl SET TRANSACTION startet keine Transaktion, sondern teilt dem System nur die o.a. Parameter der folgenden Transaktion mit.[23] Das erfolgreiche Ende einer Transaktion wird dem DBMS durch COMMIT oder COMMIT WORK oder COMMIT WORK AND NO CHAIN mitgeteilt. Soll eine weitere Transaktion unmittelbar folgen, wird der Befehl COMMIT (WORK) AND CHAIN benutzt. Bei Commit muss das DBMS die Ergebnisse der Transaktion dauerhaft machen, und es kann nach Durchführung von Commit die Ergebnisse der Transaktion für andere Transaktionen sichtbar machen. Nach Commit ist das Rückgängigmachen einer Transaktion durch das DBMS nicht mehr möglich, dies muss dann per Anwendungsprogramm getan werden („Stornierung“ im kommerziellen Bereich).

Stellt das Anwendungsprogramm einen Fehler in der Transaktion fest oder einen Zustand, in dem die Transaktion nicht zu Ende geführt werden soll, so kann die Transaktion mit ROLLBACK (WORK) beendet werden. Bei ROLLBACK setzt das DBMS die Transaktion zurück, d.h. alle von der Transaktion in die Datenbank eingebrachten Veränderungen werden auf den Zustand vor START TRANSACTION bzw. dem impliziten Start der Transaktion zurückgesetzt. ROLLBACK kann ebenfalls mit AND NO CHAIN oder AND CHAIN benutzt werden.

Im Unterschied zu diesem Selbstabbruch der Transaktion können natürlich Fehler auftreten, bei denen das Anwendungsprogramm die Kontrolle verliert. Hierzu gehören Hardware- und Softwarefehler, aber auch beispielsweise *Deadlock,* ein Zustand, in dem Transaktionen wechselseitig aufeinander warten, um auf Objekte zugreifen zu können, so dass keine Transaktion mehr weiterarbeiten kann. In diesen Fällen muss das DBMS von sich aus je nach Fall einzelne, mehrere oder alle Transaktionen zurücksetzen, die noch kein COMMIT durchgeführt haben.

Eine Transaktion könnte folgende Struktur aufweisen:

```
START TRANSACTION
        READ WRITE
        ISOLATION LEVEL REPEATABLE READ
        DIAGNOSTICS SIZE 10
                <Code in Wirtssprache und SQL>
                If condition then ROLLBACK else ...
                <Code in Wirtssprache und SQL>
COMMIT
```

23 Zu Zugriffsmodi, Isolationsleveln und Diagnostics Area verweisen wir auf Kurseinheit 4 des Kurses 1665 oder /MELT02/ S.512 ff oder /GULU99/ S. 691 ff.

Wird START TRANSACTION ohne Parameter angegeben, oder START TRANSACTION nicht benutzt, wird eine Standardtransaktion gestartet, die sowohl lesenden als auch schreibenden Zugriff erlaubt, mit dem höchsten Isolationslevel SERIALIZABLE arbeitet und eine Diagnostic Area mit Standardgröße anlegt.

Im SQL:1999 Standard wurde das Konzept der Savepoints eingeführt, mit dem Transaktionen mit ROLLBACK nicht nur komplett, sondern bis zu definierten Zeitpunkten zurückgefahren werden können.

Beispiel 3.41a:
Wir betrachten das Einfügen eines neuen Mitarbeiters, der ein neues Projekt in die Firma mitbringt und an einem weiteren Projekt mitarbeitet, das dadurch einen neuen Schwerpunkt bekommt.

```
START TRANSACTION
        INSERT INTO ANGEST...
SAVEPOINT ANGEST_GESICHERT
        INSERT INTO PROJEKT...
SAVEPOINT PROJEKT_GESICHERT
        INSERT INTO ANG_PRO...
        INSERT INTO ANG_PRO...
SAVEPOINT BEZIEHUNG_GESICHERT
        update PROJEKT – neue Beschreibung...
```

Es können nun folgende ROLLBACKs erfolgen:
ROLLBACK TO SAVEPOINT BEZIEHUNG GESICHERT
Das UPDATE Statement wird zurückgefahren.
ROLLBACK TO SAVEPOINT PROJEKT_GESICHERT
Das UPDATE Statement wird zurückgefahren.
Die INSERT Statements in die Tabelle ANG_PRO werden zurückgefahren.
ROLLBACK TO SAVEPOINT ANGEST_GESICHERT
Das UPDATE Statement wird zurückgefahren.
Die INSERT Statements in die Tabelle ANG_PRO werden zurückgefahren.
Das INSERT Statement, das ein neues Projekt in die Tabelle PROJEKT einfügt, wird zurückgefahren.
ROLLBACK
Die gesamte Transaktion wird zurückgefahren.

Übung 3.16 Part I

... das DBMS läuft

```
START TRANSACTION
        INSERT INTO ANGEST
        VALUES ( 200, ´MEIERLING´, ´HAGEN´, ´INGENIEUR´, 5000 , 3 )
        INSERT INTO ANGEST
        VALUES ( 201, ´MÜLLER´, ´HAGEN´, ´INGENIEUR´, 5000 , 3 )
        INSERT INTO ANGEST
        VALUES ( 202, ´MAIEN´, ´HAGEN´, ´INGENIEUR´, 5000 , 3 )
COMMIT
```

```
START TRANSACTION
        UPDATE ANGEST
        SET GEHALT = GEHALT + 500
        WHERE ANGNR = 200
COMMIT

START TRANSACTION
        UPDATE ANGEST
        SET GEHALT = GEHALT  + 500
        WHERE ANGNR = 201
ROLLBACK

START TRANSACTION
        UPDATE ANGEST
        SET GEHALT = GEHALT  + 500
        WHERE ANGNR = 202
... Fehler, Stromausfall
```

Die DB wird wieder hochgefahren. Welche Gehälter haben die Angestellten mit den Angestelltennummern 200, 201 und 202? Sind alle Transaktionen verloren gegangen?

Übung 3.16 Part II

Der Angestellte Nr. 117 verlässt die Firma und beteiligt sich daher auch nicht mehr an Projekten. Kein Mitarbeiter übernimmt seine Aufgaben. Alle Projekte, in denen er Projektleiter war, werden aufgelöst. Wir treffen außerdem die Annahme, dass Projekte, in denen er Projektleiter war, keine weiteren Mitarbeiter haben, weil er keine Leitungsbefugnisse hatte.

Deshalb wird folgende Transaktion durchgeführt:

```
START TRANSACTION
        DELETE FROM ANGEST
                WHERE       ANGNR=117

        DELETE FROM PROJEKT
                WHERE       P_LEITER=117

        DELETE FROM ANG_PRO
                WHERE       ANGNR=117

COMMIT
```

Frage:

a) Gegen welche der vier Transaktionseigenschaften (ACID) würde man verstoßen, wenn diese drei DELETE-Befehle nicht in einer zusammengefassten, sondern in drei einzelnen Transaktionen durchgeführt werden würden? Die Reihenfolge der Befehle bleibt bestehen.

b) Welche Umstellung der Reihenfolge der DELETE-Befehle ist vorteilhafter für Referenz-Integrität (Jeder Fremdschlüssel verweist auf einen bestehenden Primärschlüssel)?

c) Was könnte zum Zeitpunkt „t“ passieren, wenn man anstelle des oben genannten Vorschlages folgende Transaktionen hätte? Stellen Sie sich vor, dass mehrere Benutzer gleichzeitig auf die Datenbank Zugriff haben.
Wie kann man das Problem zum Zeitpunkt „t“ verhindern?

```
START TRANSACTION
    DELETE FROM PROJEKT
            WHERE       P_LEITER=117

    DELETE FROM ANG_PRO
            WHERE       ANGNR=117

COMMIT
```

Zeitpunkt “t“

```
START TRANSACTION
      DELETE FROM ANGEST
            WHERE       ANGNR=117
COMMIT
```

3.3.3 Kopplung von SQL an eine Programmiersprache

SQL kann nicht nur als eigenständige Sprache, sondern auch zusammen mit einer Wirtssprache verwendet werden. Von den Herstellern relationaler Datenbanksysteme werden eine Reihe von Programmiersprachen unterstützt, beispielsweise C#, C++ oder JAVA.

Es existieren verschiedene Kopplungsmöglichkeiten. Sie können für die verschiedenen SQL-Sprachkonstrukte als Prozeduren zur Verfügung gestellt werden, die von einem Programm in der Wirtssprache direkt aufgerufen werden.

Durchgesetzt hat sich allerdings die Kopplungsart, bei der SQL-Ausdrücke direkt in das Programm aufgenommen werden. Diese Kopplungsart nennt man *Embedded SQL*. Aufgrund der SQL-Befehle kann ein solches Programm nicht direkt von einem Compiler für die Wirtssprache übersetzt werden. Zuerst muss dieses Pro-

Embedded SQL

gramm durch einen *Precompiler* in ein syntaktisch korrektes Programm der Wirtssprache umgesetzt werden, woraufhin es dann mit einem Compiler für die Wirtssprache weiter übersetzt werden kann.

Wir erläutern diese Kopplungsart an einigen stark vereinfachten Beispielen.

Beispiel 3.42 (Embedded SQL)

Das folgende Programmstück in einer Pseudosprache liest einen Namen ein und löscht dann das Tupel mit diesem Namen aus der Relation ANGEST entsprechend **Beispiel 3.4** („//“ leitet Kommentare ein):

```
program LöscheAngest;
begin
   EXEC SQL BEGIN DECLARE SECTION
        variable name : string;          // Deklariere "name" vom Typ string
   EXEC SQL END DECLARE SECTION

   name := read;          // lese einen Namen vom Terminal ein
                          // und weise ihn der Variablen "name" zu

   EXEC SQL EXECUTE IMMEDIATE
        DELETE FROM ANGEST WHERE NAME = :name;
                // ":name" ist in der SQL-Anweisung durch den
                // Doppelpunkt als Variable aus der Wirtssprache
                // zu erkennen.

program end.
```

In den ersten drei Zeilen wird dem Precompiler mitgeteilt, dass die Programmvariable „name“ auch in einem SQL-Befehl benutzt werden soll. Der Precompiler benötigt hier vor allem den Typ der Variablen, um in anderen SQL-Anweisungen eventuell geeignete Typkonvertierungen vornehmen zu können.

In der nächsten Zeile wird ein Name eingelesen und in der Variablen „name“ abgelegt. In den beiden letzten Zeilen schließlich werden in der Relation ANGEST alle diejenigen Tupel gelöscht, die für das Attribut NAME den Wert besitzen, der sich in der Programmvariablen „name“ befindet.

Wir haben im obigen Beispiel bewusst nicht eine SELECT-Abfrage betrachtet, weil SELECT-Anweisungen einer besonderen Behandlung bedürfen. Das Problem besteht darin, dass eine relationale Abfragesprache als Ergebnis einer Abfrage wiederum eine Relation (also eine Menge von Tupeln) liefert, während man in der Wirtssprache nur ein Tupel nach dem anderen sinnvoll betrachten kann (*one-tupel-at-a-time*). Um diesen Konflikt zu lösen, wurde das Cursor-Konzept eingeführt. Einen *Cursor* kann man sich vorstellen als einen Zeiger, mit dem die Ergebnisrelation einer Abfrage Tupel für Tupel durchlaufen werden kann. Eine übliche Programmsequenz kann dann folgendermaßen skizziert werden:

Cursor

(1) „Deklariere Programmvariablen a_1, ..., a_n."

(2) EXEC SQL DECLARE C CURSOR FOR „gewünschte SELECT-Anweisung"

(3) EXEC SQL OPEN C

(4) while "Cursor noch nicht vollständig durchlaufen" do
(4.1) EXEC SQL FETCH C INTO :a_1, ..., :a_n
"Verarbeitung der gefundenen Werte"
end while

(5) EXEC SQL CLOSE C

In (1) werden n Variablen deklariert. Die Deklaration geschieht ganz analog zu dem Beispiel 3.42. In (2) wird ein Cursor für die gewünschte SQL-Anweisung deklariert. In (3) wird diese Abfrage durchgeführt, wobei der Cursor auf das erste Tupel der Ergebnisrelation gesetzt wird. Die nachfolgende while-Schleife (4) durchläuft nacheinander alle Tupel der Ergebnisrelation. Durch (4.1) werden die in der SELECT-Klausel der Abfrage angegebenen Attribute den Programmvariablen a_1, ..., a_n zugewiesen. Hier muss natürlich eine Variable für jedes Attribut in der SELECT-Klausel angegeben werden. Die letzte Anweisung (5) schließt den Cursor.

Dynamisches SQL

Wird embedded SQL so wie bisher beschrieben eingesetzt, dann müssen sowohl die durch ein Programm betrachteten Relationenschemata, als auch die im Programm benötigten Abfragen zur Übersetzungszeit des Programms bekannt sein. *Dynamisches SQL* erlaubt Programmen darüber hinaus, SQL-Anweisungen zur Laufzeit zu erzeugen und vom DBMS ausführen zu lassen. Bei der Erstellung des Programm muss nicht einmal das zugrundeliegende Datenbankschema bekannt sein – alle Statements können, ggf. nach Abfrage des Katalogs, „ad hoc" generiert und an das DBMS zur Ausführung weitergegeben werden. Im folgenden Beispiel skizzieren wir eine Anwendung, in der eine SQL-Abfrage dynamisch, d.h. zur Laufzeit des Anwendungsprogramms, nach den Vorgaben eines Benutzers, zusammengestellt und dann erst ausgeführt wird.

Beispiel 3.43: (Anwendung für dynamisches SQL)

Das Landesregister für archäologische Funde in Schleswig-Holstein soll wissenschaftliche Untersuchungen, die Erstellung von Katalogen und andere Recherchen unterstützen. Nach den Wünschen von Fachleuten sollen Listen der vorhandenen Funde erstellen werden. Aufgrund der Anzahl der Objekte und der vielfältigen Ordnungskriterien genügen Listen der Art „Von 10000 a.d. bis 7000 a.d., aus Stein" nicht. Bei komplexen Objekten gibt es beispielsweise Unter- und Oberteile, die wiederum besonderen Bedingungen genügen:

FINDE MITTELALTERLICHE SCHLAGWAFFEN (OBEROBJEKT) MIT EINEM SCHMUCK-BESATZ (UNTEROBJEKT) MIT OST-EUROPÄISCHER HERKUNFT.

Ferner gibt es zwar einige gemeinsame Eigenschaften aller Objekte, aber zur Bestimmung verschiedener Spezialisierungen (Waffen, Schmuck, Kleidung, Hausrat usw.) sind teilweise besondere Kriterien wesentlich. In SQL ergeben sich daraus geschachtelte Abfragen oder Joins (für Unter- und Oberobjekte), die verschiedene Tabellen (allgemeines Objekt oder Spezialisierung) einbeziehen. Hinzu kommt, dass sich auch die Attribute der Ergebnisrelation je nach Abfrage ändern. Die Experten für Archäologie beherrschen i.d.R. weder SQL noch kennen sie sich mit dem DB-Schema des Registers aus.

Deshalb wird in Zusammenarbeit mit einigen Fachleuten ein Programm entwickelt, mit dem solche komplexen Anfragen am Bildschirm ohne DV-Kenntnisse erstellt werden können. Einstiegspunkt ist beispielsweise eine Bildschirmmaske mit den gemeinsamen Attributen aller im Register erfassten Objekte. Für diese können je Attribut verschiedene Werte oder Intervalle eingegeben werden und diese Angaben „per Mausklick“ logisch verknüpft werden. Es kann eine Spezialisierung gewählt werden und für deren besondere Attribute können ebenfalls Werte festgelegt werden. Dann können optional Bedingungen an Unter- oder Oberteile gestellt werden, die wiederum (rekursiv) genauso behandelt werden, wie das Ausgangsobjekt.

Sollten nun statische SQL-Abfragen in das Programm eingebettet werden, so müssten eine oder mehrere entsprechende „Generalabfragen“ formuliert werden, die alle diese Listen ausgeben könnten. Man kann sich leicht klar machen, dass es nur durch „Tricks“ möglich ist, die logischen Verknüpfungen (Schlüsselworte AND, OR, NOT) oder die Einbeziehung verschiedener möglicher Spezialisierungen dabei variabel zu halten. Insbesondere die möglichen Abfragen nach Unter- oder Oberobjekten (Rekursion mit variabler Tiefe) können nicht einfach vorab „codiert“ werden.

Dynamisches SQL hingegen bietet die nötige Flexibilität. Während der Benutzer die verschiedenen Bildschirmmasken ausfüllt, setzt das Anwendungsprogramm eine entsprechende SQL-Abfrage folgendermaßen zusammen: Jede ausgefüllte Bildschirmmaske mit Attributwerten wird vom Programm in den Teil einer WHERE-Klausel umgesetzt:

```
SELECT Herkunft, Epoche, Lagerort
FROM Objekt
WHERE Epoche = 'Mittelalter'
```

Wird eine Spezialisierung ausgewählt, so wird die SELECT-Klausel um die speziellen Attribute ergänzt, die entsprechende Tabelle zur FROM-Klausel hinzugefügt und in der WHERE-Klausel mit der Tabelle der allgemeinen Objekte verbunden und die Bedingungen an die speziellen Attribute werden angehängt:

```
SELECT ..., Transport, Bedienung
FROM ..., Waffe
WHERE ... AND Objekt.ID = Waffe.ID AND Bedienung = 'Schlagen'
```

Wird schließlich ein Unter- oder Oberteil beschrieben, dann wird das als geschachtelte Abfrage eingebaut:

```
... EXISTS(  SELECT *
             FROM Objekt UT ...
             WHERE UT.Oberteil=Objekt.ID ...)
```

Alle diese Angaben werden vom Programm zu einem String zusammengestellt, der am Ende der Bearbeitung die gewünschte SQL-Abfrage enthält, etwa:

```
SELECT Herkunft, Epoche, Lagerort, Transport, Bedienung
FROM Objekt, Waffe
WHERE  Epoche = 'Mittelalter' AND
       Objekt.ID = Waffe.ID AND
       Bedienung = 'Schlagen' AND
       EXISTS(  SELECT *
                FROM Objekt UT, Schmuck
                WHERE  UT.Oberteil = Objekt.ID AND
                       UT.ID = Schmuck.ID AND
                       UT.Herkunft = 'Osteuropa')
```

Diese Abfrage wird an das DBS übergeben. Das liefert nicht nur das Ergebnis (wie üblich via Cursor) zurück, sondern auch die Anzahl der Attribute der Ergebnisrelation und deren Wertebereiche. So kann das Anwendungsprogramm daraus eine formatierte Liste erzeugen.

Die Benutzung von dynamischem SQL ist bei weitem komplexer als die von statisch codierten SQL-Abfragen bei embedded SQL, da ohne Precompiler und feststehendes DB-Schema die Erkennung von Fehlern und die anwendungsorientierte Strukturierung der Ergebnisse nun allein dem Anwendungsprogramm unterliegt.

Dynamisches SQL und Embedded SQL wird im fünften Teil des SQL:1999 Standards SQL/Bindings beschrieben. /ANSI99/. Eine genauere Beschreibung finden Sie in /MELT02/.

Literaturverzeichnis

/ANSI99/ American National Standards Institute Inc. Hg., American National Standard for Information Technology – Database Languages – SQL – Parts 1-5, American National Standards Institute, New York, 1999, ISO/IEC 9075-1:1999, ISO/IEC 9075-2:1999, ISO/IEC 9075-3:1999, ISO/IEC 9075-4:1999, ISO/IEC 9075-5:1999

/BECK04/ Becking, Dominic: SQL, die Sprache relationaler Datenbanken. Elektronischer Kurs, Fernuniversität in Hagen, 2004.

/COD70/ Codd, E.F.: A relational model for large shared data banks. Comm. ACM 13:6, pp 377 - 387. 1970

/DATE04/ Date, C. J.: An Introduction to Database Systems 8th ed., Addison-Wesley, 2004, 839 S., ISBN 0-201-82458-2

/GULU99/ Gulutzan, P., Pelzer, T.: SQL-99 Complete, Really. R&D Books, Lawrence KS. 1990 (1078 S., ISBN 0-87930-568-1)

/MELT02/ Melton, J., Simon, A. R.: SQL:1999 Understanding Relational Language Components. Morgan Kaufman Publishers, San Francisco. 2002. (983 S., ISBN 0-321-19784-4)

/MELT03/ Melton, J., Advanced SQL:1999 Understanding Object-Relational and Other Advanced Features, Morgan Kaufman Publishers, San Francisco; 2003, ISBN 1-55860-677-7

/ULL89/ Ullman, J.: Principles of Database and Knowledgebase Systems, Vol. 1 and 2, Computer Science Press, 1989

Lösungen zu den Übungsaufgaben

Übung 3.1:

Verbund/Join:

Seien $R(R_1, ..., R_n)$ und $S(S_1, ..., S_m)$ Relationen.

Sei $\{R_1, ..., R_n\} \cap \{S_1, ..., S_m\}$ die leere Menge,
$i \in \{1, ..., n\}$, $j \in \{1, ..., m\}$ mit $dom(R_i) = dom(S_j)$ und
θ ein zweistelliger Operator auf $dom(R_i)$, der einen Wahrheitswert liefert.

Dann heißt $V = R \bowtie_{Ri\,\theta\,Sj} S$ der *Verbund(join) von R und S bzgl. der Verbundbedingung* $R_i\ \theta\ S_j$ und ist folgendermaßen definiert:

$v = (r_1, ..., r_n, s_1, ..., s_m) \in V$, falls
$(r_1, ..., r_n) \in R$, $(s_1, ..., s_m) \in S$ und $r_i\ \theta\ s_j$

Übung 3.2:

Die letzten Zeilen der Definition des natürlichen Verbunds müssen ersetzt werden durch:

...und ist folgendermaßen definiert:

$v = (r_1, ..., r_n, s_{l1}, ..., s_{l\,m-k}) \in V$, falls
$(r_1, ..., r_n) \in R$, $(s_1, ..., s_m) \in S$ und $(r_{i1}, ..., r_{ik}) = (s_{j1}, ..., s_{jk})$

Übung 3.3:

Schnittmenge:

Seien $R(R_1, ..., R_n)$ und $S(S_1, ..., S_m)$ vereinigungsverträgliche Relationen.

Dann heißt $C = R \cap S$ die *Schnittmenge von R und S* und ist folgendermaßen definiert:

$C = R - (R - S)$

Übung 3.4:

a) Gib für alle Angestellten, die an einem Projekt mitarbeiten, ihren Namen und die Projektnummer aus.

b) Gib Name und Beruf aller Projektleiter aus.

c) Gib für alle Projektleiter ihren Namen und den Namen ihres Projektes sowie den Arbeitszeitanteil, den sie für das Projekt aufwenden, aus.

d) Gib die Namen aller Personen aus, die mehrere Projekte leiten.

Erläuterung:
Die Relationenalgebra kann nicht „zählen“ - es gibt keinen Operator, der dies leistet. Deshalb suchen wir für jedes Tupel aus PROJEKT alle anderen Tupel, die

in P_LEITER übereinstimmen (NATJOIN), nicht aber in PNR (Umbenennung vor NATJOIN und JOIN mit P1.PNR<>P2.PNR). Eine solche Kombination ergibt sich, wenn ein Projektleiter für zwei (oder mehr) Projekte verantwortlich ist. Um die Namen zu erhalten, wird ein Verbund mit ANGEST (mit P_LEITER=ANGNR) und dann eine Projektion auf NAME durchgeführt. Dabei werden dann auch gleich Doppelnennungen von Projektleitern mit mehr als zwei Projekten eliminiert.

Übung 3.5:

a) $\pi_{ABT\_NR}(\sigma_{PNR = 23}(ANGEST \bowtie ANG\_PRO))$

b) Eine mögliche Lösung:

$\pi_{NAME,ANGNR}$
(
(ANGEST ⋈ ANG_PRO)
⋈
$\pi_{PNR}(\sigma_{NAME = \text{'Kaufmann'}}(ANGEST \bowtie ANG\_PRO))$
)

Erläuterung:

(1) (ANGEST ⋈ ANG_PRO)
liefert alle Angestellten in Verbindung mit ihren Projekten.

(2) $\pi_{PNR}(\sigma_{NAME = \text{'Kaufmann'}}(ANGEST \bowtie ANG\_PRO))$
liefert die Projekt-Nummern der Projekte, an denen Frau Kaufmann mitarbeitet.

(3) Der natural join von (1) und (2) liefert alle Angestellten, die an denselben Projekten wie Frau Kaufmann mitarbeiten; Frau Kaufmann selbst befindet sich auch darunter.

(4) Die letzte Projektion auf (NAME,ANGNR) liefert die Namen und Angestelltennummern der so gefundenen Angestellten.

Übung 3.6:

Ja!

Alternative:

RANGE ANG_PRO X
RANGE ANG_PRO Y
$\{(ANGEST.NAME) \mid \exists X (X.ANGNR = ANGEST.ANGNR \wedge \exists Y(Y.PNR = X.PNR \wedge Y.ANGNR = 10)) \wedge ANGEST.ANGNR \neq 10 \}$

Übung 3.7:

RANGE ANG_PRO X
{ (ANGEST.NAME) | ¬ ∃X(X.ANGNR = ANGEST.ANGNR)}

Übung 3.8:

a) RANGE ANG_PRO X
{(ANGEST.ABT_NR) | ∃X(X.ANGNR = ANGEST.ANGNR ∧
X.PNR = 23)}

b) Eine mögliche Lösung:

RANGE ANG_PRO X
RANGE ANG_PRO Z
RANGE ANGEST Y
{ (ANGEST.NAME, ANGEST.ANGNR) |
∃X ∃Y ∃Z (Y.NAME = 'KAUFMANN' ∧
Z.ANGNR = Y.ANGNR ∧
X.PNR = Z.PNR ∧
ANGEST.ANGNR = X.ANGNR)}

Übung 3.9:

Wird für eine Relation in der FROM-Klausel keine Tupelvariable explizit deklariert, dann kann ihr eigener Name wie eine Tupelvariable verwendet werden. (Der Name der Relation wird also quasi implizit als Tupelvariable deklariert.) Also kann eine der beiden Tupelvariablen P1 und P2 aus Beispiel 3.23: entfallen und stattdessen der Name der Relation PROJEKT verwendet werden:

```
SELECT DISTINCT NAME
FROM ANGEST, PROJEKT, PROJEKT AS P2
WHERE  ANGEST.NR = PROJEKT.ANGEST_NR AND
       ANGEST.NR = P2.ANGEST_NR AND
       PROJEKT.NR != P2.NR
```

Übung 3.10:

a) SELECT DISTINCT ANGEST.NAME, ANG_PRO.NR
FROM ANGEST, ANG_PRO
WHERE ANGEST.ANGNR = ANG_PRO.ANGEST_NR

b) SELECT DISTINCT NAME, BERUF
FROM ANGEST, PROJEKT
WHERE ANGEST.NR = PROJEKT.P_LEITER

c) SELECT DISTINCT NAME, P_NAME, PROZARB
FROM PROJEKT, ANGEST, ANG_PRO
WHERE PROJECT.P_LEITER = ANGEST.ANGNR
AND ANGEST.ANGNR = ANG_PRO.ANGNR
AND PROJECT.PNR = ANG_PRO.PNR

Übung 3.11:

SQL-Abfragen entsprechen eher Ausdrücken im Relationenkalkül. Wie diese haben auch SELECT-Anweisungen deklarativen Charakter, d.h. sie beschreiben das Ergebnis („Was?“), während mit der relationalen Algebra außerdem weitgehend die Berechnung des Ergebnisses beschrieben wird („Wie?“).

Entsprechend lassen sich in der Struktur der Ausdrücke Parallelen aufzeigen:

Aufbau eines Ausdrucks im Tupelkalkül:

RANGE <*Relationen und Tupelvariablen*>, ...
{ <*weitere Relationen*>,<*Attributnamen*> | <*Bedingung*> }

Aufbau einer SELECT-Anweisung:

SELECT <*Attributnamen*>
FROM <*Relationen und Tupelvariablen*>, ..., <*weitere Relationen*>
WHERE <*Bedingung*>

Eine ähnlich „einfache“ Übereinstimmung mit einem Ausdruck der Relationenalgebra kann nicht angegeben werden.

Übung 3.11a:

```
SELECT P_NAME, ANG_PRO.ANGNR, ANGEST.NAME
FROM (PROJEKT LEFT OUTER JOIN ANG_PRO
ON PROJEKT.PNR = ANG_PRO.PNR) LEFT OUTER JOIN ANGEST
ON ANG_PRO.ANGNR = ANGEST.ANGNR
```

Übung 3.12:

Differenz ohne EXCEPT:

```
SELECT DISTINCT NAME
FROM ANGEST
WHERE NOT EXISTS (  SELECT NAME
                    FROM KUNDE
                    WHERE NAME = ANGEST.NAME)
```

Durchschnitt ohne INTERSECT:

```
SELECT DISTINCT NAME
FROM ANGEST
WHERE EXISTS (  SELECT NAME
                FROM KUNDE
                WHERE NAME = ANGEST.NAME)
```

Bzgl. der Beispieldaten wäre die Angabe von DISTINCT nicht erforderlich, um das gleiche Ergebnis zu erzielen. Aber angenommen, es gäbe zwei Angestellte namens 'MÜLLER' bzw. 'SCHMIDT', dann wären die Ergebnisse nicht mit denen aus Beispiel 3.28 identisch.

Übung 3.13:

Mit MAX-Funktion:

```
SELECT MAX(GEHALT)
FROM ANGEST
WHERE BERUF = 'PROGRAMMIERER'
```

Ohne Aggregierungsfunktion:

```
SELECT DISTINCT GEHALT
FROM ANGEST
WHERE  BERUF = 'PROGRAMMIERER' AND
       GEHALT >= ALL ( SELECT GEHALT
                       FROM ANGEST
                       WHERE BERUF = 'PROGRAMMIERER')
```

Zusatzaufgabe: Ausdruck der Relationenalgebra:

1. π_{GEHALT}
2. ($\sigma_{Beruf = 'Programmierer'}$ (ANGEST)
3.)
4. -
5. π_{GEHALT}
6. (π_{GEHALT}
7. ($\sigma_{Beruf = 'Programmierer'}$ (ANGEST)
8.)
9.
10. $\bowtie_{Gehalt < VGL}$
11. π_{VGL}
12. ($\sigma_{Beruf = 'Programmierer'}$ ($\rho_{VGL \leftarrow Gehalt}$ (ANGEST))
13.)
14.)

Erläuterung:
Alle Gehälter von Programmierern aus ANGEST (Zeilen 6-8) werden mit allen Gehältern VGL von Programmierern aus ANGEST (Zeilen 11-13) verglichen, wobei diejenigen ausgewählt werden, welche kleiner als irgendein anderes Programmierer-Gehalt sind (JOIN-Bedingung in Zeile 10). Nur das größte Gehalt (oder die größten Gehälter) werden so nicht erfasst! Wenn also dieses Ergebnis von der Menge aller Gehälter abgezogen (Zeilen 1-4) wird, dann bleibt das größte Gehalt übrig.

Übung 3.14:

```
SELECT SUM(PROZ_ARB)
FROM ANG_PRO
WHERE PNR >= 10 AND PNR < 20
GROUP BY PNR
HAVING COUNT(*) > 10
```

Übung 3.15:

a)
```
SELECT ARTNR, ARTBEZ
FROM ARTIKEL
WHERE VERKAUFSPREIS < 100
```

b)
```
SELECT ARTIKEL.ARTNR, ARTBEZ
FROM ARTIKEL, PROD_GRUPPE
WHERE ARTIKEL.ARTNR =  PROD_GRUPPE.ARTNR
                       AND GRUPPENNR = 93
```

c)
```
SELECT GRUPPEN_NR, AVG(EINKAUFSPREIS),
FROM PROD_GRUPPE, ARTIKEL
WHERE PROD_GRUPPE.ARTNR = ARTIKEL.ARTNR
GROUP BY GRUPPENNR
```

d)
```
CREATE VIEW PG_93_97 (  ARTNR, ARTBEZ,
                        EINKAUFSPREIS, VERKAUFSPREIS) AS
SELECT *
FROM ARTIKEL
WHERE  ARTIKEL.ARTNR IN
       (SELECT ARTNR
        FROM PROD_GRUPPE
        WHERE GRUPPENNR IN (93,97))
```

Übung 3.16 Part 1

Angestellter 200 hat am Ende ein Gehalt von 5500 €.
Angestellter 201 hat am Ende immer noch ein Gehalt von 5000 €.
Angestellter 202 hat am Ende immer noch ein Gehalt von 5000 €
Allerdings wird die Applikation im Falle des Stromausfalls auf das nicht durchgeführte COMMIT aufmerksam und kann z.B. später versuchen, die Transaktion doch noch durchzuführen.
Aufgrund der Dauerhaftigkeitseigenschaft von Transaktionen (Durability, das „D“ von ACID) sind die Ergebnisse aller abgeschlossenen Transaktionen bestehen geblieben. Die letzte Transaktion wurde nicht mit COMMIT abgeschlossen, befindet sich in einem Zwischenzustand, darf daher nicht sichtbar sein und muss von der Applikation wiederholt werden.

Übung 3.16 Part II

a) Nach dem Löschen des Angestellten 117 ist die Datenbank inkonsistent. In den Tabellen ANG_PRO und PROJEKT ist die Angestelltennummer 117 von einem Angestellten, der gar nicht mehr existiert! Verstoß gegen Consistency, das „C“ von „ACID“.

b) Bevor man ein Primärschlüssel-Tupel löscht, sollte man alle Tupel mit einem Fremdschlüssel, der auf diesen Primärschlüssel zeigt, löschen. Daher ist diese Reihenfolge „besser“:

```
START TRANSACTION
    DELETE FROM ANG_PRO
        WHERE    ANGNR=117

    DELETE FROM PROJEKT
        WHERE    P_LEITER=117

    DELETE FROM ANGEST
        WHERE    ANGNR=117
COMMIT
```

c) Zum Zeitpunkt „t“ kann eine Transaktion einer anderen Applikation dem Angestellten ein weiteres Projekt zuteilen (INSERT ...).

Applikation „Delete Angestellten“	Applikation Projektzuteilung
START TRANSACTION DELETE FROM ANG_PRO WHERE ANGNR=117 DELETE FROM PROJEKT WHERE P_LEITER=117 COMMIT	
Zeitpunkt „t“	START TRANSACTION INSERT INTO PROJEKT VALUES ... COMMIT
START TRANSACTION DELETE FROM ANGEST WHEREANGNR=117 COMMIT	

Dieses Tupel wird nicht mehr gelöscht. Der Versuch, den Angestellten Nr. 117 zu löschen, würde eine inkonsistente DB zurücklassen, daher sollte ein korrektes, gut programmiertes DBMS das zweite COMMIT nicht durchführen und den Löschvorgang des Angestellten 117 rückgängig machen. Eine dumme Applikation wird wiederholt versuchen, den Angestellten zu löschen (nicht aber das Löschen der Projekte des Angestellten, das wurde ja mit COMMIT korrekt durchgeführt). Das wird möglicherweise zu einem wiederholten Rückgängigmachen von Seiten des DBMS führen (da das neu zugeteilte Projekt eine Referenz auf den Angestellten hat) und so weiter.

Wären die zwei DELETE-Transaktionen zu einer einzigen Transaktion zusammengefasst, bestünde das Problem nicht (aufgrund von Atomicity, das „A“ von ACID und Isolation, das „I“ von ACID). Werden die zusammengefassten DELETE-Befehle mit COMMIT beendet bevor die INSERT Transak-

tion zum Zuge kommt, wird die INSERT-Transaktion nicht durchgeführt, da sie eine inkonsistente DB hinterlassen würde. Wird die INSERT-Transaktion vorher durchgeführt, so löschen die DELETE-Befehle die Projekteintragung. Die INSERT-Anweisung trifft nicht auf einen Zwischenzustand der zusammengefassten DELETE-Transaktion, da die Zwischenzustände aufgrund der Isolationseigenschaft von Transaktionen nicht sichtbar sind.

Anhang: Beispieltabellen

(Trennen Sie diese Seite bitte heraus, um die Beispiel-Schemata und -Daten immer bei der Hand zu haben.)

ANGEST	ANGNR	NAME	WOHNORT	BERUF	GEHALT	ABTNR
	112	MÜLLER	KARLSRUHE	PROGRAMMIER	4500	3
	205	WINTER	HAGEN	ANALYTIKER	7800	3
	117	SEELER	MARBURG	INGENIEUR	6000	5
	198	SCHMIDT	KARLSRUHE	KAUFMANN	7500	4
	⋮	⋮	⋮	⋮	⋮	⋮

PROJEKT	P_NAME	PNR	P_BESCHR	P_LEITER
	DATAWAREHOUSE	12	...	205
	INTRANET	18	...	117
	PROJEKT 2000	17	...	198
	VU	33	...	198
	⋮	⋮	⋮	⋮

ANG_PRO	PNR	ANGNR	PROZ_ARB
	12	205	100
	18	117	20
	33	117	80
	17	198	30
	18	198	70
	17	112	100
	⋮	⋮	⋮

KUNDE	KDNR	NAME	WOHNORT	TÄTIG_ALS
	743	KLÖTERJAHN	HAMBURG	KAUFMANN
	801	LEANDER	DAVOS	ARZT
	324	OSTERLOH	DAVOS	VERWALTUNG
	227	SCHMIDT	KARLSRUHE	KAUFMANN
	⋮	⋮	⋮	⋮