Kapitel 1

Wiederholung 3. Jahrgang

Dieses Kapitel stellt nur eine Wiederholung mit Ergänzungen des 3. Jahrgangs dar.

1.1 ER-Modellierung

siehe Unterricht

1.2 Umwandlung ER-Modell ins Relationenmodell

siehe Unterricht

1.3 Normalformen

Die Kriterien eines guten Datenbankentwurfs sind einerseits möglichst geringe Redundanz, andererseits die Vermeidung von Einfüge-, Lösch- und Änderungsanomalien. Um Redundanzen und Anomalien zu vermeiden, führen wir den Begriff der Normalformen für Datenbankschemata ein. Eine Normalform ist eine Einschränkung auf dem Datenbankschema, die uns hilft, unerwünschte Eigenschaften der Datenbank zu verhindern. Ziel der Normalisierung ist ein Entwurf, der mindestens die Dritte Normalform (3NF) erfüllt. Erste und Zweite Normalform sind nur ein Zwischenschritt in dieser Entwicklung, sie helfen jedoch beim Verständnis der schrittweisen Verbesserung des Entwurfs.

Normalisierung ist der Prozess der Verfeinerung eines relationalen Schemas, um Seiteneffekte, Redundanzen und dadurch verursachte Mehrdeutigkeiten und Inkonsistenzen zu vermeiden.

Dualitätsprinzip besagt, dass es möglich sein muss, nach der Normalisierung die ursprüngliche Relation wiederherzustellen.

1.3.1 Anomalien und Dekomposition

Aufgrund schlechter konzeptioneller Entwürfe können nicht zusammengehörende Informationen in einer Relation stehen. Diese fehlerhaften Relationen können in der Praxis zu sogenannten Anomalien führen.

Anomalien sind Verarbeitungsfehler durch die Manipulation von unkorrekten Relationen. Man unterscheidet:

- **Änderungs-Anomalie**: Bei Datenänderungen werden nicht alle Entitäten bei der Änderung berücksichtigt.
- **Einfüge-Anomalie**: Wurden nicht zusammengehörende Informationen in einer Relation vermischt, so können Probleme bei der Eingabe neuer Entitäten auftreten. So können in derartigen Fällen neue Entitäten erst eingetragen werden, wenn unbeteiligte Entitäten oder neue Relationen vorhanden sind.
- **Lösch-Anomalie**: Durch das Mischen nicht zusammengehörender Informationen können beim Löschen von Entitäten unbeteiligte Entitäten ebenso gelöscht werden.

Um derartige Anomalien zu vermeiden, müssen die Relationen gegebenenfalls zerlegt werden. Dieser Vorgang wird **Dekomposition** oder Zerlegung genannt. Die Vorschrift, wie Relationen zu zerlegen sind, um Anomalien auszuschließen, wird **Normalisierung** genannt. Die zugehörigen Regeln heißen **Normalformen**.

1.3.2 Erste Normalform

Ziel Bereinigt nicht-einfache Wertebereiche in Relationen, wobei unter nicht-einfachen Wertebereichen Attribute verstanden werden, die Wiederholungsgruppen darstellen.

Definition Eine Relation liegt in der ersten Normalform (1NF) vor, wenn jeder Attributwert eine atomare, nicht weiter zerlegbare Dateneinheit ist.

Nicht in 1NF

Eltern		
Vater	**Mutter**	**Kinder**
Johann	**Martha**	**{Else, Lucia}**
Johann	**Maria**	**{Theo, Josef}**
Heinz	**Martha**	**{Cleo}**

In 1NF

Eltern		
Vater	**Mutter**	**Kind**
Johann	**Martha**	**Else**
Johann	**Martha**	**Lucia**
Johann	**Maria**	**Theo**
Johann	**Maria**	**Josef**
Heinz	**Martha**	**Cleo**

Überführung in 1NF

1. Für jede Wiederholungsgruppe eine neue Tabelle anlegen.
2. Nicht atomare Attribute in der alten Tabelle auflösen.
3. In jeder Tabelle einen geeigneten Primärschlüssel festlegen, falls der ursprüngliche Primärschlüssel nicht eindeutig ist.

Anomalien vor Überführung in 2NF

Änderungs-Anomalie (Update-Anomalie): Wird eine Änderung in redundanten Wertebereichen vorgenommen, muss jedes Tupel, dass die redundanten Daten enthält, geändert werden.

Einfüge-Anomalie (Insert-Anomalie): Eine Einfügung eines neuen Wertes in ein Attribut eines zusammengesetzten Schlüssels kann erst erfolgen, wenn auch die anderen Schlüsselfelder Werte erhalten. (Es wird kein Nullwert in Primärschlüsselattributen zugelassen!)

Lösch-Anomalie (Delete-Anomalie): Wird der letzte Wert eines Schlüsselattributes in zusammengesetzten Schlüsseln gelöscht, gehen die Werte des gesamten Datensatzes verloren, da alle Schlüsselattribute mit Werten vorhanden sein müssen.

Beseitigung dieser Anomalien durch die 2. Normalform.

1.3.3 Zweite Normalform

Ziel Attribute, die nur von einem Teil des Primärschlüssels abhängig sind, werden behandelt.

Definition Eine Relation liegt in der zweiten Normalform vor, wenn sie in der 1NF ist und jedes Nichtschlüsselattribut voll funktional abhängig vom Primärschlüssel ist.

Nicht in 2NF

Studenten_Vorlesungen			
MatrNr	**VorlNr**	**Name**	**Semester**
22120	**500**	**Steiner**	**10**
22150	**500**	**Huber**	**5**
22150	**345**	**Huber**	**5**
22170	**450**	**Simon**	**3**
22170	**452**	**Simon**	**3**
22170	**670**	**Simon**	**3**

Begründung

Kandidatenschlüssel:
$MatrNr, VorlNr$
Funktionale Abhängigkeit:
$MatrNr \rightarrow Name, Semester$

Name und Semester sind somit nicht voll funktional von $MatrNr, VorlNr$ **abhängig.**

Lösung: Man zerlegt die Relation in mehrere Teilrelationen (verlustlos und abhängigkeitstreu), die dann der 2NF genügen:

- hören: [MatrNr, VorlNr]
- Studenten: [MatrNr, Name, Semester]

Kurz: Redundanzen (doppelte Attribute-Einträge) sollen verhindert werden.

Überführung in 2NF

1. Entfernen der Attribute (Spalten), die nicht voll funktional vom Schlüssel abhängig sind.
2. Entfernte Attribute in neue Tabellen zusammenfassen, wobei die jeweils von einem Schlüssel abhängigen Attribute eine gesonderte Tabelle bilden.
3. In den neuen Tabellen werden die entsprechenden Schlüsselattribute hinzugefügt.

Anomalien vor Überführung in 3NF

Änderungs-Anomalie (Update-Anomalie): Transitiv abhängige Wertebereiche müssen in jedem Tupel geändert werden, das die Werte enthält, von denen der zu ändernde Wert abhängt.

Einfüge-Anomalie (Insert-Anomalie): Transitiv abhängige Werte können erst eingetragen werden, wenn die Relation für sie angelegt wurde.

Lösch-Anomalie (Delete-Anomalie): Beim Löschen der Relation entfallen alle abhängigen Werte.

Beseitigung dieser Anomalien durch die 3. Normalform

1.3.4 Dritte Normalform

Ziel Beseitigung von Abhängigkeiten zwischen Nichtschlüsselattributen.

Definition Eine Relation liegt in der dritten Normalform vor, wenn sie sich in der 2NF befindet und jedes Nichtschlüsselattribut nicht transitiv abhängig vom Primärschlüssel ist.

(Transitive Abhängigkeit: Ein Nichtschlüsselattribut ist von einem anderen Nichtschlüsselattribut abhängig.)

Nicht in 3NF

CD_Interpret			
CDId	**Album**	**Interpret**	**Gründungsjahr**
4811	**Not That Kind**	**Anastacia**	**1999**
4823	**Freak Of Nature**	**Anastacia**	**1999**
4712	**Wish You Wher Here**	**Pink Floyd**	**1965**

Probleme:

- Gründungsjahr ist vom Interpret abhängig, nicht von der CDId.
- Gründungsjahr ist mehrfach (redundant) gespeichert.

Lösung:

Modellierung als m:n Beziehung, d.h. Einführen einer Relation CD_Künstler:

- CD: [CDId, Album]
- Künstler: [KId, Interpret, Gründungsjahr]
- CD_Künstler: [CDId, KId]

Überführung in 3NF

1. Entfernen der Attribute (Spalten), die transitiv abhängig vom Schlüssel sind.
2. Anlegen der entfernten Attribute in neuen Tabellen, wobei die Attribute, die vom selben Nichtschlüsselattribut abhängig sind, in eine Tabelle kommen.
3. Hinzufügen des Nichtschlüsselattributes als Schlüssel in der neuen Tabelle.

1.3.5 Boyce-Codd Normalform

Ziel Behandlung der Anomalien in zusammengesetzten Schlüsselkandidaten, bei denen sich die Attribute überlappen (Schlüsselkandidaten haben mindestens ein gemeinsames Attribut).

Definition Eine Relation befindet sich in der BCNF genau dann, wenn sie in der 3NF steht und alle voll funktionalen Abhängigkeiten vom Primärschlüssel ausgehen, (d.h. wenn jede Determinante ein Schlüssel ist).

(Determinante: Eine Determinante ist ein Attribut oder eine Gruppe von Attributen, von der beliebige andere Attribute voll funktional abhängig sind.)

Nicht in BCNF

Kosten_Positionen			
RechnungsNr	**Kostenart**	**KostenartenNr**	**Anzahl**
345	5	12	4
345	6	13	2
456	9	15	8

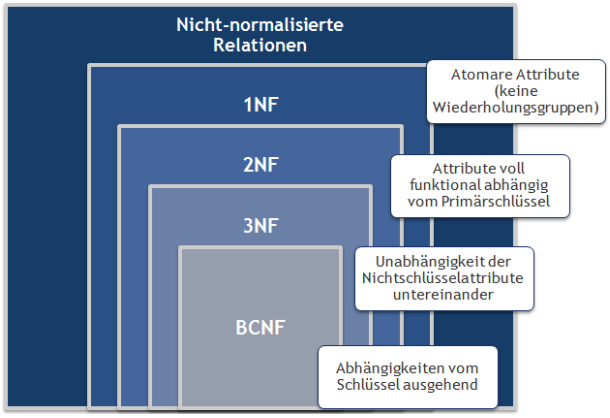
Probleme:

- RechnungsNr und Kostenart sind Schlüssel.
- KostenartenNr ist eindeutig für jede Kostenart, folglich wären RechnungsNr und KostenartenNr ebenfalls Schlüsselkandidat.
- Relation nicht in BCNF, da Kostenart abhängig ist.

Lösung:

- Kostenart und KostenartenNr in eigene Relation: Kosten: [Kostenart, KostenartenNr]
- In Positionen-Tabelle nur Kostenart speichern: Positionen: [RechnungsNr, Kostenart, Anzahl]

Man sollte die Normalisierung als Feinabstimmung im konzeptuellen Entwurf verwenden. Es sollten aus dem ER-Modell nahezu ausschließlich Relationen in 3NF als Ergebnis der Transformation ins relationale Modell entstehen.



Anomalien beruhen auf der Tatsache, dass nicht zusammenpfassende Informationen zusammen gespeichert wurden. Deshalb zerlegt man ein Schema "sinnvoll" in Teilschemata.

Es existieren zwei Korrektheitskriterien für die Zerlegung von Relationenschemata:

- Verlustlosigkeit (Verlust von Information)
- Abhängigkeitstreue (Verlust von Metainformation)

Defintion (Verlustlosigkeit)

[Intuitiv] Die in der Ausprägung R des Schemas $\Re$ enthaltenen Informationen müssen aus den Ausprägungen $R_1, ..., R_n$ der neuen Schemata $\Re_1, ..., \Re_n$ rekonstruierbar sein.

[Formal] Gegeben ein Schema $\Re = \Re_1 \cup \Re_2$, mit den Ausprägungen R, $R_1 = \Pi_{\Re_1}(R)$ und $R_2 = \Pi_{\Re_2}(R)$. Die Zerlegung von $\Re$ in $\Re_1$ und $\Re_2$ ist verlustlos, falls für jede mögliche (gültige) Ausprägung von R von $\Re$ gilt: $R = R_1 \bowtie R_2$

Eine Zerlegung von R in R_1 und R_2 ist verlustlos, wenn die Joinattribute in einer der Teilrelationen Schlüssel sind.

Definition (Abhängigkeitstreue). Die auf $\Re$ geltenden funktionalen Abhängigkeiten müssen auf die Schemata $\Re_1, ..., \Re_n$ übertragbar sein.

[Formal] Gegeben ein Schema $\Re = \Re_1 \cup ... \cup \Re_n$. Die Zerlegung von $\Re$ in $\Re_1$ und $\Re_2$ ist abhängigkeitstreu, wenn $F_{\Re} \equiv (F_{\Re_1} \cup ... \cup F_{\Re_n})$ bzw. $F^+_{\Re} = (F_{\Re_1} \cup ... \cup F_{\Re_n})^+$

Wie kann ein Relationenschema "sinnvoll" in Teilrelationen zerlegt werden?

Gegeben: Relationenschema $\Re$ mit FDs F

Gesucht: Zerlegung in Teilschemata $\Re_1, ..., \Re_n$, für die gilt:

- Zerlegung in $\Re_1, ..., \Re_n$ ist verlustlos,
- Zerlegung in $\Re_1, ..., \Re_n$ ist abhängigkeitstreu,
- alle $\Re_1, ..., \Re_n$ sind in dritter Normalform.

Lösung: Synthesealgorithmus

Synthesealgorithmus

1. Bestimme kanonische Überdeckung F_c zu F.
2. Für jede funktionale Abhängigkeit $\alpha \to \beta \in F_c$:
   (a) Erstelle ein Relationenschema $\Re_\alpha = \alpha \cup \beta$
   (b) Ordne $\Re_\alpha$ die FDs $F_\alpha = \{\alpha' \to \beta' \in F_c | \alpha' \cup \beta' \subseteq \Re_\alpha\}$ zu. (d.h. aus jeder FD wird ein eigenes Schema; Berücksichtigung der kanonischen Überdeckung, um nicht zu viele Teilschemata zu generieren)
3. Falls eines der in Schritt 2 erzeugten Teilschemata einen Schlüssel von $\Re$ bzgl. F_c enthält $\Rightarrow$ fertig
   Andernfalls: Wähle einen Schlüssel $k \in \Re$ aus und definiere folgendes zusätzliches Schema: $\Re_k = k$ mit $F_k = \{\}$ (d.h. Schema zum Verknüpfen der Teilschemata wird zusätlich erzeugt)
4. Eliminiere die in einem anderen Schemata $\Re_{\alpha'}$ enthaltenen Schemata $\Re_\alpha$ (d.h. Kürzen von überflüssigen Schemata)

Beispiel $\Re$ ={PersNr, Name, Rang, Raum, Ort, Straße, PLZ, Vorwahl, BLand, EW, Landesregierung} = {P,N,R,Z,O,S,Plz,V,B,E,L} und F ={$P \to NRZOSB$, $Z \to P$, $SBO \to Plz$, $OB \to EV$, $B \to L$, $Plz \to BO$}

Lösung

1. Kanonische Überdeckung ist bereits gegeben
2. Generierung der Teilschemata und Zuordnung aller FDs:
   $\{PNRZOSB\}$ es gelten: $P \to NRZOSB$ und $Z \to P$
   $\{ZP\}$ es gelten: $Z \to P$ und $P \to Z$
   $\{SBOPlz\}$ es gelten: $SBO \to Plz$ und $Plz \to BO$
   $\{OBEV\}$ es gilt: $OB \to EV$
   $\{BL\}$ es gilt: $B \to L$
   $\{PlzBO\}$ es gilt: $Plz \to BO$
3. Enthält eines der Teilschemata einen Schlüssel von $\Re$ bzgl. F? Ja, P war Schlüssel und ist in $\{PNRZOSB\}$ enthalten $\Rightarrow$ fertig
4. Schemaelimination:
   $\{ZP\}$ ist schon in $\{PNRZOSB\}$ enthalten $\Rightarrow$ kürzen
   $\{PlzBO\}$ ist schon in$\{SBOPlz\}$ enthalten $\Rightarrow$ kürzen

5. Ergebnis:
   Professoren: {[PersNr, Name, Rang, Raum, Ort, Straße, BLand]}
   PLZListe: {[Straße, Ort, BLand, PLZ]}
   Staedte: {[Ort, BLand, EW, Vorwahl]}
   Regierung: {[BLand, Landesregierung]}

Beispiel $\Re = (ABCDEF)$, $F = \{A \rightarrow EC,\ BC \rightarrow F,\ D \rightarrow B\}$

Lösung

1. Kanonische Überdeckung ist bereits gegeben
2. Generierung der Teilschemata und Zuordnung aller FDs:
   $\Re_1 = (AEC)$ es gilt: $A \rightarrow EC$
   $\Re_2 = (BCF)$ es gilt: $BC \rightarrow F$
   $\Re_3 = (DB)$ es gilt: $D \rightarrow B$
3. Enthält eines der Teilschemata einen Schlüssel von $\Re$ bzgl. F? Nein, d.h. Schlüssel von $\Re$:
   AD hinzufügen von $\Re_4 = (AD)$
4. Schemaelimination:
   Nichts zu eliminieren
5. Ergebnis:
   $\Re = (AEC) \cup (BCF) \cup (DB) \cup (AD)$

1.4 SQL - DDL (Data Definition Language)

1.4.1 Datentypen in SQL

SQL kennt verschiedene Arten von Datentypen, die sie sich auf folgender Seite durchlesen sollen:

http://de.wikibooks.org/wiki/Einf%C3%BChrung_in_SQL:_Datentypen

1.4.2 Anlegen von Tabellen

```
CREATE TABLE-Anweisung :=
   CREATE TABLE Tabellenname
   (   <Spaltendefinition> [, <Spaltendefinition> ...]
       [, <Constraint>]
   );
```

wobei

```
<Spaltendefinition> :=
   Spaltenname <Datentyp>
   [DEFAULT <Ausdruck> ] [NOT NULL ][NULL]
   [, <Constraint>]
```

```
<Constraint> :=
   { [CONSTRAINT Constraintname ]
   | [NULL | NOT NULL]
   | [UNIQUE | PRIMARY KEY]
   | [FOREIGN KEY (Spaltenname [,...] ) ]
   | [REFERENCES Tabellenname ( Spaltenname[,...] ) [ON DELETE CASCADE]]
   | [CHECK (<Checkbedingung>)] }
```

```
<Checkbedingung> :=
   {<Ausdruck> <Vergleichsoperator> <Ausdruck>
   | <Ausdruck> [NOT] BETWEEN <Ausdruck> AND <Ausdruck>
   | <Ausdruck> [NOT] IN ( Konstante [, Konstante ...] )
   | <alphanumerischer Ausdruck> [NOT] LIKE <Schablone>
   | <Ausdruck> IS [NOT] NULL }
```

Anmerkung:

NULL	legt fest, dass eine Spalte NULL-Werte haben kann
NOT NULL	legt fest, dass eine Spalte keine NULL-Werte haben kann
UNIQUE	bestimmt eine oder mehrere Spalten als eindeutige Schlüssel
PRIMARY KEY	bestimmt eine oder mehrere Spalten als Primärschlüssel
FOREIGN KEY	bestimmt eine oder mehrere Spalten als Fremdschlüssel, die der referentiellen Integrität genügen müssen
REFERENCES	identifiziert den Primärschlüssel, der als FOREIGN-KEY festgelegt wurde
ON DELETE CASCADE	Referentielle Integrität wird gesichert, indem automatisch Zeilen, die diese Integrität verletzen, entfernt werden. (Löschen eines Datensatzes führt zum kaskadierenden Löschen der über foreign key constraints verbundenen Datensätze)
CHECK	legt eine Bedingung fest, die jede Zeile der Tabelle erfüllen muss

1.4.3 Ändern von Tabellen

Mit der ALTER-TABLE-Anweisung können nachträglich Spalten in Tabellen verändert, gelöscht oder hinzugefügt werden. Insbesondere können auch CONSTRAINTS nachträglich bearbeitet oder zeitweise außer Kraft gesetzt werden.

```
ALTER-TABLE-Anweisung :=
   ALTER TABLE Tabellenname
   { ADD { (<Spaltendef> [, <Spaltendef> ... ] ) | CONSTRAINT Constraintname }
   | DROP { (<Spaltendef> [, <Spaltendef> ... ] ) | CONSTRAINT Constraintname }
   | MODIFY (<Spaltendef> [, <Spaltendef> ... ] ) }

   [ENABLE | DISABLE ] {
   ALL TRIGGERS
   | UNIQUE (Spaltenname, [, Spaltenname ...])
   | PRIMARY KEY
   | CONSTRAINT Constraintname } ]
```

1.4.4 Löschen von Tabellen

```
DROP-TABLE-Anweisung :=
  DROP TABLE Tabellenname;
```

1.4.5 Aufgabe 9

Entwickeln Sie die CREATE TABLE - Statements für nachfolgende Relationen.

```
Weinsorte   (  WeinsorteName, Anmerkung                                          )
Winzer      (  WinzerCode, Name, Adresse, StartJahr                              )
Ernte       (  WeinsorteName, WinzerCode, ErnteJahr, Menge                       )
Kunde       (  KundenID, Name, Adresse, Kontostand                               )
bevorzugt   (  KundenID, WeinsorteName, WinzerCode                               )
Bestellung  (  KundenID, Nummer, WeinsorteName, WinzerCode, ErnteJahr, Anzahl    )
Geschenk    (  KundenID, Datum, Beschreibung                                     )
```

1.4.6 Aufgabe

Entwickeln Sie die nötigen SQL-Anweisungen um die soeben erstellen Relationen abzuändern:

- Fügen Sie zur Relation Ernte eine CHECK-Klausel hinzu (`Menge>0`)
- Fügen Sie zur Relation Ernte ein Attribut Preis (inkl. CHECK-Klausel) hinzu
  `Ernte (WeinsorteName, WinzerCode, Jahr, Menge, Preis)`
- Entfernen Sie aus der Relation Winzer das Attribut StartJahr
  `Winzer (WinzerCode, Name, Adresse)`
- Fügen Sie zur Relation Bestellung ein Attribut Datum zum Primärschlüssel hinzu
  `Bestellung(Datum, KundenId, Nummer, WeinsorteName, WinzerCode, ErnteJahr, Anzahl)`
- Entfernen Sie aus der Relation bevorzugt das Attribut WinzerCode
  `bevorzugt (KundenId, WeinsorteName)`
  (Welches Problem könnte dabei entstehen?)
- Löschen Sie die Relation Geschenk

1.4.7 Ergänzungen

Beispiel: zyklische Foreign Keys

Es gibt 2 Relationen: `Abteilungen, Mitarbeiter`

Jeder Mitarbeiter ist einer Abteilung zugeordnet, jede Abteilung hat einen Chef (unter den Mitarbeitern)

Problem:

- Create Table: Wie wird der Foreign Key der ersten Tabelle definiert?
- Drop Table: Wie löscht man Tabellen mit zyklischen Foreign Keys?
- Insert: Wie kann man zB neue Abteilungen einfügen?

Lösung:

- Constraints nachträglich einführen mit Alter Table
- Drop Constraint oder Cascade
- Deferred Constraints: Überprüfung erst beim "commit" (Abschluss der Transaktion)

DEFERRED CONSTRAINTS IN ORACLE

```
CREATE TABLE Abteilungen(
  AbtNr INTEGER,
  Chef INTEGER,
  CONSTRAINT pk_abt PRIMARY KEY(AbtNr)
  . . .
);
CREATE TABLE Mitarbeiter(
  PersNr INTEGER,
  Name VARCHAR(30) ,
  AbtNr INTEGER,
  CONSTRAINT pk_mitarb (PersNr),
  CONSTRAINT fk_mitarb_abt  REFERENCES Abteilungen (AbtNr)
  DEFERRABLE INITIALLY DEFERRED
);

ALTER TABLE Abteilungen
   ADD CONSTRAINT fk_abt_mitarb FOREIGN KEY (Chef) REFERENCES Mitarbeiter (PersNr)
            DEFERRABLE INITIALLY DEFERRED;

INSERT INTO Abteilungen VALUES (1001, 102, . . . );
INSERT INTO Abteilungen VALUES (1002, 203, . . . );
INSERT INTO Abteilungen VALUES (1003, 301, . . . );
INSERT INTO Mitarbeiter VALUES (101, . . . , 1001);
INSERT INTO Mitarbeiter VALUES (102, . . . , 1001);
etc.
COMMIT;
DROP TABLE Mitarbeiter CASCADE;
DROP TABLE Abteilungen CASCADE;
```

Beim Löschen der Abteilungen-Tabelle wäre CASCADE nicht nötig gewesen (weil die Mitarbeiter-Tabelle zu diesem Zeitpunkt schon gelöscht ist)

FOREIGN_KEY_CHECKS IN MYSQL

MySQL unterstützt im Zusammenhang mit der Engine InnoDB Foreign Keys und überprüft deren Integrität. MySQL kennt das Attribut DEFERRABLE aber nicht. Die Lösung wäre hier:

```
SET foreign_key_checks = 0;
 ... insert statements ...
SET foreign_key_checks = 1;
```

Kapitel 2

DQL (Data Query Language) und DML (Data Manipulation Language)

2.1 Einfügen von Datensätzen

```
INSERT INTO Tabellenname [(Spaltenname [,Spaltenname ...] ) ]
    VALUES ( <Ausdruck> [, Ausdruck ...] )
```

oder

```
INSERT INTO Tabellenname [(Spaltenname [,Spaltenname ...] ) ]
    <SELECT-Abfrage>
```

2.2 Ändern von Datensätzen

```
UPDATE Tabellenname
    SET <Spaltenname>=<Ausdruck> [,<Spaltenname>=<Ausdruck> ...]
    [WHERE <Bedingung>]
```

2.3 Löschen von Datensätzen

```
DELETE FROM Tabellenname
    [WHERE <Bedingung>]
```

2.4 Abfragen von Datensätzen

```
SELECT [DISTINCT] {   | Spaltenname [AS <Alias>] [, Spaltenname [AS <Alias>]...] }
FROM Tabellenname [<Alias>] [, Tabellenname [<Alias>] ...]
WHERE <Bedingung>
GROUP BY Spaltenname [, Spaltenname ...]
HAVING <Bedingung>
ORDER BY Spaltenname [ ASC | DESC ] [, Spaltenname ...]
```

WHERE-Klausel	Diese Klausel gibt an, welche Zeilen aus den benannten Tabellen in der FROM-Klausel ausgewählt werden. Sie kann benutzt werden, um Joins zwischen mehreren Tabellen einzurichten
GROUP BY-Klausel	Sie können nach Spalten, Aliasnamen oder Funktionen gruppieren. Das Ergebnis der Abfrage enthält eine Zeile für jede unterschiedliche Menge von Werten in den benannten Spalten, Aliasse oder Funktionen. Die sich ergebenden Zeilen werden oftmals auch Gruppen genannt. Für diese Gruppen können Aggregatfunktionen (COUNT, MAX, MIN, SUM, AVG) angewendet werden.
HAVING-Klausel	Diese Klausel wählt die Zeile auf der Grundlage der Gruppenwerte und nicht der einzelnen Zeilenwerte aus. Die HAVING-Klausel kann nur verwendet werden, wenn entweder die Anweisung eine GROUP BY-Klausel hat oder die Auswahlliste nur aus Aggregatfunktionen besteht. Sämtliche Spaltennamen, die in der HAVING-Klausel referenziert werden, müssen entweder in der GROUP BY-Klausel enthalten sein oder in der HAVING-Klausel als ein Parameter für eine Aggregatfunktion verwendet werden.
ORDER BY-Klausel	Diese Klausel sortiert die Ergebnisse einer Abfrage. Jedes Element in der ORDER BY-Liste kann als ASC für aufsteigende Sortierfolge (der Standardwert) oder DESC für absteigende Sortierfolge benannt werden.

Folgende Punkte werden behandelt:

- Abfragen aus einer Tabelle
- Sortierung (auf-, absteigend, mehrere Spalten)
- Umbenennung von Spalten und Tabellen
- Aggregatfunktionen
- WHERE-Bedingungen: Verknüpfungen, NULL-Überprüfung, IN bzw. NOT IN, ...
- Abfragen über mehrere Tabellen
- Joinarten (Cross Join, Natural Join, Equi Join, Theta Join, Outer Join, ...)
- Überprüfungen von Ergebnissen aus Aggregatfunktionen (HAVING - Klausel, ALL, ANY, ...)
- Abbildung einer Division in SQL
- UNION, INTERSECT, EXCEPT

2.4.1 Aggregatfunktionen

Eine Aggregatfunktion berechnet ein einzelnes Ergebnis aus mehreren Eingabezeilen. Zum Beispiel gibt es Aggregatfunktionen, die die Anzahl der Zeilen (`count`), die Summe der Eingabewerte (`sum`), den Durchschnitt (`avg`), die Minimal- (`min`) oder den Maximalwert (`max`) jeweils aus mehreren Zeilen berechnen. Bei `count` ist zu beachten, dass dabei alle Datensätze gezählt werden, bei denen die entsprechende Spalte nicht NULL ist (Ausnahme: `count(*)`)

2.4.2 Subqueries

Subqueries sind eine andere Möglichkeit zur Formulierung von Anfragen, die mehrere Relationen umfassen. Dabei werden mehrere SELECT-Anfragen ineinander geschachtelt. Meistens stehen Subqueries dabei in der WHERE-Zeile.

```
1 SELECT <attr-list>
2 FROM <table-list>
3 WHERE <attribute> <rel> <subquery>;
```

wobei `<subquery>` eine SELECT-Anfrage (Subquery) ist und

- falls `<subquery>` nur einen einzigen Wert liefert, ist `rel` eine der Vergleichsrelationen $=, <, >, <=, >=$,
- falls `<subquery>` mehrere Werte/Tupel liefert, ist `rel` entweder `IN` oder von der Form Φ ANY oder Φ ALL wobei Φ eine der o.g. Vergleichsrelationen ist.

Bei den Vergleichsrelationen muss die Subquery ein einspaltiges Ergebnis liefern, bei `IN` sind im SQL-Standard und in Oracle seit Version 8 auch mehrere Spalten erlaubt.

Man unterscheidet unkorrelierte Subqueries und korrelierte Subqueries: Eine Subquery ist unkorreliert, wenn sie unabhängig von den Werten des in der umgebenden Anfrage verarbeiteten Tupels ist. Solche Subqueries dienen dazu, eine Hilfsrelation oder ein Zwischenergebnis zu bestimmen, das für die übergeordnete Anfrage benötigt wird. In diesem Fall wird die Subquery vor der umgebenden Anfrage einmal ausgewertet, und das Ergebnis wird bei der Auswertung der WHERE-Klausel der äußeren Anfrage verwendet.

Eine Subquery ist korreliert, wenn in sie von Attributwerten des gerade von der umgebenden Anfrage verarbeiteten Tupels abhängig ist. In diesem Fall wird die Subquery für jedes Tupel der umgebenden Anfrage einmal ausgewertet.

Beispiel für eine korrelierte Abfrage:

Es existieren 2 Relationen:

```
City(Name, Country, Population, ....)
Country(Name, Population, Code, ....)
```

Es sollen alle Städte bestimmt werden, in denen mehr als ein Viertel der Bevölkerung des jeweiligen Landes wohnt:

```
1  SELECT Name, Country
2  FROM City
3  WHERE Population * 4 >
4  (SELECT Population
5  FROM Country
6  WHERE Code = City.Country);
```

Das Schlüsselwort `EXISTS` bzw. `NOT EXISTS` bildet den Existenzquantor nach. Subqueries mit `EXISTS` sind i.a. korreliert um eine Beziehung zu den Werten der äußeren Anfrage herzustellen.

```
1  SELECT <attr-list>
2  FROM <table-list>
3  WHERE [NOT] EXISTS
4  (<select-clause>);
```

Beispiel:

Gesucht seien diejenigen Länder, für die Städte mit mehr als 1 Mio. Einwohnern in der Datenbasis abgespeichert sind.

```
1  SELECT Name
2  FROM Country
3  WHERE EXISTS
4  (SELECT *
5  FROM City
6  WHERE Population > 1000000
7  AND City.Country = Country.Code);
```

Subqueries in der FROM-Zeile

Zusätzlich zu den bisher gezeigten Anfragen, wo die Subqueries immer in der WHERE-Klausel verwendet wurden, sind [in einigen Implementierungen; im SQL-Standard ist es nicht vorgesehen] auch Subqueries in der FROM-Zeile erlaubt.

```
SELECT <attr-list>
FROM <table/subquery-list>
WHERE <condition>;
```

Beispiel:

Gesucht ist die Zahl der Menschen, die nicht in den gespeicherten Städten leben.

```
SELECT Population-Urban_Residents
FROM
(SELECT SUM(Population) AS Population
FROM Country),
(SELECT SUM(Population) AS Urban_Residents
FROM City);
```

2.4.3 UNION, INTERSECT

Mit dem `UNION` Befehl kann man die Result Sets von zwei oder mehr `SELECT` kombinieren. Doppelte Werte werden dabei allerdings ignoriert. Bei `UNION` muss man darauf achten, dass die selektierten Spalten beider Tabellen vom gleichen Typ sind.

Die Ergebnismenge bei `INTERSECT` enthält die Schnittmenge der Teilmengen, d.h. die Datensätze müssen in beiden Abfragen enthalten sein. Oft ist dieser Operator auch durch `EXISTS` oder `IN` realisierbar.

2.5 Aufgabe

Entwickeln Sie die nötigen SQL-Anweisungen, um folgende Aufgaben durchzuführen:

- Fügen Sie zur Relation `Kunde` einige Datensätze ein.(Es soll dabei einen Kunden mit einer `ID=1` geben und auch einige, die einen negativen Kontostand haben)
- Erstellen Sie eine neue Relation `Kunde_Mahnung` mit den gleichen Attributen wie die Relation `Kunde`. Fügen Sie anschließend jene Tupel der Relation `Kunde` in die Relation `Kunde_Mahnung` ein, die einen negativen Kontostand besitzen.
- Der Weinhändler möchte jene Kunden, die ihm kein Geld schulden (`Kontostand`$\geq$`0`), belohnen und schreibt ihnen einen Betrag von 10 Euro auf ihrem Kontostand gut.
- Der Kunde mit der `KundenID=1` hat auf die Mahnung des Weinhändlers bezahlt, hat jedoch versichert, dass er nie wieder Kunde bei ihm sein wird. Löschen Sie diesen Kunden aus der Relation `Kunde`.
- Löschen Sie alle Tupel der Relation `Kunde_Mahnung` und danach die gesamte Relation.

2.6 Aufgabe

In einem Weinkeller werden viele Weinflaschen (`WEIN`) gelagert. Jede Weinflasche wurde von einem Winzer (`WINZER`) befüllt und kann anhand der Datenbank leicht in den Regalreihen (`KELLER`) des Kellers gefunden werden. Es kann dabei angenommen werden, dass immer ausreichend Platz in den Regalen des Kellers vorhanden ist. Wird eine Flasche aus dem Keller entfernt, wird ein entsprechender Eintrag (`PROTOKOLL`) vermerkt. Wenn der Eigentümer des Kellers eine Flasche selbst trinkt, wird im Protokoll als Verwendung '`Eigenbedarf`' eingetragen.

2.6.1 Tabellen und deren Inhalt

```
SQL> SELECT * FROM winzer;
+-----+-------------------------+--------------------+------+-----------------+----------+
| wnr | name                    | strasse            | plz  | ort             | telefon  |
+-----+-------------------------+--------------------+------+-----------------+----------+
|   1 | Lackner Tinnacher       | Steinbach 8        | 4567 | Gamlitz         | 1234567  |
|   2 | Weingut Prager          | Weissenkirchen 48  | 3610 | Weissenkirchen  | 1234567  |
|   3 | Weingut Emmerich Knoll  | Unterloiben 10     | 3601 | Unterloiben     | 1234456  |
|   4 | Weingut F.X.Pichler     | Unterloiben 27     | 3601 | Unterloiben     | 11122233 |
|   5 | Weingut Spatlese        | Weintalstrasse 23  | 1136 | Wien            |          |
|   6 | Freie Weingarten Wachau | Kremstalstrasse 23 | 3600 | Krems           | 2304002  |
|   7 | Stiftskellerei          |                    |      |                 |          |
|   8 | Weingut Biegler         | Wienerstrasse 88   | 2352 | Gumpoldskirche  | 54564565 |
+-----+-------------------------+--------------------+------+-----------------+----------+

SQL> SELECT * FROM wein;
+-----+---------------------+---------------+----------+-------+--------+------+----------+
| nr  | bezeichnung         | sorte         | jahrgang | preis | anzahl | wnr  | position |
+-----+---------------------+---------------+----------+-------+--------+------+----------+
|   1 | Riesling Kellerberg | Riesling      |     1999 | 28.00 |     24 |    4 |        1 |
|   2 | Loibenberg          | Gr. Veltliner |     2000 | 19.00 |     36 |    4 |        2 |
|   3 | Ried Kreutles       | Gr. Veltliner |     2000 | 19.00 |     15 |    3 |        4 |
|   4 | Riesling Smaragd    | Riesling      |     2000 | 21.00 |     30 |    2 |        5 |
|   5 | Grauburgunder       | Grauburgunder |     2003 | 16.00 |     72 |    1 |        6 |
|   6 | Morillon            | Chardonnay    |     2003 |  9.00 |     55 |    1 |        7 |
|   7 | Riesling Federspiel | Riesling      |     2003 |  9.90 |     80 |    6 |        3 |
|   8 | Chardonnay          | Chardonnay    |     2003 |  9.00 |     16 |    8 |        8 |
+-----+---------------------+---------------+----------+-------+--------+------+----------+

SQL> SELECT * FROM keller;
+-----+-------+-------+------+
| knr | reihe | regal | fach |
+-----+-------+-------+------+
|   1 |     1 |     1 |    1 |
|   2 |     1 |     1 |    2 |
|   3 |     1 |     1 |    3 |
|   4 |     1 |     2 |    1 |
|   5 |     1 |     2 |    2 |
|   6 |     2 |     1 |    1 |
|   7 |     2 |     1 |    2 |
|   8 |     2 |     2 |    1 |
+-----+-------+-------+------+

SQL> SELECT * FROM protokoll;
+-----+-----+------------+---------------------------+---------+
| pnr | nr  | pDatum     | verwendung                | anzahl  |
+-----+-----+------------+---------------------------+---------+
|   1 |   1 | 2003-01-12 | Geschenk an Herrn Berger  |      12 |
|   2 |   3 | 2003-07-10 | Eigenbedarf               |       2 |
|   3 |   1 | 2003-07-23 | Eigenbedarf               |       4 |
|   4 |   6 | 2003-08-14 | Geschenk (Frau Kunz)      |       6 |
|   5 |   1 | 2003-08-27 | Glasbruch                 |       1 |
|   6 |   4 | 2003-11-03 | Korkgeruch                |       1 |
|   7 |   6 | 2003-11-03 | Eigenbedarf               |       3 |
+-----+-----+------------+---------------------------+---------+
```

2.6.2 Abfragen

1. Geben Sie für die Sorte 'Riesling' die Namen aller Winzer sowie die Flaschenanzahl aus. Sortieren Sie dabei nach der Flaschenanzahl absteigend.

```
+--------------------------+---------+
| name                     | anzahl  |
+--------------------------+---------+
| Freie Weingarten Wachau  |      80 |
| Weingut Prager           |      30 |
| Weingut F.X.Pichler      |      24 |
+--------------------------+---------+
```

2. Ermitteln Sie für jeden Winzer den durchschnittlichen Flaschenpreis und die Gesamtanzahl der Flaschen im Keller. Berücksichtigen Sie dabei nur Winzer, von denen bekannt ist, aus welchem Ort sie kommen. Sortieren Sie die Liste nach dem Preis absteigend.

```
+--------------------------+--------------------+---------------+
| name                     | durchschnittspreis | gesamtanzahl  |
+--------------------------+--------------------+---------------+
| Weingut F.X.Pichler      |              23.50 |            60 |
| Weingut Prager           |              21.00 |            30 |
| Weingut Emmerich Knoll   |              19.00 |            15 |
| Lackner Tinnacher        |              12.50 |           127 |
| Freie Weingarten Wachau  |               9.90 |            80 |
| Weingut Biegler          |               9.00 |            16 |
+--------------------------+--------------------+---------------+
```

3. Geben Sie eine Liste aller Weinbezeichnungen sowie den Namen des erzeugenden Winzers aus, von denen im Jahr 2003 keine Flasche getrunken worden ist (Verwendung in Tabelle Protokoll = 'Eigenbedarf'). Sie brauchen dabei nur Winzer berücksichtigen, von denen mindestens eine Flasche im Keller vorhanden ist.

```
+----+---------------------+--------------------------+
| nr | bezeichnung         | name                     |
+----+---------------------+--------------------------+
|  2 | Loibenberg          | Weingut F.X.Pichler      |
|  4 | Riesling Smaragd    | Weingut Prager           |
|  5 | Grauburgunder       | Lackner Tinnacher        |
|  7 | Riesling Federspiel | Freie Weingarten Wachau  |
|  8 | Chardonnay          | Weingut Biegler          |
+----+---------------------+--------------------------+
```

4. Suchen Sie die Winzer, von denen der Kellereigentümer die meisten Flaschen getrunken (Verwendung in Tabelle Protokoll = 'Eigenbedarf') hat. Geben Sie jeweils den Namen des Winzers sowie die Gesamtkosten des von diesem Winzer konsumierten Weines aus.

```
+---------------------+---------+---------+
| name                | anzahl  | kosten  |
+---------------------+---------+---------+
| Weingut F.X.Pichler |       4 |  112.00 |
+---------------------+---------+---------+
```

5. Geben Sie für jeden Winzer aus,
wie viele günstige (Preis $\leq$ 10 Euro, Preisklasse niedrig),
wie viele im Mittelfeld (10 Euro - 20 Euro, Preisklasse mittel) und
wie viele teure (Preis $>$ 20 Euro, Preisklasse gehoben) Weinflaschen im Keller liegen.

```
+--------------------------+---------+--------------+
| name                     | anzahl  | preisklasse  |
+--------------------------+---------+--------------+
| Freie Weingarten Wachau  |      80 | niedrig      |
| Lackner Tinnacher        |      55 | niedrig      |
| Lackner Tinnacher        |      72 | mittel       |
| Weingut Biegler          |      16 | niedrig      |
| Weingut Emmerich Knoll   |      15 | mittel       |
| Weingut F.X.Pichler      |      24 | gehoben      |
| Weingut F.X.Pichler      |      36 | mittel       |
| Weingut Prager           |      30 | gehoben      |
+--------------------------+---------+--------------+
```

6. Erstellen Sie eine Liste, die angibt, wie viele Flaschen jedes in der Datenbank gespeicherten Winzers sich im Keller befinden. Sortieren Sie dabei nach der Flaschenanzahl absteigend.

```
+--------------------------+---------+
| name                     | anzahl  |
+--------------------------+---------+
| Lackner Tinnacher        |     127 |
| Freie Weingarten Wachau  |      80 |
| Weingut F.X.Pichler      |      60 |
| Weingut Prager           |      30 |
| Weingut Biegler          |      16 |
| Weingut Emmerich Knoll   |      15 |
| Weingut Spatlese         |         |
| Stiftskellerei           |         |
+--------------------------+---------+
```

2.7 DQL - rekursive Abfragen

Klassische Anwendungsfälle für rekursive Anfragen sind das Vorgesetzten- oder das Stücklisten-Problem. Beide Datenbäume zeichnen sich durch die unterschiedlich tiefen Zweige aus. Bei den Vorgesetzten sind z.B. abhängig von der Position eines Mitarbeiters unterschiedlich viele Vorgesetzten-Ebenen gegeben. Während es für einen "normalen" Mitarbeiter vielleicht 5-6 Hierarchieebenen bis zum Chef sind, sind es für das mittlere Management nur 2-3. Ein anderes Charakteristikum dieser Bäume ist, dass die maximale Tiefe in der Regel unbekannt ist bzw. variieren kann. Rekursive Anfragen zeichnen sich dadurch aus, ohne Programmänderungen in der Lage zu sein, sich unbekannte Tiefen zu erarbeiten.

Beispiel:
In einem Unternehmen gibt es mehrere Hierarchiestufen. Ganz oben stehen die Oberbosse. Diese sind die Vorgesetzten der Mittelbosse, die wiederum die Unterbosse leiten. Jeder Unterboss hat dann noch seine Arbeiter.

SELECT * FROM ORGA;

persnr	name	persnrchef
1	Oberboss 1	<null>
2	Oberboss 2	<null>
3	Oberboss 3	<null>
11	Mittelboss 11	1
12	Mittelboss 12	1
13	Mittelboss 13	1
21	Mittelboss 21	2
22	Mittelboss 22	2
23	Mittelboss 23	2
31	Mittelboss 31	3
32	Mittelboss 32	3
33	Mittelboss 33	3
111	Unterboss 111	11
112	Unterboss 112	11
113	Unterboss 113	11
114	Unterboss 114	11
121	Unterboss 121	12
122	Unterboss 122	12
123	Unterboss 123	12
211	Unterboss 211	21
212	Unterboss 212	21
213	Unterboss 213	21
311	Unterboss 311	31
312	Unterboss 312	31
313	Unterboss 313	31
321	Unterboss 321	32
1111	Arbeiter 1111	111
1112	Arbeiter 1112	111
1113	Arbeiter 1113	111
1114	Arbeiter 1114	111
1121	Arbeiter 1121	112
1122	Arbeiter 1122	112
1211	Arbeiter 1211	121
1212	Arbeiter 1212	121
1231	Arbeiter 1231	123
1232	Arbeiter 1232	123
1311	Arbeiter 1311	131
1312	Arbeiter 1312	131
2111	Arbeiter 2111	211
2112	Arbeiter 2112	211
2113	Arbeiter 2113	211
3111	Arbeiter 3111	311
3112	Arbeiter 3112	311
3311	Arbeiter 3311	331
3211	Arbeiter 3211	321

Abbildung 2.1:

Fragestellung: Welche Untergebenen hat "Oberboss 1"? Der klassische SELECT:
SELECT B.* FROM ORGA A INNER JOIN ORGA B ON A.PERSNR=B.PERSNRCHEF WHERE A.NAME='Oberboss 1'; scheitert, da hier nur die direkten Untergebenen erfasst werden.

persnr	name	persnrchef
11	Mittelboss 11	1
12	Mittelboss 12	1
13	Mittelboss 13	1

Abbildung 2.2:

Nicht erfasst werden hingegen die Untergebenen der Untergebenen und die Untergebenen der Untergebenen der Untergebenen usw.
Hier liegt eine typische Anwendung für rekursives SQL vor.

Zum Aufbau des Statements beantworten wir nun einfach die obigen Fragen:

1. Welche Spalten sollen in meiner Ergebnismenge auftauchen und/oder werden für die Rekusionsbedingung benötigt? PERSNR , NAME
2. Wie lautet der SELECT für den Satz, von dem die Rekursion ausgehen soll?

```
SELECT PERSNR , NAME
FROM ORGA
WHERE PERSNRCHEF = 1;
```

3. Wie lautet die Rekursionsbedingung? verbal: Wer ist Untergebener eines in der vorherigen Rekursion ermittelten Chefs.

```
neuhinzuzufuegen.PERSNRCHEF = gerade_ermittelt.PERSNR
```

mit diesem Wissen ist das Aufbauen des SQL-Statements nun nicht mehr schwierig:

```
WITH hilfstabelle ( PERSNR , NAME ) AS
(
SELECT PERSNR, NAME FROM ORGA WHERE PERSNRCHEF = 1
UNION ALL
SELECT A.PERSNR , A.NAME
FROM ORGA A INNER JOIN hilfstabelle B ON A.PERSNRCHEF = B.PERSNR
)
SELECT * FROM hilfstabelle;
```

Ergebnis:

persnr	name
11	Mittelboss 11
12	Mittelboss 12
13	Mittelboss 13
111	Unterboss 111
112	Unterboss 112
113	Unterboss 113
114	Unterboss 114
121	Unterboss 121
122	Unterboss 122
123	Unterboss 123
131	Unterboss 131
1111	Arbeiter 1111
1112	Arbeiter 1112
1113	Arbeiter 1113
1114	Arbeiter 1114
1121	Arbeiter 1121
1122	Arbeiter 1122
1211	Arbeiter 1211
1212	Arbeiter 1212
1231	Arbeiter 1231
1232	Arbeiter 1232
1311	Arbeiter 1311
1312	Arbeiter 1312

Abbildung 2.3:

Wäre als Ursprungs-SELECT ein:

```
SELECT PERSNR, NAME FROM ORGA
WHERE NAME = 'Oberboss 1'
```

gewählt worden, so wäre das Ergebnis um eine Zeile (nämlich um den Oberboss selbst) größer gewesen. Es ist übrigens auch möglich (und durchaus sinnvoll), die Anzahl der Iterationsstufen mitzuzählen. Hier z.B: Auf welcher Hierarchiestufe steht ein Mitarbeiter (Oberboss=1), alle anderen 2,3,4 ...

Die erweiterte Query sieht dann so aus:

```
WITH hilfstabelle ( PERSNR , NAME , STUFE ) AS
( SELECT PERSNR, NAME , 1 AS STUFE FROM ORGA WHERE PERSNR = 1
UNION ALL
SELECT A.PERSNR , A.NAME , B.STUFE + 1 FROM ORGA A
INNER JOIN hilfstabelle B ON A.PERSNRCHEF = B.PERSNR )
SELECT PERSNR , NAME , STUFE FROM hilfstabelle ;
```

Ein derartiges "Mitzählen" ist grundsätzlich sinnvoll um die Anzahl der Iterationen begrenzen zu können. Entsprechende Daten (oder eine fehlerhaft formulierte Bedingung) könnten nämlich durchaus zu einer Endlosschleife führen.

Hat man allerdings mitgezählt, dann kann man nach einer bestimmten Anzahl Iterationen abbrechen.

persnr	name	stufe
1	Oberboss 1	1
11	Mittelboss 11	2
12	Mittelboss 12	2
13	Mittelboss 13	2
111	Unterboss 111	3
112	Unterboss 112	3
113	Unterboss 113	3
114	Unterboss 114	3
121	Unterboss 121	3
122	Unterboss 122	3
123	Unterboss 123	3
1111	Arbeiter 1111	4
1112	Arbeiter 1112	4
1113	Arbeiter 1113	4
1114	Arbeiter 1114	4
1121	Arbeiter 1121	4
1122	Arbeiter 1122	4
1211	Arbeiter 1211	4
1212	Arbeiter 1212	4
1231	Arbeiter 1231	4
1232	Arbeiter 1232	4

Abbildung 2.4:

```
1  WITH hilfstabelle ( PERSNR , NAME , STUFE ) AS
2  ( SELECT PERSNR, NAME , 1 AS STUFE FROM ORGA WHERE PERSNR = 1
3  UNION ALL
4  SELECT A.PERSNR , A.NAME , B.STUFE + 1 FROM ORGA A
5  INNER JOIN hilfstabelle B ON A.PERSNRCHEF = B.PERSNR
6  WHERE B.STUFE < 3 )
7  SELECT PERSNR , NAME , STUFE FROM hilfstabelle ;
```

Bei obiger Abfrage wäre das Ergebnis das gleiche wie bei:

```
1  WITH hilfstabelle ( PERSNR , NAME , STUE ) AS
2  ( SELECT PERSNR, NAME , 1 AS STUFE FROM ORGA WHERE PERSNR = 1
3  UNION ALL
4  SELECT A.PERSNR , A.NAME , B.STUFE + 1 FROM ORGA A
5  INNER JOIN hilfstabelle B ON A.PERSNRCHEF = B.PERSNR )
6  SELECT PERSNR , NAME , STUFE FROM hilfstabelle
7  WHERE STUFE <= 3 ;
```

Allerdings ist erste Query performanter und "ungefährlicher". Bei der zweiten Query wird zuerst alles ermittelt und erst später gefiltert - zu viel Arbeit und es besteht das Risiko der Endlosschleife.

2.8 Abfragenoptimierung

Höhere, nichtprozedurale Abfragesprachen wie z.B. SQL verlangen keine Kenntnisse des Benutzers über die Implementierung, müssen aber zur Ausführung vom System selbst in eine andere Form (z.B. Relationenalgebra) umgesetzt werden. Um eine effiziente Bearbeitung von Abfragen zu erzielen, gibt es eine interne Umformulierung der Abfragen in eine optimierte Form (Query Optimization). Die dabei verwendeten Strategien erzielen keine optimale Lösung, sondern meist nur eine Verbesserung. Im folgenden Abschnitt wird ein Algorithmus vorgestellt, der zur Optimierung verwendet wird.

2.8.1 Logische Abfragenoptimierung

Die aufwendigsten Operationen in relationalen Sprachen sind kartesisches Produkt und Join. Bei einfachster Implementierung eines Joins zwischen A und B ist ein Durchlauf aller Tupel von B für jedes Tupel von A notwendig. Besonders aufwendig werden diese Operationen in verteilten Datenbanken[1].

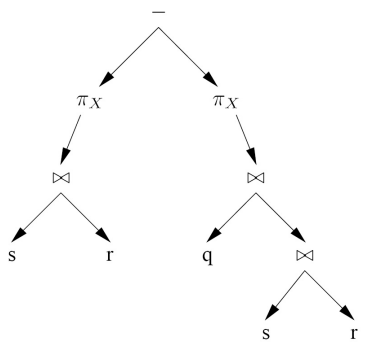
- Die Projektion ist aufwendig, da die durch das Projizieren entstandenen Duplikate entfernt werden müssen.
- Wenn die Selektionen so früh wie möglich durchgeführt wurde, führt das zu kleineren Zwischenresultaten und erleichtert somit die nachfolgenden Operationen.
- Unäre Operationen bedingen je einen Durchlauf aller Tupel, daher sollten mehrere möglichst zusammengezogen werden.
- Gemeinsame Teilausdrücke brauchen nur einmal ausgewertet werden. Der Aufwand kann dadurch reduziert werden.

Der Zeitaufwand für das Untersuchen der verschiedenen Möglichkeiten, eine Abfrage durchzuführen, ist im allgemeinen viel geringer als für das Durchführen einer ineffizienten Abfrage.

Algebraische Optimierung

Bei der algebraischen Optimierung verwenden wir zur Darstellung von Abfragen einen Operatorbaum.

Beispiel:
Wir stellen den Ausdruck $\pi_X(s \rhd\lhd r) - \pi_X(q \rhd\lhd s \rhd\lhd r)$ in einem Operatorbaum dar.



Der Operatorbaum wird Bottom-up ausgewertet. Daher ist es für die Laufzeit nicht günstig, wenn die Join-Operatoren nahe bei den Blättern stehen. Andererseits steigt der Zeit- und Platzaufwand binärer Operationen mit der Anzahl der Tupel und der Anzahl der Attribute in den Argumentrelationen. Eines der Grundprinzipien bei der Optimierung ist daher, die unären Operatoren in Richtung der Blätter des Baumes zu verschieben, um eine möglichst frühzeitige Reduktion der Größe der Operanden von binären Operationen (Join, Differenz etc.) zu bewirken. Außerdem erreichen wir durch die Zusammenfassung unärer Operationen, dass diese in einem Schritt durchgeführt werden. Da Projektion eine Entfernung doppelter Tupel bedingt, ist es besser, die Selektion möglichst vor der Projektion durchzuführen.

[1] Wenn die logisch zusammengehörenden, gemeinsam verwalteten Daten einer Datenbank physisch auf mehrere, in einem Netz verbundene Rechner verteilt sind, spricht man von einer verteilten Datenbank (engl.: distributed database).

Zusammenfassen gleicher Teilausdrücke

Das Zusammenfassen gleicher Teilausdrücke geschieht in einem Baum von unten nach oben. Dabei werden die Ausdrücke schrittweise zusammengefasst. Allerdings können algebraische Transformationen die Existenz gleicher Teilausdrücke verschleiern.

Beispiel
Betrachten wir nochmals den vorigen Operatorbaum. In einem ersten Schritt fassen wir die Blätter s und r zusammen, im zweiten den Join, der die beiden Relationen verbindet. Es folgt, dass wir die Operation $s \bowtie r$ nur einmal durchführen müssen.

Regeln für Join und kartesisches Produkt

Der Join und das kartesische Produkt sind kommutativ und assoziativ.

1. Kommutativität:
   $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$
   $E_1 \times E_2 \equiv E_2 \times E_1$
2. Assoziativität:
   $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$
   $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$

Regeln für Selektion und Projektion

In diesem Abschnitt beschreiben wir, unter welchen Umständen wir Selektion und Projektion in Richtung der Blätter verschieben können.

3. Zusammenfassung von Projektionen:
   Die Zusammenfassung von zwei Projektionen zu einer geht nur unter der Voraussetzung, dass die Menge der Attribute der äußeren Projektion eine Teilmenge der Attribute der inneren Projektion ist.
   Formal: falls $\{A_i | i = 1, ..., n\} \subseteq \{B_i | i = 1, ..., m\}$ so gilt: $\pi_{A_1,...,A_n}(\pi_{B_1,...,B_m}(E)) \equiv \pi_{A_1,...A_n}(E)$
4. Zusammenlegung/Kommutativität von Selektionen:
   $\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$
   $\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_2}(\sigma_{F_1}(E))$
5. Kommutativität Selektion-Projektion:
   Die Projektion ist mit der Selektion eingeschränkt kommutativ. Wenn sich die Bedingung F der Selektion nur auf Attribute Ai bezieht, nach denen projiziert wird, so gilt
   $\pi_{A_1,...A_n}(\sigma_F(E)) \equiv \sigma_F(\pi_{A_1,...A_n}(E))$
   Wenn sich F auf alle Attribute B_j und möglicherweise auf Attribute A_i bezieht, dann gilt
   $\pi_{A_1,...A_n}(\sigma_F(E)) \equiv \pi_{A_1,...,A_n}(\sigma_F(\pi_{A_1,...,A_n,B_1,...B_m}(E)))$
   Das bedeutet, dass wir nur nach jenen Attributen zuerst projizieren dürfen, die wir in der Selektion noch brauchen. Erst in einem zweiten Schritt können wir die gewünschte Projektion durchführen.
6. Kommutativität Selektion-Kartesisches Produkt:
   Die Selektion kommutiert mit dem kartesischen Produkt nur unter der Bedingung, dass sich $F = F_1 \times F_2$ auf die Attribute von E_1 und E_2 beschränkt. Wenn sich F_i nur auf Attribute von E_i beziehen, so gilt $\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$
   Wenn F_1 sich nur auf Attribute von E_1, F_2 aber auf Attribute von E_1 und E_2 bezieht, so gilt:
   $\sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2)$
7. Kommutativität Selektion-Vereinigung:
   $\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$
8. Kommutativität Selektion-Mengendifferenz/Durchschnitt: $\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$
   Daher gilt natürlich analog auch die Kommutativität zwischen Selektion und Durchschnitt (Überlegen Sie sich warum)!

9. Kommutativität Projektion-Kartesisches Produkt: Seien B_i Attribute von E_1, C_i Attribute von E_2 und $\{A_i|i=1,...n\} = \{B_i|i=1,...,m\} \cup \{C_i|i=1,...,k\}$, so gilt:
$\pi_{A_1,...A_n}(E_1 \times E_2) \equiv \pi_{B_1,...,B_m}(E_1) \times \pi_{C_1,...C_k}(E_2)$
10. Kommutativität Projektion-Vereinigung:
$\pi_{A_1,...A_n}(E_1 \cup E_2) \equiv \pi_{A_1,...,A_n}(E_1) \cup \pi_{A_1,...,A_n}(E_2)$

Der Join ist darstellbar als Kombination von kartesischem Produkt, Projektion und Selektion. (Wollen wir zwei Relationen R_1 und R_2 nach dem Attribut A joinen, so müssen wir zuerst das kartesische Produkt der beiden Relationen bilden, wobei wir das Attribut A einmal umbenennen müssen, da das Ergebnis ansonsten zwei gleiche Attribute enthält. Dann selektieren wir jene Tupel, die auf A und dem umbenannten Attribut gleich sind, und zum Schluss projizieren wir das umbenannte Attribut wieder weg.) Daher folgen die Regeln für Kommutativität von Selektion und Join aus den Regeln 4, 5, und 6.

Es gilt keine Kommutativität zwischen Mengendifferenz und Projektion und daher auch keine Kommutativität zwischen dem Durchschnitt und der Projektion (überlegen Sie sich warum)!

Ein einfacher Optimierungsalgorithmus

1. Zerlege Selektionen der Art $\sigma_{F_1 \wedge ... \wedge F_n}(E)$ nach Regel 4 in $\sigma_{F_1}(...(\sigma_{F_n}(E))...)$.
2. Schiebe jede Selektion soweit wie möglich in Richtung Blätter mit den Regeln 4-8.
3. Schiebe jede Projektion soweit wie möglich in Richtung Blätter mit den Regeln 3,5,9 und 10.
4. Fasse alle direkt aufeinanderfolgenden Selektionen und Projektionen zu einer einzigen Selektion, einer einzigen Projektion oder einer Selektion gefolgt von einer Projektion mit den Regeln 3-5 zusammen.

Beispiel:
Betrachten wir die Relationenschemata Restaurant, Speise und Mitarbeiter. Wir wollen folgende Abfrage formulieren: Wie hoch ist das Gehalt und wie ist der Name der Mitarbeiter, die in Restaurants mit mehr als 2 Hauben arbeiten, in denen es Speisen zu mehr als 30 Euro gibt und wie heißen die Restaurants?

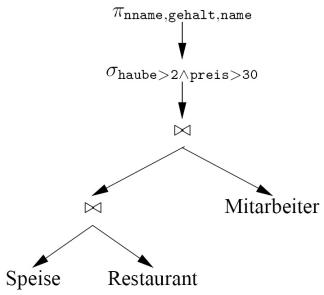
Die Abfrage sieht in SQL wie folgt aus:

```
1  select distinct m.nname, m.gehalt, r.name
2  from mitarbeiter m, restaurant r, speise s
3  where m.rnr = r.rnr and
4        r.haube > 2 and
5        r.rnr = s.rnr and
6        s.preis > 30;
```

In Relationale Algebra übersetzt ergibt sich folgende Abfrage:

$\pi_{nname,gehalt,name}[\sigma_{haube>2 \wedge preis>30}(Mitarbeiter \bowtie Restaurant \bowtie Speise)]$

Der Operatorbaum zu dieser Abfrage sieht wie folgt aus:



Im nächsten Schritt verschieben wir die Selektion nach den Regeln 4-8 so weit wie möglich in Richtung der Blätter, danach die Projektion nach den Regeln 3, 5, 9 und 10. Zum Schluss fassen wir die Kaskaden von Selektionen und Projektionen zusammen (Regeln 3 und 4), was in diesem Beispiel nicht notwendig ist, aber Regel 5 ist zweimal anwendbar, also werden die zwei untersten Projektionen mit den darüberliegenden Selektionen permutiert und mit der nächsten Projektion verschmolzen. Das garantiert uns, dass die Selektion zuerst ausgeführt wird.

Der Operatorbaum nach Verschiebung der Selektion und Projektion und der optimierte Operatorbaum sieht daher wie folgt aus 2.8.1: