Praktikum zur Vorlesung Datenbanken
Handreichung

Ioannis Karatassis, M.Sc.
Alfred Sliwa, M.Sc.

Wintersemester 2019/2020

Datum	
Team (Account)	
Passwort	

Aktuelle Informationen, Ansprechpartner, Materialien und Uploads unter:
`http://www.is.inf.uni-due.de/courses/db_ws18/`

v1.1

1 Ablauf des Praktikums

Eine Praktikumssitzung ist für zwei Stunden angesetzt. Diese Zeit dürfte nicht ausreichen, um alle Aufgaben des Blocks zu lösen. Bitte beachtet, dass – so wie bei einer Vorlesung Selbststudienzeit und bei einer Übung das heimische Bearbeiten von Übungsaufgaben vorgesehen ist – auch für das Praktikum zusätzlicher Zeitaufwand in der Größenordnung von mindestens zwei Stunden pro Woche zur Vor- und Nachbereitung des Stoffs eingeplant werden muss.

Das Praktikum findet in insgesamt zehn Gruppen in Raum LF 230 statt (nur jeweils **ein** Termin muss besucht werden). Die Teilnehmer des Praktikums sind gebeten, Verspätungen zu vermeiden, um nicht wichtige Erklärungen zu verpassen.

Die Abgabe der Aufgaben wird bewertet. Ebenso findet eine bewertete Abnahme des im letzten Arbeitsblock zu erstellenden Software-Projektes statt. Die dabei erreichten Punkte bilden zusammen mit den Klausurpunkten die Gesamtnote für die Veranstaltung „Datenbanken". Wenn nötig, wird zusätzlich ein unbenoteter Schein über den erfolgreichen Besuch des Praktikums ausgestellt. Details zur Benotung wurden bereits in der Vorlesung und in der Übung besprochen.

Im Laufe des Praktikums wird der Umgang mit einem kommerziellen Datenbanksystem anhand von Anfragen geübt und dann eine Fallstudie von der Modellierung bis hin zum Einsatz in zwei Aufgabenblöcken bearbeitet. Die Bearbeitung soll in Kleingruppen von zwei Teilnehmern geschehen. Dabei sollten die Aufgaben **gemeinsam** bearbeitet werden, so dass jeder Teilnehmer einen Einblick in die benutzten Werkzeuge und Sprachen erhält. Ein Praktikumsblock endet damit, dass die Ergebnisse des Blocks über das Webformular hochgeladen wurden (falls in dem Block Abgaben vorgesehen sind). Beide Teilnehmer einer Kleingruppe sollten in der Lage sein, die erarbeiteten Lösungen der Gruppe auf Nachfrage zu erklären bzw. vorzuführen.

1.1 Accounts

Die am Rechner zu bearbeitenden Aufgaben finden an den Maschinen in Raum LF 230 unter einer Linux-Umgebung statt (wir benutzen Ubuntu und damit die grafische Oberfläche Gnome). Eine kurze Einführung wird in der ersten Praktikumssitzung gegeben. Als Datenbanksystem kommt IBM DB2 V9.7 zum Einsatz, das auch nach vorheriger Registrierung kostenfrei von der IBM-Webseite bezogen werden kann. Accounts für jede Kleingruppe werden in der ersten Sitzung ausgegeben.

Jeder Accountname ist nach dem Schema `dbpxxx` aufgebaut, wobei `xxx` die Nummer der Kleingruppe ist. Das ausgegebene Passwort ist in der ersten Sitzung zu ändern, und sollte von beiden Mitgliedern der Kleingruppe sorgfältig gemerkt werden.

Zu jedem Account existiert eine Datenbankmanagerinstanz, die von der Kleingruppe genutzt werden kann. Innerhalb dieser Instanz hat die Gruppe alle notwendigen Rechte, um eigene Datenbanken anzulegen und zu löschen.

1.2 Lernziele

Der Umgang mit einem kommerziellen Datenbanksystem soll anhand von Anfragen über einem vorgegebenen Schema eingeübt werden. Dabei kommen auch fortgeschrittene Konzepte wie Sichten, Rekursion und Trigger zur Anwendung. Zu den SQL-Grundlagen wird auf Folien, Buch und Übungen zur Vorlesung verwiesen.

Der Entwurf und die Implementierung einer Datenbank sollen anhand einer Fallstudie eingeübt werden. Dabei ist zunächst eine konzeptionelle Modellierung mittels eines Entity-Relationship-Diagramms vorzunehmen. Diese wird umgesetzt in ein Datenbankschema, ergänzt um notwendige statische und referentielle Integritätsbedingungen. Die Datenbank ist anschließend mit Beispieldaten zu füllen.

In den letzten Wochen des Praktikums soll die Datenbank aus einer Programmiersprache heraus angesprochen werden. Mit Hilfe von HTML und JSPs bzw. Servlets wird eine einfache Benutzeroberfläche erstellt. Gegebenenfalls ist die Datenbank mit zusätzlichen Daten zu füllen, um alle Funktionalitäten der Anwendung sinnvoll zu testen.

2 Einführung in die Arbeit mit IBM DB2

2.1 Benutzungsmodi

Vor der Arbeit mit DB2 sollte man den Unterschied zwischen der Linux-Shell und der DB2-Shell beachten. Eine Shell ist eine textuelle Benutzerschnittstelle, die Befehle auf der Kommandozeile interpretiert. Wenn Ihr ein Terminalfenster auf den Datenbank-Rechnern öffnet (siehe Aufgabenteil) befindet Ihr Euch standardmäßig zunächst in der Linux-Shell (um genau zu sein, der *bash* oder *Bourne-Again shell*). Diese erlaubt z.B. die schnelle Navigation durch Verzeichnisse oder den Aufruf von Programmen. Die Shell deutet ihre Bereitschaft, Befehle entgegenzunehmen durch ein `$` und einen blinkenden Cursor an. Bei mehrzeiligen Befehlen wird das Dollarzeichen durch ein `>` ersetzt. Shell-Befehle in den Praktikumsunterlagen sind durch das vorangestellte Dollarzeichen kenntlich gemacht. Das Dollarzeichen ist nicht mit einzutippen.

DB2 besitzt ebenfalls eine Shell, die nicht mit der Linux-Shell zu verwechseln ist. DB2-Befehle werden von der Linux-Shell nicht verstanden und umgekehrt. Die DB2-Shell wird im Abschnitt „Interaktiver Modus" erklärt. Sie ist daran zu erkennen, dass sich das Befehlsprompt für die Eingabebereitschaft zu `db2 =>` ändert. Bei mehrzeiligen Befehlen wird das Prompt zu `db2 (cont.) =>`.

2.1.1 Aufruf aus der Linux-Shell

Die einfachste Benutzung von DB2 erfolgt direkt aus der Linux-Shell, indem jedem DB2-Befehl der Programmname `db2` vorangestellt wird. So kann man sich mit dem folgenden Shell-Befehl aus der Linux-Kommandozeile heraus mit einer Datenbank verbinden:

```
$ db2 "CONNECT TO mygeo"
```

Es ist empfehlenswert, grundsätzlich alle derart abgesetzten DB2-Befehle in Klammern zu setzen. Einige in DB2-Befehlen benutzte Zeichen wie $\star$, (oder) sind Sonderzeichen der Shell und werden ohne Klammerung von dieser interpretiert. In den meisten Fällen führt das zu unerwünschten Ergebnissen.

2.1.2 Interaktiver Modus

Neben dem direkten Aufruf mit einem Befehl kann man den Datenbank-Client auch im interaktiven Modus starten. Dies wird erreicht, indem man `db2` ohne Befehl aufruft. Man gelangt dann in die oben beschriebene DB2-Shell. Der Dialog mit der Shell kann mit `terminate` beendet werden.

Standardmäßig muss in diesem Modus jeder Befehl in **einer** Zeile stehen, ein abschließendes Semikolon ist nicht notwendig und führt in den meisten Fällen sogar zu einer Fehlermeldung. Wer lieber Befehle über mehrere Zeilen hinweg formulieren möchte, sollte `db2` mit zusätzlichen Parametern aufrufen: Der folgende Aufruf startet eine DB2-Shell, in der ein Befehl erst dann abgeschlossen ist, wenn eine Zeile mit einem Semikolon endet:

```
$ db2 -t
```

Eine Liste der verfügbaren Parameter beim Aufruf und die Voreinstellungen erhält man mit dem Shell-Befehl

```
$ db2 "LIST COMMAND OPTIONS"
```

Der interaktive Modus von DB2 ist leider sehr rudimentär und daher nur Hartgesottenen zu empfehlen. Insbesondere sind die Möglichkeiten zum Editieren der Befehlszeile stark eingeschränkt. Es ist allerdings möglich, sich den Befehlsverlauf mit Hilfe von

`history` oder `h`

anzeigen zu lassen. Diese zeigt eine nummerierte Liste der zuletzt in der Shell ausgeführten Befehle. Wenn man einen Befehl erneut ausführen möchte kann man dies dann tun, indem man den Befehl `rundcmd` oder kurz `r` verwendet, gefolgt von der Zeilennummer des gewünschten Kommandos, z.B.

`r 2` (um den zweiten Befehl aus dem Befehlsverlauf aufzurufen)

Will man einen früheren Befehl editieren benutzt man `edit` oder `e` gefolgt von der Zeilennummer. Dies startet dann den Standard-Texteditor des Systems. Nach Abschluß des Editierens beendet man den Editor und kann seinen geänderten Befehl nun ausführen.

2.1.3 Batch-Modus

Der komfortabelste und von uns empfohlene Arbeitsmodus ist der Batch-Modus. Im Batch-Modus benutzt Ihr einen Texteditor Eurer Wahl (es stehen eine Reihe zur Verfügung, die auch Syntax-Highlighting für SQL anbieten), um Eure

Befehle in einer Datei zu speichern. Diese Textdatei wird dann beim Aufruf an das Programm `db2` übergeben:

`$ db2 -tvf skript.sql` (führt die Befehle aus der Datei `script.sql` aus)

Dabei ermöglicht der Parameter `t` wie zuvor das Schreiben von SQL-Befehlen über mehrere Zeilen (abgeschlossen durch ein Semikolon). Die Angabe `v` macht das Programm geschwätzig und `f` weist DB2 an, seine Befehle aus der angegebenen Datei zu lesen.

Der Batch-Modus ist insbesondere deshalb günstig, weil Ihr dadurch gleich ein Versuchsprotokoll mit den verwendeten Befehlen gewinnt, das Ihr für Eure Abgabe benutzen könnt. Um Kommentare einzufügen oder einzelne Befehle in der Batch-Datei auszukommentieren, benutzt man zwei Minuszeichen zu Beginn einer Zeile.

Dies ist eine beispielhafte Batch-Datei:

```
-- Batchdatei für Gruppe 99
CONNECT TO mygeo;

SELECT name FROM dbmaster.country;

-- don't forget to disconnect
DISCONNECT mygeo;
```

Wollt Ihr nicht nur die Befehle, sondern auch die Ergebnisse protokollieren, kann die Ausgabe von DB2 einfach in eine Datei umgeleitet werden:

`$ db2 -vtf skript.sql > dateiname.txt`

Das `>` erzeugt dabei eine neue Datei `dateiname.txt` (oder überschreibt eine existierende) mit der Ausgabe des vorangestellt Befehls. Um an eine existierende Datei anzuhängen statt zu überschreiben, könnt Ihr stattdessen `>>` verwenden.

2.2 Grundlegendes

Die grundlegende organisatorische Einheit des Datenbanksystems DB2 ist die Datenbankmanager-Instanz. Eine **Instanz** ist an einen bestimmten System-Account gebunden und verwaltet alle Systemressourcen für die Datenbanken, die zu diesem Account gehören. Jede Datenbank muss innerhalb einer Instanz erstellt werden. Jede Kleingruppe ist Eigentümer einer eigenen DB2-Instanz. Innerhalb dieser könnt Ihr frei operieren, Konfigurationen ändern, Datenbanken erstellen und löschen, ohne dabei anderen Gruppen in die Quere zu kommen.

Der Datenbankmanager der Instanz kann mit dem Befehl

`db2start`

gestartet, und mit

`db2stop`

gestoppt werden. Solltet Ihr eine Fehlermeldung erhalten, dass kein Datenbankmanager läuft, dann führt zunächst diesen Befehl aus. Insbesondere muss der

Datenbankmanager neu gestartet werden, wenn Ihr Konfigurationseinstellungen ändert.

Um mit Datenbanken arbeiten zu können, die nicht zur eigenen Instanz gehören, muss man diese dem System zunächst einmal bekannt machen. Dazu benutzt DB einen Katalog, das sogenannte **Node Directory**. Im Node Directory werden alle fremden Instanzen eingetragen, mit denen man arbeiten möchte. Zu ihnen werden Verbindungsparameter (Host, Port) und Instanzbesitzer festgehalten. Zu Beginn ist das Node Directory für Eure Kleingruppe noch leer. Ihr könnt es Euch mit dem Befehl

```
list node directory
```

anzeigen lassen. Solange keine Einträge vorhanden sind wird DB2 mit der Ausgabe

```
SQL1027N Das Knotenverzeichnis wurde nicht gefunden.
```

antworten. Um nun dem System andere (entfernte) Instanzen bekannt zu machen, wird der Befehl `catalog tcpip node` genutzt. Dieser Befehl erwartet einige Parameter, die wichtig sind, damit die spätere Verbindung zur entfernten Datenbank hergestellt werden kann. Kommt es zu Verbindungsschwierigkeiten, ist oft eine fehlerhafte Katalogisierung schuld. Dann muss der Katalogeintrag gelöscht und neu angelegt werden. Die Parameter sind:

remote der Hostname, auf dem die Instanz liegt

server der Port, über den auf die Instanz zugegriffen werden kann

remote_instance der Name der Instanz

system der Name des entfernten System

ostype das Betriebssystem des entfernten Rechners

Ein beispielhafter Katalogisierungsbefehl sieht dann z.B. wie folgt aus (der Befehl gehört komplett in eine Zeile). Dieser katalogisiert die Intanz, mit der Ihr im ersten Praktikumsblock arbeiten werdet lokal unter dem Namen `sample`.

```
CATALOG TCPIP NODE sample REMOTE bijou.is.inf.uni-due.de SERVER
50005 REMOTE_INSTANCE dbmaster SYSTEM bijou OSTYPE linux
```

DB2 sollte daraufhin mit einer Erfolgsmeldung antworten. Eine erneute Anzeige des Instanzen-Katalogs (Node Directory) sollte nun die katalogisierte Instanz anzeigen. Entfernen kann man eine fehlerhaft katalogisierte Instanz mit dem Befehl

`UNCATALOG NODE` `Name`

Nun könnt Ihr Euch Zugriff auf eine entfernte Datenbank in der katalogisierten Instanz verschaffen, indem Ihr dieser einen lokalen Alias zuweist. Mit diesem Alias könnt Ihr die Datenbank dann im Weiteren so behandeln, als läge sie lokal vor. Der Befehl dazu ähnelt dem Katalogisieren einer Instanz:

`CATALOG DATABASE` *entfernterName* `AS` *lokalerAlias* `AT NODE` *entfernteInstanz*

Angenommen, Ihr habt die Instanz wie oben lokal als `sample` katalogisiert, dann könnt Ihr Euch mit dem folgenden Befehl den Zugriff auf unsere Beispieldatenbank verschaffen:

`CATALOG DATABASE mondial AS mygeo AT NODE sample`

Auch die Liste der katalogisierten Datenbanken kann man sich anzeigen lassen:

`LIST DB DIRECTORY` oder `LIST DATABASE DIRECTORY`

Nicht mehr benötigte oder fehlerhaft katalogisierte Datenbanken kann man mit dem Befehl

`UNCATALOG DATABASE` *Name*

auch wieder aus dem Katalog entfernen. Achtung: lokal erstellte Datenbanken werden automatisch in den Katalog eingetragen. Wenn Ihr den Katalogeintrag löscht, habt Ihr keinen Zugriff mehr auf die lokale Datenbank und müsst sie neu katalogisieren. Das Verfahren ist das gleiche wie für entfernte Datenbanken, nur muss kein Alias und keine entfernte Instanz angegeben werden (`CATALOG DATABASE` *lokalerName*).

2.3 Verbindung herstellen

Da theoretisch beliebig viele Datenbanken an einer Instanz verfügbar sein können, muss zunächst eine Verbindung zu einer bestimmten katalogisierten Datenbank hergestellt werden, bevor Befehle auf dieser ausgeführt werden können. Der Befehl dazu lautet:

`CONNECT TO` *Name* (z.B. `CONNECT TO mygeo`)

Alle weiteren Befehle bis zum Beenden der Verbindung oder bis eine Verbindung zu einer anderen Datenbank beziehen sich nun auf diese Datenbank. Nicht mehr benötigte Verbindungen sollten immer geschlossen werden:

`DISCONNECT` *Name*

oder

`TERMINATE` (beendet alle offenen Verbindungen)

2.4 Schema und Tabellen

Ein Datenbank-Schema ist eine Sammlung von Tabellen. Dabei ist die Definition eines Schemas in DB2 etwas freier, als die aus der Vorlesung. Innerhalb einer Datenbank kann es Tabellen aus mehreren Schemata geben, und Tabellen aus einem Schema können sich in mehreren Datenbanken wiederfinden. Die Systemtabellen von DB2 finden sich im Schema `SYSIBM` und im Schema `SYSCAT`. Hier liegen beispielsweise die Tabellen, in der Eure Tabellen- und Sichtendefinitionen gespeichert werden.

Standardmäßig werden neue Tabellen in einem Schema angelegt, dessen Name dem des Instanzbesitzers entspricht, also z.B. `dbp001` oder `dbmaster`.

Um eine Liste der Tabellen im Standardschema anzuzeigen benutzt man den Befehl

```
LIST TABLES
```

Um eine Liste der Tabellen in einem anderem Schema anzuzeigen benutzt man

`LIST TABLES FOR SCHEMA` *SchemaName*

Solange man Tabellen aus einem fremden Schema benutzen will, muss man den Schemanamen dem Tabellennamen mit Hilfe der Punktnotation voranstellen. So kann man beispielsweise mit der folgenden Abfrage herausfinden, wieviele Zeilen in der Systemtabelle `tables` sind:

```
SELECT count(*) FROM sysibm.tables
```

Beim längeren Arbeiten mit einer fremden Datenbank kann es recht lästig sein, ständig das Schema voranzustellen. Dann kann man auch das Standardschema für die laufende Sitzung umstellen:

```
SET SCHEMA  SchemaName
```

Um mehr über eine Tabelle in der aktuell verbundenen Datenbank herauszufinden, also z.B. welche Spalten sie besitzt, welche Datentypen erwartet werden oder ob Null-Werte erlaubt sind, kann man sich diese vom System beschreiben lassen:

```
DESCRIBE TABLE  SchemaName.TabellenName
```

Optional kann man dem Befehl noch `SHOW DETAIL` nachstellen, um etwas mehr Information zu erhalten, doch diese Information ist eher unwichtig für die meisten Zwecke. Die meisten anderen Sachen, wie z.B. existierende Constraints auf Tabellen können aus den System-Tabellen gelesen werden. Wir werden in einem späteren Arbeitsblock darauf zurückkommen.

2.5 Verbindung mit JDBC

Um sich von außerhalb der Uni mit der DB zu verbinden:

```
public static Connection createConnection() throws
    ClassNotFoundException, SQLException {
 Class.forName("com.ibm.db2.jcc.DB2Driver");

 Properties properties = new Properties();

 properties.setProperty("user","DBPXXX");
 properties.setProperty("password","password");

 Connection connection = DriverManager.getConnection("jdbc:db2
     ://<rechnername>.is.inf.uni-due.de:50005/<DBNAME>:
     currentSchema=<DBPXXX>;", properties);

 return connection;
}
```

3 Anfragen mit SQL

3.1 Subselect

In DB2-SQL gibt es drei Formen von Anfragen: Subselects, Fullselects und das volle SELECT-Statement. Bei jeder Anfrage (Query) muss man für jede verwendete Tabelle oder Sicht zumindest das SELECT-Recht besitzen.

Ein Subselect besteht aus sieben Klauseln, wobei nur die SELECT- und FROM-Klauseln zwingend sind. Die Reihenfolge ist festgelegt:

```
SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...
FETCH FIRST ...
```

Zur vollen Spezifikation der Syntax sei zum Beispiel auf die DB2-Dokumentation verwiesen.

3.1.1 SELECT

Im SELECT-Teil wird angegeben, wie das Ergebnis aussehen soll, d.h. welche Spalten die Ergebnisrelation haben soll, und gegebenenfalls wie diese benannt werden. Die einzelnen Spalten des Ergebnisses sind durch Kommata zu trennen, die Nachstellung eines nicht qualifizierten Spaltennamens benennt die Spalte der Ergebnisrelation (um). Durch `*` oder `Tabelle.*` werden alle Spalten (der benannten Tabelle) ins Ergebnis übernommen.

Select mit Umbenennung einer Ergebnisspalte:

```
SELECT name, code AS 'kfz'
FROM country
```

SELECT DISTINCT nimmt Duplikateliminierung vor. Es ist möglich, die Werte der Ergebnisspalten im SELECT-Teil durch einen *Ausdruck* berechnen zu lassen. Die Datentypen der Ergebnisspalten entsprechen im Wesentlichen den Datentypen der Ursprungsspalten, bzw. dem Resultatdatentyp eines Ausdrucks.

3.1.2 FROM

Der FROM-Teil spezifiziert die zur Anfrage herangezogenen Relationen (Tabellen), dabei kann man durch Nachstellung eines eindeutigen Bezeichners (mit dem optionalen Schlüsselwort `AS`) Tabellen temporär einen neuen Namen geben (Stichwort: Tupelvariablen). Werden mehrere Tabellen in einer Anfrage verwendet, so sind diese im FROM-Teil durch Kommata getrennt anzugeben. Die Anfrage arbeitet dann auf dem kartesischen Produkt dieser Tabellen.

Beispiel für Tupelvariablen zur eindeutigen Bezeichnung von Tabellen:

```
SELECT c1.*
FROM city AS c1,
     city AS c2
```

Es ist möglich, im FROM-Teil als Relation eine Subquery (ein Fullselect) anzugeben. In dem Fall ist es zwingend, dieser Relation einen Namen zu geben.

Statt einer einzelnen Tabelle kann man eine "joined table"-Klausel als Zwischenrelation spezifizieren, die Ergebnis eines JOIN ist:

```
tab1 JOIN                              tab2 ON condition
tab1 INNER JOIN                        tab2 ON condition
tab1 {LEFT,RIGHT,FULL} (OUTER) JOIN tab2 ON condition
```

Die Definition der verschiedenen JOIN-Arten ist wie in der Vorlesung **Datenbanken** behandelt, bzw. wie in der begleitenden Literatur beschrieben. Ohne Angabe eines JOIN-Operators wird der INNER JOIN angenommen. Bei mehreren JOINs kommt es bei OUTER JOINs auf die Reihenfolge an, daher ist zu beachten, dass die JOINs in der Regel von links nach rechts berechnet werden, wobei die Position der JOIN-Bedingung eine Rolle spielt.

```
SELECT DISTINCT ci.name,co.name
FROM city AS ci, country AS co
```

bildet alle möglichen Kombinationen (also das kartesische Produkt) der Tupel aus `city` und `country` und liefert aus der kombinierten Relation den Namen der Städte und Länder zurück. Eine solche Kombination ohne JOIN-Bedingung ist in den seltesten Fällen gewünscht.

```
SELECT ci.name, co.name
FROM city AS ci JOIN country AS co
  ON ci.country = co.code
```

bildet nur die Kombinationen der Tupel aus `city` und `country`, bei denen der Ländercode aus `city` mit dem Ländercode in `country` übereinstimmt, also genau die Kombinationen von `city` und `country`, bei denen die Stadt in dem betreffenden Land liegt.

Das folgende Statement bewirkt das gleiche wie das letzte Beispiel:

```
SELECT ci.name, co.name
FROM city AS ci, country AS co
WHERE ci.country = co.code
```

3.1.3 WHERE

Über die WHERE-Klausel werden alle Tupel ausgewählt, für welche die angegebene Suchbedingung zu *true* evaluiert. Für die Suchbedingung muss gelten:

- die benutzten Spaltennamen müssen eindeutig eine Spalte der im FROM-Teil angegebenen Relationen oder des Ergebnisses eines JOINS aus dem FROM-Teil sein
- eine Aggregatfunktion darf nur verwendet werden, wenn die WHERE-Klausel in einer Subquery einer HAVING-Klausel verwendet wird

```
SELECT *
FROM anwender
WHERE substr(nick,1,2) = 'Op'
```

liefert alle Tupel der Relation `Anwender`, bei denen das Attribut Nick mit 'Op' beginnt. Die gleiche Bedingung könnte auch in der folgenden Form geschrieben werden:

```
SELECT *
FROM anwender
WHERE nick like 'Op%'
```

3.1.4 GROUP BY

Über die GROUP BY-Klausel können die Tupel des bisherigen Ergebnisses gruppiert werden, dabei ist zumindest ein Gruppierungsausdruck anzugeben. Dieser darf keine Subqueries enthalten und alle Tupel, für welche der Gruppierungsausdruck denselben Wert ergibt, gehören zu einer Gruppe. Für jede Gruppe wird ein Ergebnistupel berechnet. Meist benutzt man GROUP BY zusammen mit Aggregratfunktionen, und die Gruppen legen fest, auf welchen Bereich die Funktion angewandt wird.

Wird in einem Subselect GROUP BY oder HAVING verwendet und treten in den Ausdrücken der SELECT-Klausel Spaltennamen auf, so müssen diese in einer Aggregatfunktion vorkommen, von einem Ausdruck der GROUP BY-Klausel abgeleitet sein oder aus einer umgebenden Anfrage stammen.

3.1.5 HAVING

Über die HAVING-Klausel kann man Bedingungen an die gebildeten Gruppen stellen. Es werden die Gruppen ausgewählt, für welche die Suchbedingung zu *true* evaluiert. Wurde in der Anfrage keine GROUP BY-Klausel angegeben, wird das ganze bisherige Ergebnis als eine Gruppe behandelt.

Es dürfen in dieser Suchbedingung nur Spaltennamen verwendet werden, die aus einer äußeren Anfrage stammen, innerhalb der Aggregratfunktion verwendet werden, oder die in einem Gruppierungsausdruck vorkommen.

3.1.6 ORDER BY

Über die ORDER BY-Klausel wird angegeben, wie die Ergebnisrelation sortiert werden soll, d.h. in welcher Reihenfolge die Ergebnistupel ausgegeben werden sollen. Es können mehrere Sortierschlüssel angegeben werden, dann wird zuerst gemäß dem ersten Schlüssel sortiert, dann für in diesem Schlüssel übereinstimmende Tupel nach dem zweiten Schlüssel, usw.

Über die Angaben von ASC bzw. DESC wird festgelegt, ob die Sortierung aufsteigend oder absteigend erfolgt. Der Standard ist ASC, wobei der NULL-Wert als größer als alle anderen Werte angenommen wird.

Als Sortierschlüssel ist entweder ein Spaltenname der Ergebnisrelation (alternativ kann auch ein Integerwert benutzt werden, um die Spalte anhand ihrer Position auszuwählen), oder einer der in der FROM-Klausel verwendeten Relationen oder ein Ausdruck erlaubt. Wurde im SELECT-Teil mit DISTINCT gearbeitet, so muss der Sortierschlüssel exakt einem Ausdruck der SELECT-Liste entsprechen. Ist das Subselect gruppiert, so können nur Ausdrücke der SELECT-Liste, Gruppierungsausdrücke und Ausdrücke, die eine Aggregatfunktion, Konstante oder Hostvariable enthalten verwendet werden.

Die folgende Anfrage listet alle Filmdaten nach Jahren und Titeln geordnet auf, mit den neuesten Filmen zuerst:

```
SELECT *
FROM film
ORDER BY drehjahr DESC,titel ASC
```

3.1.7 FETCH FIRST

Die FETCH FIRST-Klausel erlaubt es, die Ergebnisrelation auf eine bestimmte Zahl von Tupeln einzuschränken. Das ist in jedem Fall für das Testen von Anfragen zu empfehlen, weil unter Umständen nicht alle Tupel berechnet werden müssen und so die Performance steigt.

Die ersten 5 Serienfolgen, bei denen der Folgentitel nicht leer (Null) und auch nicht die leere Zeichenkette ist:

```
SELECT serientitel,drehjahr,staffelnr,folgennr,folgenTitel
FROM serienfolge
WHERE folgentitel IS NOT NULL
  AND folgentitel<>''
ORDER BY drehjahr,serientitel,staffelnr,folgennr DESC
FETCH FIRST 5 ROWS ONLY
```

3.2 Aggregatfunktionen

Aggregatfunktionen werden auf eine Menge von Werten angewendet, die von einem Ausdruck abgeleitet sind. Die verwendeten Ausdrücke können sich auf Attribute, nicht aber auf skalarwertige Anfragen oder andere Aggregatfunktionen beziehen. Der Bereich, auf den eine Aggregatfunktion angewendet wird, ist eine Gruppe oder eine ganze Relation. Ist das Ergebnis der bisherigen Bearbeitung eine leere Menge und wurde GROUP BY verwendet, so werden die Aggregatfunktionen nicht berechnet, und es wird eine leere Menge zurückgegeben. Wurde GROUP BY verwendet, werden die Aggregatfunktionen auf die leere Menge angewendet.

Wurde in einem Subselect kein GROUP BY und kein HAVING verwendet, dürfen in der SELECT-Klausel Aggregatfunktionen nur auftreten, wenn sie alle Attributnamen umfassen.

Die folgenden Aggregratfunktionen nehmen jeweils genau ein Argument. NULL-Werte werden gegebenenfalls vor der Berechnung eliminiert. Auf die leere Menge angesetzt, liefern sie NULL zurück. Über das zusätzliche Schlüsselwort DISTINCT kann man erreichen, dass vor der Berechnung Duplikateliminierung durchgeführt wird.

AVG: Durchschnitt der Werte

MAX: Maximum der Werte

MIN: Minimum der Werte

STDDEV: Standardabweichung der Werte

SUM: Summe der Werte

VARIANCE: Varianz der Werte

Beispiel:

Welches ist das letzte Jahr, für welches Filme in der Datenbank vorliegen?

```
SELECT max(drehjahr)
FROM film
```

Die COUNT-Funktion wird ausgewertet zur Anzahl der Werte, zu denen der Argumentausdruck evaluiert. COUNT(*) liefert die Anzahl der Tupel im jeweiligen Bereich zurück, Gruppierungen werden berücksichtigt. Im Ausdruck werden NULL-Werte eliminiert. Das Ergebnis ist dann jeweils die Anzahl der (bei Verwendung von DISTINCT paarweise verschiedener) Werte im jeweiligen Bereich.

Das Ergebnis von COUNT kann nie NULL sein.

3.2.1 Beispiele

Anzahl Folgen pro „Buffy"-Staffel:

```
SELECT serientitel,drehjahr,staffelnr,COUNT(folgennr) AS anzahl
  FROM serienfolge
  WHERE serientitel like 'Buffy%'
  GROUP BY serientitel,drehjahr,staffelnr
```

Längster Filmtitel:

```
SELECT titel
FROM film
WHERE length(titel) >= ALL (SELECT DISTINCT length(titel) FROM film)
```

Wie viele Folgen haben die Staffeln von „Buffy" durchschnittlich? (Mit einer Genauigkeit von 2 Stellen hinter dem Komma.)

```
SELECT serientitel,drehjahr,DEC(AVG(FLOAT(anzahl)),4,2) AS avgfolgen FROM
  (SELECT serientitel,drehjahr,staffelnr,COUNT(folgennr) AS anzahl
     FROM serienfolge
```

```
    WHERE serientitel like 'Buffy%'
    GROUP BY serientitel,drehjahr,staffelnr
  ) as tmp
  GROUP BY serientitel,drehjahr
```

Weitere Beispiele zu Subselects, Joins und Gruppierung finden sich in der Dokumentation zu DB2, bzw. können dem Begleitmaterial zur Vorlesung entnommen werden.

3.3 Fullselect

Ein Fullselect besteht aus zwei oder mehr Teilselects, die über Mengenoperatoren verknüpft werden. Dabei müssen die Ergebnistupel der Teilselects jeweils die gleiche Stelligkeit besitzen, und die Datentypen an den entsprechenden Positionen müssen kompatibel sein. Die verfügbaren Mengenoperatoren in SQL sind UNION, UNION ALL, EXCEPT, EXCEPT ALL, INTERSECT, INTERSECT ALL. Zwei Tupel werden im Folgenden als gleich betrachtet, wenn die jeweils korrespondierenden Komponenten gleich sind (wobei zwei NULL-Werte als gleich betrachtet werden).

Werden mehr als zwei Verknüpfungsoperatoren verwendet, so werden zuerst die geklammerten Fullselects berechnet, dann werden alle INTERSECT-Operationen ausgeführt, und zuletzt werden die Operationen von links nach rechts ausgewertet.

3.3.1 UNION

UNION ist die Mengenvereinigung, dabei findet Duplikateliminierung statt. Bei Verwendung von UNION ALL werden bei der Vereinigung keine Duplikate eliminiert.

Alle Schauspieler aus der Serie „Buffy the Vampire Slayer" und dem Film „Buffy the Vampire Slayer" von 1992:

```
  SELECT schauspielerid
  FROM spieltInFolge
  WHERE serientitel='Buffy the Vampire Slayer'
UNION
  SELECT schauspielerid
  FROM spieltInFilm
  WHERE titel='Buffy the Vampire Slayer' and drehjahr=1992
```

3.3.2 INSERSECT

INTERSECT ist der Mengendurchschnitt, d.h. das Ergebnis wird durch alle Tupel gebildet, die in beiden Relationen liegen. Bei der Verwendung von INTERSECT ALL enthält das Ergebnis auch Duplikate, im Falle von INTERSECT nicht.

Alle Schauspieler, die sowohl in der Serie von 1997, als auch dem Film „Buffy the Vampire Slayer" von 1992 mitgespielt haben:

```
  SELECT schauspielerid
  FROM spieltInFolge
  WHERE serientitel='Buffy the Vampire Slayer'
INTERSECT
  SELECT schauspielerid
  FROM spieltInFilm
  WHERE titel='Buffy the Vampire Slayer' and drehjahr=1992
```

3.3.3 EXCEPT

Bei EXCEPT ALL wird das Ergebnis dadurch bestimmt, dass man – unter Beachtung von Duplikaten – aus der linksstehenden Relation alle Tupel entfernt, die auch in der rechten Relation vorkommen. Bei Verwendung von EXCEPT wird das Ergebnis durch alle Tupel der linken Relation gebildet, die **nicht** in der rechten Relation vorkommen. Dann enthält das Ergebnis auch keine Duplikate.

Alle Schauspieler aus der ersten „Buffy"-Staffel, die nicht (!) auch in der siebten Staffel mitgespielt haben:

```
  SELECT schauspielerid
  FROM spieltInFolge
  WHERE serientitel='Buffy the Vampire Slayer'
    AND staffelnr=1
EXCEPT
  SELECT schauspielerid
  FROM dbprak.spieltInFolge
  WHERE serientitel='Buffy the Vampire Slayer'
    AND staffelnr=7
```

4 Views

Ein wichtiges Datenbankkonzept sind Sichten oder Views. Diese können in SQL mit dem CREATE VIEW-Statement erzeugt werden.

Eine Sicht, die nur die Nicks, Wohnorte und Mailadressen der Anwender angibt, Passwörter und Realnamen dagegen verschweigt:

```
CREATE VIEW Nicks AS
  SELECT nick,wohnort,email
  FROM anwender
```

Ein View liefert eine bestimmte Ansicht auf die Datenbank. Man kann Sichten für verschiedene Zwecke einsetzen. Im Datenschutz kann man durch Sichten etwa für bestimmte Nutzergruppen sensible Daten über eine Sicht ausschließen. Man kann Anfragen vereinfachen, wenn man Sichten als eine Art Makro betrachtet, das an geeigneter Stelle in ein SELECT-Statement übernommen werden kann. Schließlich lassen sich mit Sichten auch Generalisierungsmodelle nachempfinden:

```
CREATE TABLE film (
   titel VARCHAR(200) NOT NULL,
   drehjahr INTEGER NOT NULL
)

CREATE TABLE serienfolge (
   serientitel VARCHAR(200) NOT NULL,
   drehjahr INTEGER NOT NULL
   folgentitel VARCHAR(200),
   staffelnr INTEGER NOT NULL,
   folgennr INTEGER NOT NULL
)

CREATE VIEW produktionen AS
    SELECT titel,drehjahr FROM film
  UNION
    SELECT serientitel AS titel,drehjahr FROM serienfolge
```

Ebenso wie in diesem Beispiel der Obertyp als Sicht realisiert wurde, lassen sich auch Untertypen als Sichten eines Obertyps realisieren.

Sichten können wie Tabellen benutzt werden. Sichten haben aber das Problem, dass sie oft nicht updatefähig sind. Eine Sicht ist veränderbar (d.h. erlaubt UPDATE), wenn sie weder Aggregratfunktionen, noch Anweisungen wie DISTINCT, GROUP BY oder HAVING enthält, in der SELECT-Klausel nur eindeutige Spaltennamen stehen und ein Schlüssel der Basisrelation enthalten ist, und sie nur genau eine Tabelle verwendet, die ebenfalls veränderbar ist.

5 Unterstützte Funktionen

Eine vollständige Beschreibung der Funktionen, die DB2 unterstützt, findet sich in der Online-Dokumentation. Hier eine Liste der wichtigsten Funktionen.

Beispiele:

- `FLOAT(anzahl)`

  wandelt die Werte des Attributes `anzahl` vor der Weiterverarbeitung in einen anderen Datentyp um

- `DEC(durchschnittAlsFloat,4,2)`

  stellt den Wert von `durchschnittAlsFloat` als Dezimalzahl mit insgesamt 4 Stellen, davon 2 Stellen hinter dem Komma, dar

- `SUBSTR(nick, 1, 2)`

  liefert die ersten 2 Zeichen der Werte von `nick`

- `CAST(null AS int)`

  führt eine Typkonvertierung des NULL-Wert auf integer durch

Typkonvertierung: BIGINT, BLOB, CAST, CHAR, CLOB, DATE, DBCLOB, DECIMAL, DREF, DOUBLE, FLOAT, GRAPHIC, INTEGER, LONG, LONG_VARCHAR, LONG_VARGRAPHIC, REAL, SMALLINT, TIME, TIMESTAMP, VARCHAR, VARGRAPHIC

Mathematik: ABS, ACOS, ASIN, ATAN, CEIL, COS, COT, DEGREES, EXP, FLOOR, LN, LOG, LOG10, MOD, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQRT, TAN, TRUNC

Stringmanipulation: ASCII, CHR, CONCAT, DIFFERENCE, DIGITS, HEX, INSERT, LCASE, LEFT, LENGTH, LOCATE, LTRIM, POSSTR, REPEAT, REPLACE, RIGHT, RTRIM, SOUNDEX, SPACE, SUBSTR, UCASE, TRANSLATE

Datumsmanipulation: DAY, DAYNAME, DAYOFWEEK, DAYOFYEAR, DAYS, HOUR, JULIAN_DAY, MICROSECOND, MIDNIGHT_SECONDS, MINUTE, MONTH, MONTHNAME, QUARTER, SECOND, TIMESTAMP_ISO, TIMESTAMPDIFF, WEEK, YEAR

System: EVENT_MON_STATE, NODENUMBER, PARTITION, RAISE_ERROR, TABLE_NAME, TABLE_SCHEMA, TYPE_ID, TYPE_NAME, TYPE_SCHEMA

Sonstige: COALESCE, GENERATE_UNIQUE, NULLIF, VALUE

6 SQL Programming Language

Die SqlPl ist im SQL99-Standard definiert, und ist eine sehr einfache Programmiersprache, die innerhalb eines DBMS benutzt werden kann. Insbesondere findet sie in Stored Procedures, Triggern und bei der Implementierung von SQL-UDFs Verwendung. Im Falle von Triggern und Funktionen werden die SqlPl-Anweisungen im DBMS abgelegt und bei Bedarf interpretiert.

Dabei umfasst die SqlPl unter anderem die im Folgenden kurz beschriebenen Anweisungen. Prozeduren erlauben zusätzliche Kontroll-Statements, die hier aber nicht behandelt werden sollen.

WICHTIG: Getrennt werden die einzelnen Anweisungen in SqlPl durch Semikolons. Daher muss man beachten, dass man Zeilen im CLP nicht mit einem Semikolon enden lässt (mit Ausnahme des tatsächlichen Ende des Statements), oder ein alternatives Terminierungszeichen benutzen. So erwartet die Kommandozeile beim Aufruf des CLP mit `db2 -c -td@` z.B. das Zeichen `@` als Ende einer Zeile.

Um mehrere SqlPl-Statements zusammenzufassen, kann man im Rumpf einer Trigger- oder Funktionsdefinition einen mit BEGIN ... END geklammerten Anweisungsblock benutzen. Dieses ist ausführlicher in der Dokumentation beschrieben, und sieht z.B. so aus:

```
name:
BEGIN ATOMIC
     DECLARE counter INTEGER;
```

```
    SET counter = (SELECT count(*) FROM table);

    Anweisungen;
END name
```

Durch das Schlüsselwort ATOMIC wird festgelegt, dass bei Fehlern innerhalb des Anweisungsblocks **alle** Anweisungen des Blocks zurückgenommen werden. Variablen, die man innerhalb des Anweisungsblocks benutzen möchte, definiert man mit DECLARE. Dabei gibt man der Variable einen lokalen Namen, spezifiziert den Datentyp und definiert gegebenenfalls einen Default-Wert für die Variable (standardmäßig NULL). Mittels SET wird der Wert einer Variable gesetzt.

Durch das Label *name* ist es möglich, den benannten Block durch ein LEAVE-Statement zu verlassen. Außerdem kann das Label zur Qualifizierung von Variablen benutzt werden.

6.0.1 SQL-Statements

Viele der üblichen SQL-Statements sind auch innerhalb von Funktionsdefinitionen erlaubt: das Fullselect, UPDATE oder DELETE mit Suchbedingung, INSERT und SET für Variablen.

6.0.2 FOR

Das FOR-Statement führt ein oder mehrere SQL-Statements für jedes Tupel einer Relation aus. Dabei kann die FOR-Schleife durch Voranstellung von *label:* benannt werden. Dieses Label kann dann in LEAVE- oder ITERATE-Statements benutzt werden.

```
BEGIN
    DECLARE vollertitel CHAR(40);
    FOR t AS
       SELECT titel, jahr, budget FROM produktion
       DO
         SET vollertitel = t.titel || ' (' || t.jahr || ')';
         INSERT INTO einetabelle VALUES (vollertitel, t.budget);
    END FOR;
END
```

Das obige Beispiel benutzt den '||'–Operator zur Stringkonkatenation (Zusammenfügen zweier Zeichenketten zu einer neuen).

6.0.3 IF

Das IF-Statement ermöglicht bedingte Ausführung von Statements. Die Anweisung wird ausgeführt, wenn die Suchbedingung zu *true* evaluiert. Liefert sie *false* oder *unknown*, dann wird stattdessen die nächste Suchbedingung hinter einem ELSEIF ausgeführt. Evaluiert keine Suchbedingung zu *true*, wird stattdessen der ELSE-Zweig ausgeführt.

```
IF anzahl_postings < 10 THEN
   SET typ = 'Anfnger';
ELSEIF anzahl_postings < 100 THEN
   SET typ = 'Mitglied';
ELSEIF anzahl_postings < 1000 THEN
   SET typ = 'Seniormitglied';
ELSE
   SET typ = 'Profi';
END IF

IF folgen_gesamt > (SELECT count(*)
        FROM seriefolge
        WHERE serientitel = titel1
          AND drehjahr = titel2)
THEN RETURN 'unvollstndig';
ELSE RETURN 'vollstndig';
END IF
```

6.0.4 WHILE

Eine WHILE-Schleife wird solange durchlaufen, bis die angegebene Suchbedingung nicht mehr als *true* evaluiert. Bei jedem Durchlauf wird der Schleifenkörper ausgefhrt. Dabei kann die WHILE-Schleife durch Voranstellung von *label:* benannt werden. Dieses Label kann in LEAVE- oder ITERATE-Statements benutzt werden.

```
WHILE counter < 10
DO
   Schleifenkörper
END WHILE
```

6.0.5 ITERATE

Mit dem ITERATE-Statement verlässt man die Abarbeitung des aktuellen Schleifendurchlaufs und springt zum Anfang der benannten Schleife (FOR, LOOP, REPEAT oder WHILE) zurück.

```
ITERATE schleifenname
```

6.0.6 LEAVE

Mit dem LEAVE-Statement verlässt man vorzeitig eine benannte Schleife (FOR, LOOP, REPEAT, WHILE) oder einen Anweisungsblock. Dabei werden gegebenenfalls alle Cursor geschlossen, mit Ausnahme derer, welche die Ergebnisrelation definieren.

```
LEAVE schleifenname
```

6.0.7 RETURN

Mit dem RETURN-Statement kann man aus einer Funktion heraus springen. Dabei muss ein Rückgabewert definiert werden, der mit dem Rückgabetyp der Funktion kompatibel ist. Bei Tabellenfunktionen muss ein Fullselect benutzt werden.

```
RETURN substring(name,locate(name,','))

RETURN SELECT name, todesdatum
       FROM person
       WHERE todesdatum IS NOT NULL
```

6.0.8 Zuweisungen

Über SET kann man innerhalb eines Anweisungsblock oder im Rumpf eines Kontroll-Statements Zuweisungen an Variables tätigen.

SIGNAL

Mit dem SIGNAL-Statement kann man einen SQL-Fehler oder eine SQL-Warnung werfen. Der erklärende Fehlertext darf maximal 70 Zeichen lang und kann auch eine Variable sein. Dabei gilt zur Benennung des SQL-Status:

- ein SQL-Status ist ein fünf Zeichen langer String aus Großbuchstaben und Ziffern,
- er darf nicht mit '00' beginnen (Erfolg),
- in Triggern oder Funktionen darf er nicht mit '01' (Warnung) oder '02' beginnen,
- ein SQL-Status, der nicht mit '00', '01' oder '02' beginnt, signalisiert einen Fehler,
- beginnt der String mit 0-6 oder A-H, so müssen die letzten drei Zeichen mit I-Z beginnen

```
SIGNAL SQLSTATE  '03ILJ'
   SET MESSAGE_TEXT='Fehler: Illegales Jahr'
```

7 Rekursive Anfragen

In einem vollen SELECT-Statement ist es auch möglich, rekursive Anfragen zu stellen. Rekursive Anfragen sind solche Anfragen, die sich in ihrer Definition auf sich selbst beziehen. Derart kann man etwa die *transitive Hülle* eines Tupels oder Stücklisten berechnen.

Ein kurzer Ausflug, um das Konzept der Rekursion zu erklären:

Rekursion
aus Wikipedia, der freien Enzyklopädie
(`http://de.wikipedia.org/wiki/Rekursion`)

Rekursion bedeutet Selbstbezüglichkeit. Sie tritt immer dann auf, wenn etwas auf sich selbst verweist oder mehrere Dinge aufeinander, so dass merkwürdige Schleifen entstehen. So ist z.B. der Satz "Dieser Satz ist unwahr" rekursiv, da er von sich selber spricht. [...]

Dabei ist Rekursion ein allgemeines Prinzip zur Lösung von Problemen, das in vielen Fällen zu "eleganten" mathematischen Lösungen führt. Als Rekursion bezeichnet man den Aufruf bzw. die Definition einer Funktion durch sich selbst. Ohne geeignete Abbruchbedingung geraten solche rückbezüglichen Aufrufe in einen sogenannten infiniten Regress (Endlosrekursion)

[...] Die Definition von rekursiv festgelegten Funktionen ist eine grundsätzliche Vorgehensweise in der funktionalen Programmierung. Ausgehend von einigen gegebenen Funktionen (wie zum Beispiel unten die Summen-Funktion) werden neue Funktionen definiert, mithilfe derer weitere Funktionen definiert werden können.

Hier ein Beispiel für eine Funktion $sum : N_0 \to N_0$, die die Summe der ersten n Zahlen berechnet:

$$sum(n) = \begin{cases} 0, & \text{falls } n = 0 \text{ (Rekursionsbasis)} \\ sum(n-1) + n, & \text{falls } n \neq 0 \text{ (Rekursionsschritt)} \end{cases}$$

Bei einer rekursiven Anfrage (RA) muss man folgende Regeln beachten:

- Jedes Fullselect, das Teil einer RA ist, muss mit SELECT beginnen (aber nicht mit SELECT DISTINCT). Es darf nicht geschachtelt sein. Als Vereinigung **muss** UNION ALL benutzt werden.
- Im WITH-Teil müssen Spaltennamen explizit angegeben werden.
- Das erste Fullselect in der ersten Vereinigung darf sich nicht auf die zu definierende Tabelle beziehen. Es bildet die Rekursionsbasis.
- Die Datentypen und Längen der Spalten der zu definierenden Tabelle werden durch die SELECT-Klausel der Rekursionsbasis festgelegt.
- Kein Fullselect in der rekursiven Definition darf eine Aggregatfunktion, eine GROUP BY- oder eine HAVING-Klausel enthalten. Die FROM-Klauseln in der rekursiven Definition dürfen höchstens eine Referenz auf die zu definierende Tabelle besitzen.
- Es dürfen keine Subqueries verwendet werden.

Wenn die Daten Zyklen enthalten können, dann ist es möglich, dass durch eine RA eine Endlosrekursion erzeugt wird. Dann ist es wichtig, eine Abbruchbedingung zu definieren:

- Die zu definierende Tabelle muss ein Attribut enthalten, das durch eine INTEGER-Konstante initialisiert wird, und regelmäßig erhöht wird.
- In jeder WHERE-Klausel muss eine Prüfbedingung auftreten.

Die Auswertung einer rekursiven Anfrage läuft wie folgt ab: Zuerst werden die Tupel der Rekursionsbasis berechnet, dann werden ausgehend von diesen Tupeln gemäß des zweiten Teils des Fullselects (nach dem UNION ALL) weitere Tupel berechnet. Dabei werden jedoch nur diejenigen Tupel aus der rekursiv definierten Relation verwendet, die beim letzten Berechnungsschritt neu hinzugekommen sind (beginnend also mit den Tupeln der Rekursionsbasis). Kamen in einem Berechnungsschritt keine neuen Tupel hinzu, so endet die Berechnung.

Beispiel:

Gegeben sei eine Tabelle mit Bauteilen, in denen die Komponenten für Werkstücke festgehalten werden. Dabei können diese Komponenten selber wiederum aus Einzelteilen zusammengesetzt sein.

```
CREATE TABLE bauteile (
       stueck     VARCHAR(8),
       komponente VARCHAR(8),
       menge      INTEGER);
```

Um nun zu ermitteln, welche Basiskomponenten insgesamt nötig sind, um ein Beispiel-Bauteil herzustellen, kann man eine rekursive Anfrage benutzen. In der folgenden RA ist die im ersten Teil der WITH-Klausel definierte Query `rek` die Rekursionsbasis, im zweiten Teil der WITH-Klausel folgt dann die Definition der Rekursion. Die so rekursiv definierte Ergebnisrelation wird dann im SELECT DISTINCT-Teil benutzt.

```
WITH rek (stueck, komponente, menge) AS (
        SELECT r.stueck, r.kompenente, r.menge
        FROM bauteile r
        WHERE r.stueck = 'Beispiel-Bauteil'
   UNION ALL
        SELECT kind.stueck, kind.komponente, kind.menge
        FROM rek vater,
             bauteile kind
        WHERE vater.komponente = kind.stueck
    )
SELECT DISTINCT stueck, komponente, menge
FROM rek
ORDER BY stueck, komponente, menge
```

8 Trigger

Als Trigger (deutsch: Auslöser) bezeichnet man in SQL eine Anweisungsfolge (eine Abfolge von Aktionen), die ausgeführt wird, wenn eine verändernde Anweisung auf einer bestimmten Tabelle ausgeführt werden soll. Wir erinnern uns aus der Vorlesung, dass verändernde Anweisungen für Tabellen DELETE, INSERT und UPDATE sind.

Man kann Trigger zum Beispiel nutzen, um

- neu eingefügte oder zu verändernde Tupel auf ihre Gültigkeit bezüglich vorgegebener Bedingungen zu überprüfen,
- Werte zu generieren,
- in anderen Tabellen zu lesen (etwa zur Auflösung von Querverweisen) oder zu schreiben (etwa zur Protokollierung)

Insbesondere kann man mit Triggern Regeln in der Datenbank modellieren. Einen Trigger erstellt man mit dem CREATE TRIGGER-Statement, mit DROP TRIGGER wird er gelöscht.

Man kann Trigger entweder so schreiben, dass sie für jedes zu ändernde Tupel einmal (FOR EACH ROW) oder für jedes verändernde Statement einmal (FOR EACH STATEMENT) ausgeführt werden. Letzteres ist allerdings nur für die sogenannten AFTER-Trigger erlaubt, die im Anschluss an eine verändernde Operation tätig werden. In diesem Fall wird der Trigger auch dann abgearbeitet, wenn kein Tupel von dem DELETE- oder dem UPDATE-Statement betroffen ist.

Über die REFERENCING-Klausel werden Namen für die Übergangsvariablen festgelegt: `OLD AS name` bzw. `NEW AS name` definiert `name` als Name für die Werte des betroffenen Tupels vor bzw. nach der Ausführung der auslösenden Operation. Entsprechend definieren `OLD_TABLE AS identifier` bzw. `NEW_TABLE AS identifier` diesen als Name für eine hypothetische Tabelle, welche die betroffenen Tupel vor bzw. nach der auslösenden Operation enthält. Dabei gilt :

auslösendes Ereignis und Zeitpunkt	**ROW-Trigger darf benutzen**	**STATEMENT-Trigger darf benutzen**
BEFORE INSERT	NEW	-
BEFORE UPDATE	OLD, NEW	-
BEFORE DELETE	OLD	-
AFTER INSERT	NEW, NEW_TABLE	NEW_TABLE
AFTER UPDATE	OLD, NEW, OLD_TABLE, NEW_TABLE	OLD_TABLE, NEW_TABLE
AFTER DELETE	OLD, OLD_TABLE	OLD_TABLE

Über vorhandene Trigger auf einer Datenbank kann man die System-Tabelle `SYSIBM.SYSTRIGGERS` konsultieren:

```
SELECT name,text
FROM SYSIBM.SYSTRIGGERS
```

Beispiel:

Angenommen, wir führten eine Tabelle `afps` (Anzahl Folgen pro Serie), dann könnte ein Trigger für das Einfügen von neuen Tupeln in die Tabelle `serienfolge` eventuell derart aussehen:

```
CREATE TRIGGER neue_folge
   AFTER INSERT ON serienfolge
   REFERENCING NEW AS neu
   FOR EACH ROW MODE DB2SQL
   UPDATE afps
```

```
      SET anzahl_folgen = anzahl_folgen + 1
      WHERE neu.serientitel = afps.serientitel
        AND neu.drehjahr = afps.drehjahr
```

Und ein Einfügen eines Tupels in `serienfolge` würde automatisch zu einem UPDATE der Tabelle `afps` führen:

```
INSERT INTO afps (serientitel, drehjahr, anzahl_folgen)
VALUES ('Babylon 5 - The Telepath Tribes', 0);

INSERT INTO serienfolge (serientitel, drehjahr, staffelnr,
        folgennr, folgentitel)
VALUES ('Babylon 5 - The Telepath Tribes', 2007, 1, 1,
        'Forget Byron');

INSERT INTO serienfolge (serientitel, drehjahr, staffelnr,
        folgennr, folgentitel)
VALUES ('Babylon 5 - The Telepath Tribes', 2007, 1, 2,
        'Bester''s Best');

SELECT anzahl_folgen AS Anzahl
FROM afps
WHERE land='Babylon 5 - The Telepath Tribes'

ANZAHL
------
     2
```

Noch ein Beispiel:

```
CREATE TRIGGER check_ausleihe
   NO CASCADE BEFORE INSERT ON ausleihe
   REFERENCING NEW AS neu
   FOR EACH ROW MODE DB2SQL
   IF neu.ausleihdatum > current date
   THEN
      SIGNAL SQLSTATE '75000'
      SET MESSAGE_TEXT =
        'Ein Ausleihvorgang kann nicht im voraus eingefgt werden.';
   END IF
```

Dieser Trigger verbietet das Einfügen neuer Tupel in `ausleihe`, bei denen das Ausleihdatum in der Zukunft liegt. (Es sollen also nur bereits stattgefundene Ausleihvorgänge eingetragen werden.) Der Versuch führt zu einem Fehler und das Statement wird nicht ausgeführt.

9 Vorgehen bei der Modellierung

Der Entwurf einer Datenbank beschreibt den Prozess der Umsetzung einer Mini-Welt in ein Datenbankschema, das in der Lage ist, die gewünschten Daten dieser

Welt mit ihren Eigenschaften und Beziehungen darzustellen.

Der Entwurf, an den sich dann die Implementierung anschließt, besteht im Wesentlichen aus diesen Schritten:

- Modellierung
- Umsetzung in ein Relationenschema
- Normalisierung

10 Entwurfsschritt 1: Modellierung

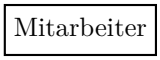
Die Informationen in diesem Abschnitt sollten schon aus der Vorlesung bekannt sein.

10.1 Modellierung mit E/R-Diagrammen

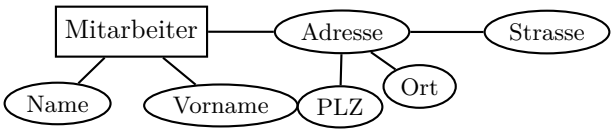
Die Modellierung dient der systematischen Darstellung einer Abstraktion der abzubildenden Miniwelt. Er dient auch zur Kommunikation mit Nichtfachleuten, die in der Regel nicht das Denken in Relationen oder Attributen gewohnt sind. Als eine Möglichkeit zur Modellierung einer Miniwelt ist das Entity-Relationship-Modell weit verbreitet. Zur Darstellung dieser Modelle werden ER-Diagramme benutzt.

10.1.1 Entitäten und Attribute

Ein Objekt der realen Welt wird als **Entität** modelliert. Eine Entität ist normalerweise eine Person, ein Prozess oder ein Gegenstand der realen Welt, z.B. ein Mitarbeiter, eine Lieferung, ein Inventargegenstand oder ein Schriftstück. Gleichartige Entitäten bilden einen **Entitätstyp**, z.B. alle Wissenschaftlichen Angestellten, alle Autoteile oder alle Arbeitsverträge. In einem ER-Diagramm wird ein Entitätstyp durch ein Rechteck dargestellt, das den Namen des Entitätstypen umschließt.

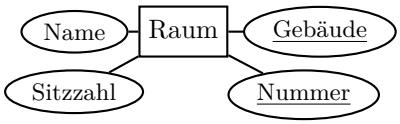


Eine Entität (bzw. ein Entitätstyp) wird durch eine Menge von **Attributen** beschrieben. In einem ER-Diagramm umschließt eine Ellipse den Namen des Attributes und eine Linie verbindet es mit dem zugehörigen Entitätstyp. Ein Attribut von Mitarbeiter ist z.B. der Name. Man spricht von einem **zusammengesetzten Attribut**, wenn ihm wiederum verschiedene Attribute zugeordnet sind. In einem ER-Diagramm werden zusammengesetzte Attribute genau wie atomare Attribute dargestellt.



Attributkombinationen, die zur eindeutigen Identifikation einer Entität dienen, werden **Schlüssel** genannt. Beim Entitätstyp Mitarbeiter könnte z.B. (Name, Vorname, Adresse) oder (Mitarbeiternummer) als Schlüssel dienen. Ein Schlüssel identifiziert über die Attributwerte eine Entität eindeutig, insbesondere können keine zwei Entitäten in allen ihren Schlüsselattributen übereinstimmen. Unter allen Schlüsseln wird ein Schlüssel ausgezeichnet, dieser heißt **Primärschlüssel**.

In einem ER-Diagramm werden die Namen aller Attribute, die zum Primärschlüssel gehören, unterstrichen dargestellt.

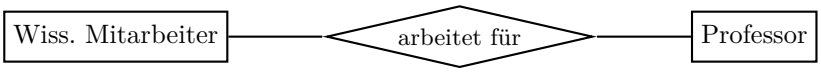


10.1.2 Beziehungen (Relationships)

Die Objekte der realen Welt stehen zueinander in Beziehung. Durch **Beziehungen** werden im ER-Modell Verknüpfungen, Zusammenhänge und Interaktionen zwischen Entitäten, bzw. den Entitätstypen, modelliert. Beispiele sind etwa ist-Teil-von, arbeitet-für, usw.

Beziehungen derselben Art werden zu sogenannten **Beziehungstypen** zusammengefaßt. In einem ER-Diagramm werden Beziehungstypen durch eine Raute dargestellt, die den Namen des Beziehungstyps enthält und mit allen Entitätstypen verbunden ist, die an dieser Beziehung teilnehmen. Ein Entitätstyp kann mehrfach an einer Beziehung teilnehmen. Es ist auch möglich, dass ein Entitätstyp in Beziehung mit sich selbst steht (das nennt man dann *rekursive* Beziehung).

Verbindet ein Beziehungstyp nur zwei Entitätstypen jeweils einmal, so spricht man von einem binären Beziehungstyp, ebenso kennt man ternäre Beziehungen, usw.



Beziehungen können wie Entitätstypen Attribute besitzen. Diese werden im ER-Diagramm genauso dargestellt wie bei Entitätstypen. Da Beziehungstypen nur Zusammenhänge und Beziehungen modellieren, können sie keine identifizierenden Schlüssel besitzen. Im ER-Diagramm werde keine Schlüssel für Beziehungstypen ausgezeichnet.

Bei Beziehungstypen ist es sinnvoll, sich schon in der Modellierungsphase klarzumachen, wie oft die verschiedenen Entitäten an einer Beziehung teilnehmen können. Diese überlegungen führen zum Konzept der **Funktionalitäten**:

Nimmt jede Entität eines Typs genau n-mal an einer Beziehung teil, so wird im ER-Diagramm die Verbindungslinie mit dieser Zahl markiert. Nimmt jede Entität mindestens p-mal und höchstens k-mal an einer Beziehung teil, so wird in der (*min*,*max*)-Notation im ER-Diagramm die Verbindungslinie mit (p,k) markiert. Das Zeichen *N* als Maximum zeigt normalerweise an, dass eine Entität dieses Typs beliebig oft an dieser Beziehung teilnehmen kann.

Im ER-Modell ist es nicht möglich, dass Beziehungen selbst wieder an Beziehungen teilnehmen. Das kann aber manchmal nötig und sinnvoll sein. Hierzu fasst man einen Beziehungstyp mit den an ihm beteiligten Entitätstypen zu einem aggregierten Entitätstyp zusammen. Dieser kann wiederum an Beziehungen teilnehmen. Im ER-Diagramm wird ein aggregierter Entitätstyp dadurch dargestellt, dass die ihn definierende Entitätstypen- und Beziehungstypensymbole in ein Rechteck eingeschlossen werden.

10.1.3 Generalisierung

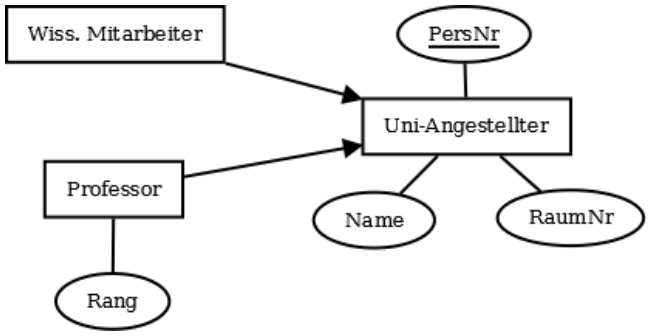
Um die Entitätstypen besser zu strukturieren, kann man im Entwurf **Generalisierungen** einführen. Dieses objektorientierte Konzept abstrahiert Entitätstypen und bildet Obertypen. Die Entitätstypen, die zu einem Obertyp generalisiert wurden, heißen Kategorien oder Untertypen des Obertyps.

Ein Obertyp faßt allen Kategorien gemeinsame Eigenschaften zusammen. Diese werden vom Untertyp geerbt; zusätzlich kann ein Untertyp Eigenschaften haben, die nur für diesen charakterisierend sind. Die Entitätsmenge des Untertyps ist eine Teilmenge der Entitätsmenge des Obertyps, wobei üblicherweise zwei Fälle besonders interessant sind:

- *disjunkte* Spezialisierung: die Schnittmenge von je zwei Untertypen ist leer

- *vollständige* Spezialisierung: Obertyp ergibt sich als Vereinigung der Untertypen

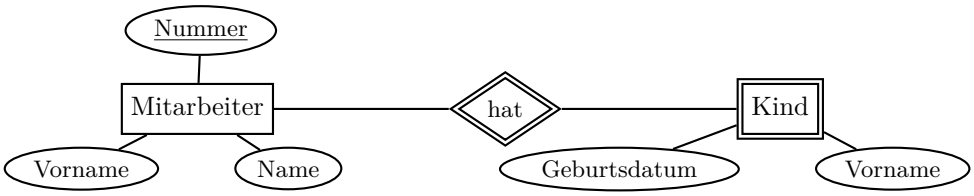
In einem ER-Diagramm modelliert man die Generalisierungsbeziehung folgendermaßen:



10.1.4 Schwache Entitätstypen

Wenn der Fall auftritt, dass ein Entitätstyp keinen eigenen Schlüsselkandidaten besitzt, spricht man von **schwachen Entitätstypen**. Sie nehmen an einer Beziehung mit einem anderen Entitätstyp teil, dem Eltern-Entitätstyp. Jedes Objekt des schwachen Entitätstyp nimmt mindestens an einer Beziehung zum Eltern-Entitätstyp teil. Das Objekt wird über diese Beziehung identifiziert. Der Primärschlüssel des Eltern-Entitätstyps wird auch für den schwachen Entitätstyp verwendet (eventuell durch einen Teilschlüssel erweitert).

In der Regel sind die Objekte des schwachen Entitätstyps von den Objekten des Eltern-Entitätstyp abhängig, d.h. ohne diese sind sie nicht von Interesse bzw. besitzen keine eigene Existenz. Im ER-Diagramm werden schwache Entitätstypen von normalen Entitätstypen dadurch unterschieden, dass ihr Name durch ein doppeltes Rechteck eingeschlossen ist. Die Teilschlüssel werden durch eine gestrichelte Unterstreichung ausgezeichnet.



11 Entwurfsschritt 2: Relationenschema

Als zweiter Schritt beim Entwurf einer Datenbank steht nach der Modellierung die Umsetzung des Modells in ein Relationenschema an. Ziel soll ein Schema sein,

das möglichst direkt in einer konkreten relationalen Datenbank implementiert werden kann.

Die Informationen in diesem Abschnitt sollten schon aus der Vorlesung bekannt sein.

Im Allgemeinen gibt es bei dieser Umsetzung mehrere geeignete Schemavarianten. Eine erste Annäherung für die Umsetzung kann wie folgt aussehen:

- ein Entitätentyp wird in eine Relation mit den gleichen Attributen wie der Entitätentyp überführt
- eine Beziehung wird in eine Relation überführt, deren Attribute die Schlüssel der von ihr verbundenen Relationen sind

Normalerweise gibt es allerdings noch einige andere Dinge zu beachten. Im Folgenden werden ein paar einfache Faustregeln dazu angegeben. Dabei ist zu bedenken, dass in Hinblick auf eine konkrete Anwendung es manchmal sinnvoll sein kann, eine theoretisch nicht optimale Umsetzung zu wählen.

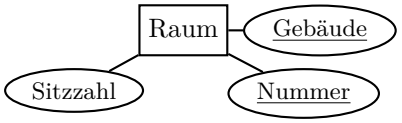
Relationenschemata werden wie in **Kemper/Eickler** beschrieben spezifiziert:

SchemaName : { [$\underline{\text{Attribut}_1}$: Typ_1, Attribut_2: Typ_2, ...] }

Dabei wird der Primärschlüssel einer Relation durch Unterstreichung gekennzeichnet. Attribute innerhalb einer Relation müssen eindeutig benannt sein.

11.1 Entitätentypen

Ein Entitätentyp wird normalerweise als Relation abgebildet, deren Attribute die Attribute dieses Typs sind. Diese Relation hat zunächst einmal keinerlei Verbindung zu den Beziehungen, an denen der Entitätentyp teilnahm.



Räume : {[Nummer: integer, Gebäude: string, Sitzzahl: integer]}

Zusammengesetzte Attribute werden entweder aufgelöst, d.h. die Attribute des zusammengesetzten Attributes werden zu Attributen der Umsetzung des Entitätstyps, oder sie werden als eigene Relation aufgefasst. Diese neue Relation enthält alle Attribute des zusammengesetzten Attributes und alle Primärschlüsselattribute der Ursprungsrelation. Diese Fremdschlüssel bilden den Primärschlüssel der neuen Relation.

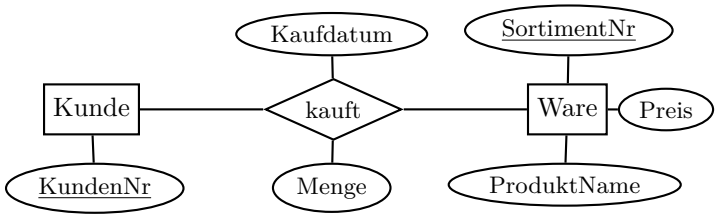
11.2 Beziehungstypen

Ein Beziehungstyp wird üblicherweise als eine eigene Relation abgebildet. Als Attribute für diese Relation werden herangezogen:

- die Attribute des Beziehungstyps

- die Primärschlüsselattribute der teilnehmenden Entitätstypen als Fremdschlüssel

Der Primärschlüssel der Relation enthält die Fremdschlüssel und eventuell zusätzliche Attribute.



kauft : {[KundenNr: integer, SortimentNr: integer,
Kaufdatum: date, Menge: integer]}

Nimmt ein Entitätentyp mehrfach an einer Beziehung teil, so müssen seine Schlüsselattribute entsprechend oft in die Relation übernommen werden, die dieser Beziehung entspricht. Dabei muss man die Attribute umbenennen, um Namenskonflikte zu vermeiden.

11.3 Schwache Entitätstypen

Ein schwacher Entitätstyp wird als eine eigene Relation abgebildet, die zusätzlich zu den eigenen Attributen auch den Primärschlüssel des Vater-Entitätstype als Fremdschlüssel enthält. Dieser bildet zusammen mit den ausgezeichneten Teilschlüsseln des schwachen Entitätstyp den Primärschlüssel der Relation.

Die Beziehung zwischen einem schwachen Entitätstypen und dem Vater-Entitätstyp muss im Allgemeinen überhaupt nicht nachgebildet werden.

11.4 Zusammenführen von Relationen

Die direkte überführung wie oben beschrieben liefert oft nicht die bestmöglichen Relationen. Oft kann man nun noch Relationen zusammenführen. Allgemein gilt: Relationen mit dem gleichen Schlüssel kann man zusammenfassen (aber auch nur diese).

Ein binärer Beziehungstyp R zwischen Entitätentypen E und F, an dem mindestens einer der beiden beteiligten Entitätstypen (sagen wir E) mit Funktionalität 1 teilnimmt, kann mit der E entsprechenden Relation zusammengeführt werden. Dazu führt man eine Relation mit folgenden Attributen ein:

- die Attribute von E
- die Primärschlüsselattribute von F
- die Attribute der Beziehung R

überlegt selbst, warum das vorteilhafter ist, als die überführung der Beziehung R in eine eigene Relation. überlegt auch, was es für Gegenargumente gibt.

12 Entwurfsschritt 3: Normalformen

Normalformen stellen eine theoretische Basis dar, um relationale Datenbankschemata bewerten zu können. Insbesondere sollen durch Normalisierung Redundanzen und implizite Darstellung von Information verhindert werden. Durch Einhalten der Normalformen bei der Modellierung können bei der späteren Benutzung änderungs-, Einfüge- oder Löschungsanomalien verhindert werden.

Die Informationen in diesem Abschnitt sollten schon aus der Vorlesung bekannt sein.

12.1 Erste Normalform

Eine Relation ist in *Erster Normalform (1NF)*, wenn alle Attribute nur atomare Werte annehmen dürfen. Zusammengesetzte oder mengenwertige Attribute sind unter der 1NF nicht zulässig. In klassischen relationalen DBMS lassen sich Relationen, die nicht in 1NF sind, nicht speichern. Die Relation

Prüfung: {Matrikel, Name, Fachbereich, FBNr, {Prüfungsnr, Prfnote}}

ist nicht in 1NF, da es ein zusammengesetzes und mengenwertiges Attribut {Prüfungsnr, Prfnote} gibt.

Nachteile einer Relation, die nicht in 1NF ist, sind unter anderem fehlende Symmetrie, Speicherung redundanter Daten und Aktualisierungsanomalien. Bei der überführung in 1NF entstehende neue Redundanzen werden im Laufe der weiteren Normalisierung beseitigt.

Vorgehen:

- Listen- bzw. mengenwertige Attribute werden durch ihre Elemente ersetzt. Auf Tupelebene wird das durch Ausmultiplikation der Liste mit dem restlichen Tupel erreicht.
- Zusammengesetzte Attribute werden durch die Teilattribute ersetzt.
- Alternativ können auch neue Relationen gebildet werden.

Dies liefert die Relation

Prüfung: {Matrikel, Name, Fachbereich, FBNr, Prüfungsnr, Prfnote}

in der für jede Prüfung mit Prüfungsnote ein eigenes Tupel, d.h. eine eigene Zeile in der Tabelle existiert.

12.2 Zweite Normalform

Eine Relation ist in *Zweiter Normalform (2NF)*, wenn jedes Nichtschlüsselattribut von jedem Schlüssel voll funktional abhängig ist. Relationen, die diese Normalform verletzen, enthalten Information über zwei Entitätstypen, was zu änderungsanomalien führt. Bei der überführung in 2NF werden die verschiedenen Entitätstypen separiert.

Die folgende Relation mit den funktionalen Abhängigkeiten *Matrikel* $\rightarrow$ *Name*, *Matrikel* $\rightarrow$ *Fachbereich*, *Matrikel* $\rightarrow$ *FBNr*, *Fachbereich* $\rightarrow$ *FBNr* und *Matrikel, Prüfungsnr* $\rightarrow$ *Prfnote* ist nicht in 2NF, da Name, Fachbereich und FBNr nur von einem Teil des Schlüssels abhängig sind.

Prüfung: {Matrikel, Name, Fachbereich, FBNr, Prüfungsnr, Prfnote}

Vorgehen:

- Ermittlung der funktionalen Abhängigkeiten zwischen Schlüssel- und Nichtschlüsselattributen wie in der Vorlesung gezeigt
- Eliminierung partieller funktionaler Abhängigkeiten, durch überführung der abhängigen Attribute zusammen mit den Schlüsselattributen in eine eigene Relation und Entfernung der abhängigen Attribute aus der ursprünglichen Relation.

Dieses Verfahren liefert die Relationen:

Student: {Matrikel, Name, Fachbereich, FBNr}

Prüfung: {Matrikel, Prüfungsnr, Prfnote}

12.3 Dritte Normalform

Eine Relation ist in *Dritter Normalform (3NF)*, wenn jedes Nichtschlüsselattribut von keinem Schlüssel transitiv abhängig ist. In der Relation Student liegt die transitive Abhängigkeit *Matrikel* $\rightarrow$ *Fachbereich* $\rightarrow$ *FBNr* vor, sie ist daher nicht in 3NF.

Vorgehen:

- Ermittlung der funktionalen und transitiven Abhängigkeiten
- Für jede transitive funktionale Abhängigkeit $A \rightarrow B \rightarrow C$ werden alle von B funktional abhängigen Attribute entfernt und zusammen mit B in eine eigene Relation überführt.

Dieses Verfahren liefert die Relationen:

Student: {Matrikel, Name, Fachbereich}

Fachbereich: {Fachbereich, FBNr}

13 Arbeiten mit IBM DB2

13.1 Struktur der DB2-Installation

Innerhalb einer DB2-Installation existieren die folgenden wichtigen Datenbanksystem-Objekte:

- Instanzen
- Datenbanken

- Schemata
- Tabellen
- Views

Für Instanzen und Schemata wird auf die Einführung aus Woche 1 verwiesen.

13.1.1 Datenbank

Eine relationale Datenbank ist eine Sammlung von Tabellen. Eine Tabelle besteht aus einer festen Zahl von Spalten, die den Attributen des Relationenschemas entsprechen, sowie einer beliebigen Zahl an Reihen oder Tupeln. Zu jeder Datenbank gehören Systemtabellen, welche die logische und physikalische Struktur der Daten beschreiben, eine Konfigurationsdatei mit den Parametern der Datenbank und ein Transaktionslog.

Eine neue Datenbank wird mit dem DB2-Befehl

```
create database {name}
```

erstellt. Dies erzeugt eine Datenbank mit der Standardkonfiguration. Für die Zwecke des Praktikums sollte es nicht notwendig sein, Änderungen an der Standardkonfiguration vorzunehmen. Gelöscht werden kann die Datenbank wieder mit dem DB2-Befehl

```
drop database {name}
```

(Achtung: der Befehl ist normalerweise nicht rückgängig zu machen und löscht den kompletten Inhalt der Datenbank).

Wie in der Einführung behandelt stellt der DB2-Befehl `connect to {name}` eine Verbindung zu einer Datenbank *name* her, die mit `terminate` wieder beendet wird.

13.1.2 Tabellen und Sichten

Eine relationale Datenbank speichert Daten als eine Menge von zweidimensionalen Tabellen. Jede Spalte der Tabelle repräsentiert ein Attribut des Relationenschemas, jede Zeile entspricht einer spezifischen Instanz dieses Schemas. Daten in diesen Tabellen werden mit Hilfe der Structured Query Language (SQL) manipuliert, einer standardisierten Sprache zur Definition und Manipulation von Daten in relationalen Datenbanken. Sichten sind persistente Anfragen, auf die in anderen Anfragen wie auf eine Tabelle zugegriffen werden kann.

Um sich in einer existierenden Datenbank die vorhandenen Tabellen und Sichten anzeigen zu lassen, benutzt man den DB2-Befehl

```
list tables for schema {name}
```

Mit Hilfe des DB2-Befehls `describe table {schemaname}.{tablename}` kann man Einzelheiten über den Aufbau einer existierenden Tabelle herausfinden. Das Erstellen von Tabellen und Views wird in späteren Kapiteln ausführlich behandelt.

13.2 Benutzerverwaltung

Das Authentifizierungskonzept von DB2 sieht keine gesonderten Benutzernamen vor, sondern baut auf das Benutzerkonzept des Betriebssystems auf. Jede Kleingruppe meldet sich unter dem Namen ihres Accounts bei der Datenbank an, zusätzlich gehört jede Kleingruppe zur Benutzergruppe `students`.

In einer DB2-Instanz gibt es drei Stufen der Zugriffsberechtigung. Als Instanzbesitzer hat jede Gruppe in ihrer Instanz das SYSADM-Recht.

SYSADM: Zugriff auf alle Daten und Ressourcen der Instanz

SYSCTRL: Verwalten der Instanz, aber kein direkter Zugriff auf Daten

SYSMAINT: niedrigste Berechtigungsstufe; kann z.B. Backups erstellen

Um Berechtigungen in der Datenbank, mit der man gerade verbunden ist, zu setzen oder zu verändern, bedient man sich des GRANT-Statements. Umgekehrt wird das REVOKE-Statement benutzt, um Rechte auf einer Datenbank zu entziehen. `GET AUTHORIZATIONS` zeigt die aktuellen Berechtigungen an. Beispiele:

```
GRANT dbadm ON DATABASE TO USER dbpw0301
GRANT connect, createtab ON DATABASE TO GROUP students
GRANT connect ON DATABASE TO public
REVOKE connect ON DATABASE TO public
```

Ein Benutzer oder eine Gruppe mit dem DBADM-Recht besitzt automatische auch alle anderen Rechte auf dieser Datenbank, insbesondere auch das Recht IMPLICIT_SCHEMA. Das Recht DBADM kann nicht an PUBLIC vergeben werden. Nur ein Benutzer mit SYSADM-Recht kann das DBADM-Recht an andere Benutzer oder Gruppen verleihen oder entziehen. Wird einem Benutzer ein Recht entzogen, bedeutet das nicht notwendig, dass dieser das Recht nun nicht mehr besitzt, da Rechte auch durch Zugehörigkeit zu Gruppen bestehen können.

Wer das Recht DBADM oder SYSADM besitzt kann Berechtigungen für ein Schema ändern. Dabei kann einem Benutzer auch das Recht gegeben werden, die erhaltenen Rechte an Dritte weiterzugeben. Der Befehl zum Vergeben bzw. Entziehen von Rechten auf Schemata ist GRANT/REVOKE ... ON SCHEMA.

13.3 DB2: Der Datenbank-Manager

Eine Datenbank-Manager-Instanz verwaltet die Systemressourcen für die Datenbanken, die zu einem bestimmten Systemaccount gehören. Dieser wird mit dem Shell-Befehl `db2start` oder dem DB2-Befehl `start database manager` gestartet. Um Ressourcen des Rechners zu schonen, sollte die Instanz angehalten werden, wenn Ihr längere Zeit nicht mit Ihr arbeitet: hierzu verwendet man den Shell-Befehl `db2stop` oder den DB2-Befehl `stop database manager`.

In DB2 kann man auf verschiedenen Zugriffsebenen Konfigurationswerte einstellen, die das Standardverhalten einer Instanz oder auch einer Datenbank regeln.

Während eine Datenbank-Manager-Instanz läuft, kann man sich deren Konfiguration wie folgt anzeigen lassen:

`get dbm cfg` bzw. `(db2) get database manager configuration`

Einen Konfigurationswert ändert man für eine (laufende) Instanz wie folgt:

`(db2) update dbm cfg using` *config-keyword value*

Dabei steht *config-keyword* für eine Eigenschaft der Konfiguration (z.B. `AUTHENTICATION`), und *value* für den neuen Wert.

Gibt man es nicht explizit anders an, so muss in den meisten Fällen die Datenbank-Manager-Instanz anschließend neu gestartet werden, damit die Konfigurationsänderung aktiv wird.

14 DB2: SQL als DDL

Eine relationale Datenbank speichert Daten als eine Menge von zweidimensionalen Tabellen. Jede Spalte der Tabelle repräsentiert ein Attribut des Relationenschemas, jede Zeile entspricht einer spezifischen Ausprägung dieses Relationenschemas.

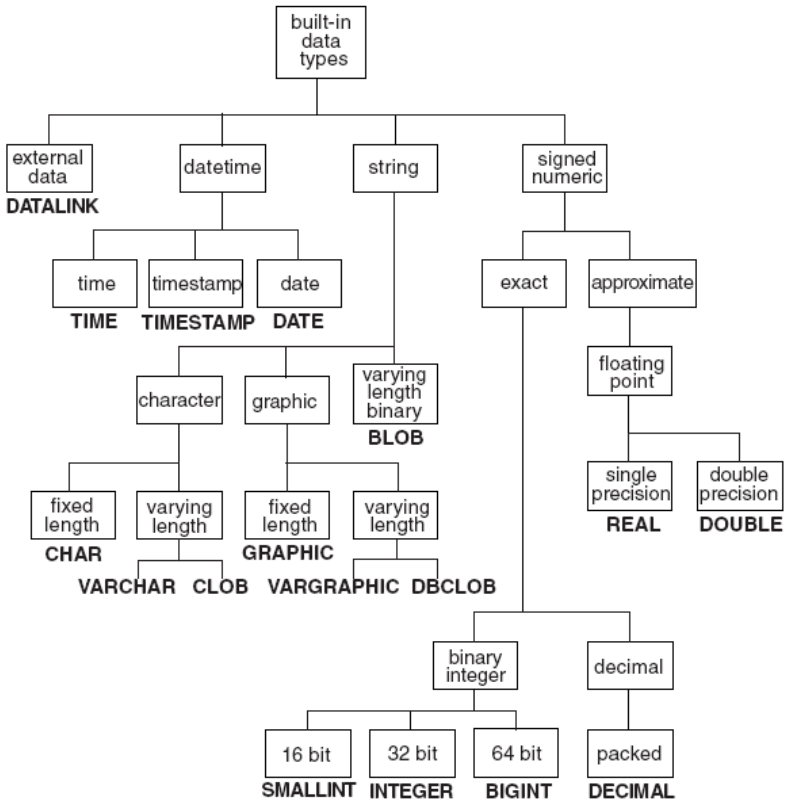
In relationalen Datenbanksystemen wie DB2 hat SQL (Structured Query Language) mehrere Rollen:

- Datendefinition – *Data Definition Language, DDL*
  Definition von Tabellenstrukturen: Erstellen, Anpassen und Löschen von Tabellen mitsamt ihren Attributen, Datentypen und Konsistenzbedingungen
- Datenmanipulation – *Data Manipulation Language, DML*
  Manipulation von Tabelleninhalten: Einfügen, Löschen und ändern von Datensätzen
- Anfragesprache – *Query Language*
  Selektieren von Datensätzen nach bestimmten Auswahlkriterien
- Zugriffskontrolle – *Data Control Language, DCL*
  Diese Rolle haben wir in der vergangenen Woche schon in Form der Rechtevergabe mittels `GRANT` kennengelernt.

Im ersten Block wurde SQL als Anfragesprache benutzt. In diesem Block beschäftigen wir uns mit den anderen drei Rollen von SQL.

14.1 Datentypen

DB2 kennt die in der nachstehenden Grafik abgebildeten Datentypen. Weitere Datentypen können vom Benutzer definiert werden – Stichwort: erweiterbare Datenbanken (dies wird aber nicht mehr im Rahmen des Praktikums behandelt). Beachtet, dass es insbesondere keinen speziellen Datentyp BOOLEAN für Wahrheitswerte gibt.



`SMALLINT`	kleine Ganzzahl
`INTEGER`	Ganzzahl
`BIGINT`	große Ganzzahl
`REAL`	Fließkommazahl mit einfacher Genauigkeit
`DOUBLE`	Fließkommazahl mit doppelter Genauigkeit
`DECIMAL(`p`,`s`)`	Dezimalbruch der Länge p mit s Nachkommastellen
`CHAR(`n`)`	String der Länge n (max. 254)
`VARCHAR(`n`)`	String variabler Länge mit maximal n Zeichen (max. 32672)
`BLOB(`sF`)`	Binary Large Object (z.B. BLOB(2 M) für einen 2MB großen BLOB
`DATE`	Datum
`TIME`	Time
`TIMESTAMP`	Zeitstempel
`DATALINK`	Verweis auf eine Datei außerhalb der Datenbank

Zu jedem Datentyp gehört ein NULL-Wert. Dieser ist von allen anderen Werten des Datentyps zu unterscheiden, da er nicht einen Wert an sich darstellt, sondern das Fehlen eines Wertes anzeigt. Der Wert 0 im Gehaltsfeld eines Angestellten könnte z.B. bedeuten, dass der Angestellte ehrenamtlich tätig ist, während der NULL-Wert bedeuten könnte, dass das Gehalt nicht bekannt ist. IBM DB2 bietet Prädikate an, die es z.B. in Anfragen erlauben, zu prüfen, ob ein NULL-

Wert vorliegt oder eine andere Ausprägung des Datentyps.

Defaultwerte sind Werte, die vom DBMS eingetragen werden, wenn für das betreffende Attribut kein Wert angegeben wird. Sie bieten die Möglichkeit, Attribute automatisch mit Daten zu versehen, z.B. einer bestimmten Konstante, dem Namen des Benutzers (USER), der aktuellen Uhrzeit (CURRENT TIME), dem aktuellen Datum (CURRENT DATE) oder automatisch generierten Werten (GENERATE). Wird kein spezieller Defaultwert angegeben, so wird der NULL-Wert als Defaultwert verwendet.

14.2 Schemata (DB2-spezifisch)

Ein Schema ist eine Sammlung von benannten Objekten (Tabellen, Sichten, Trigger, Funktionen,...). Jedes Objekt in der Datenbank liegt in einem Schema. Objektnamen müssen innerhalb eines Schemas eindeutig sein, ein Schema entspricht also in etwa einem Namensraum. Ein neues Schema kann mit Hilfe des DB2-Befehls `create schema {name}` angelegt, und mit `drop schema {name} restrict` gelöscht werden.

Ein Benutzer, der über die Datenbankberechtigung IMPLICIT_SCHEMA verfügt, kann ein Schema implizit erzeugen, wenn er in einem CREATE-Statement ein noch nicht existierendes Schema verwendet. Dieses neue, von DB2 erzeugte Schema, gehört standardmäßig SYSTEM und jeder Benutzer hat das Recht, in diesem Schema Objekte anzulegen.

14.3 Erstellen von Tabellen

Jedes Team kann aufgrund der vorgegebenen Rechte in den von ihm erstellten Datenbanken neue Tabellen anlegen, eventuell mit impliziter Erstellung eines neuen Schemas (siehe oben). Zum Anlegen einer Tabelle benutzt man das CREATE TABLE-Statement. Der Benutzer, der die Tabelle erstellt, erhält automatisch das CONTROL-Recht auf dieser Tabelle.

Beim Erstellen einer Tabelle wird auch ein Primärschlüssel angegeben. Ein Primärschlüssel besteht aus den angegebenen Attributen, die keine NULL-Werte zulassen dürfen (NOT NULL muss explizit für jedes Attribut des Primärschlüssels angegeben werden) und für die Einschränkungen bezüglich der möglichen Datentypen gelten. DB2 legt automatisch einen Index auf dem Primärschlüssel an. Jede Tabelle kann nur einen Primärschlüssel haben, dieser kann jedoch aus mehreren Attributen bestehen.

```
CREATE TABLE employee (
   id           SMALLINT NOT NULL,
   name         VARCHAR(50),
   department   SMALLINT,
   job          CHAR(10),
   hiredate     DATE,
   salary       DECIMAL(7,2),
   PRIMARY KEY (ID)
)
```

Mit PRIMARY KEY wird für die Tabelle ein Primärschlüssel festgelegt. Dieser besteht aus den angegebenen Attributen. Die Attribute im Primärschlüssel dürfen keine Nullwerte zulassen. Soll der Schlüssel nur aus einem Attribut bestehen, dann genügt es, das Schlüsselwort hinter dem jeweiligen Attribut anzugeben:

```
CREATE TABLE employee (
   id           SMALLINT NOT NULL PRIMARY KEY,
...
```

Als Shortcut kann man Tabellen auch mit folgendem Statement erstellen:

```
CREATE TABLE name LIKE other_table
```

Dies übernimmt die Attribute der Tabelle mit den exakt gleichen Namen und Definitionen aus einer anderen Tabelle oder einem View.

Gelöscht werden Tabellen mit dem DROP-Statement. Dazu benötigt man das CONTROL-Recht für die Tabelle, das DROPIN-Recht im Schema der Tabelle, das DBADM- oder das SYSADM-Recht. Wird eine Tabelle gelöscht, so werden in allen Tabellen, die diese Tabelle referenzieren, die zugehörigen Fremdschlüsselbeziehungen gelöscht. Alle Indexe, die auf der Tabelle bestehen, werden gelöscht.

14.4 Tabellenerstellung: Schlüssel und Constraints

Die im Folgenden beschrieben Konsistenzbedingungen können direkt beim Erzeugen einer Tabelle als Teil des `CREATE TABLE`-Befehls mit angegeben werden.

14.4.1 Primärschlüssel

Mit PRIMARY KEY wird für die Tabelle ein Primärschlüssel festgelegt. Dieser besteht aus den angegebenen Attributen. Die Attribute im Primärschlüssel dürfen keine Nullwerte zulassen. Soll der Schlüssel nur aus einem Attribut bestehen, dann genügt es, das Schlüsselwort hinter dem jeweiligen Attribut anzugeben:

```
id SMALLINT NOT NULL PRIMARY KEY,
```

Hilfreich kann hier sein, wenn man die Werte des Primärschlüssels als `IDENTITY` automatisch vom System generieren lässt.

```
id SMALLINT NOT NULL GENERATED ALWAYS AS IDENTITY,
```

Durch die Schlüsselworte `GENERATED ALWAYS AS IDENTITY` wird DB2 stets einen einzigartigen Wert für dieses Attribut generieren. Dabei erwartet das Datenbanksystem, dass beim Einfügen von Daten keine Werte für dieses Attribut vorgegeben werden - anderenfalls wirft es einen Fehler. Alternativ kann man `GENERATED BY DEFAULT AS IDENTITY` einsetzen, damit das System nur bei Fehlen von vorgegebenen Werten einen einzigartigen Schlüsselwert generiert.

DB2 erlaubt solche automatisch generierten Schlüssel für INTEGER, SMALLINT, BIGINT oder DECIMAL (ohne Nachkommastellen). Es ist möglich zusätzlich anzugeben, mit welchem Wert die automatisch generierten Schlüssel starten und in welchen Schritten sie inkrementiert werden sollen:

```
id      SMALLINT NOT NULL GENERATED ALWAYS AS IDENTITY
                  (START WITH 1, INCREMENT BY 1, NO CACHE),
```

14.4.2 Sekundärschlüssel

Durch das Schlüsselwort UNIQUE wird ein Sekundärschlüssel festgelegt. Auch für diesen wird automatisch ein Index erstellt und es gelten die gleichen Einschränkungen wie für Primärschlüssel. Attribute eines Sekundärschlüssels dürfen nicht schon Teil eines anderen Schlüssels sein. Beispiel:

```
jobnr INTEGER NOT NULL UNIQUE,
```

14.4.3 Fremdschlüssel

Fremdschlüssel werden benutzt, um zu erzwingen, dass ein oder mehrere Attribute einer abhängigen Tabelle nur Werte annehmen dürfen, die auch als Werte eines Schlüssels einer anderen Tabelle auftreten. Dieses Konzept wird auch *Referentielle Integrität* genannt. Der Benutzer muss das REFERENCES- oder ein übergeordnetes Recht auf der referenzierten Tabelle besitzen. Die Tabelle, auf die verwiesen wird, muss existieren und die referenzierten Attribute müssen dort als Schlüssel definiert sein.

Fremdschlüssel werden entweder als *Constraint* (s.u.) oder mit dem Schlüsselwort REFERENCES angegeben. In diesem Beispiel wird durch das Attribut `jobnr` die Spalte `internal_number` der Tabelle `jobs` referenziert:

```
jobnr INTEGER REFERENCES jobs(internal_number)
```

14.4.4 Checks

Bei der Definition eines Tabellenattributes kann zusätzlich zum Datentyp eine Einschränkung der Werte angegeben werden. Dies wird mit dem Schlüsselwort CHECK eingeleitet:

```
department SMALLINT CHECK (department BETWEEN 1 AND 5),
```

```
job CHAR(10) CHECK (job in ('Professor','WiMi','NichtWiMi')),
```

14.4.5 Constraints

Checks und *Fremdschlüssel* können auch als benannte *Constraints* mit dem Schlüsselwort CONSTRAINT angegeben werden:

```
CONSTRAINT jobref FOREIGN KEY jobnr REFERENCES jobs (int_no)
```

```
CONSTRAINT yearsal CHECK (YEAR(hiredate) > 1996 OR SALARY > 1500)
```

14.4.6 Beispiel

Ein Beispiel für das Erzeugen einer Tabelle mit mehreren Konsistenzbedingungen wäre:

```
CREATE TABLE employee (
   id           SMALLINT NOT NULL,
   name         VARCHAR(50),
   department   SMALLINT CHECK (department BETWEEN 1 AND 5),
   job          CHAR(10) CHECK (job in ('Prof','WiMi','NiWiMi')),
   hiredate     DATE,
   salary       DECIMAL(7,2),

   PRIMARY KEY (ID),
   CONSTRAINT yearsal CHECK (YEAR(hiredate) > 1996
                             OR salary > 1500)
)
```

14.5 Index

Ein Index ist ein Datenbankobjekt, das den Zugriff über bestimmte Attribute einer Tabelle beschleunigen soll und dies in den allermeisten Fällen auch tut. Man kann auch (UNIQUE)-Indexe verwenden, um die Einhaltung von Schlüsselbedingungen sicherzustellen.

Werden bei der Tabellendefinition Primärschlüssel- (PRIMARY KEY) oder Sekundärschlüsselbedingungen (UNIQUE) angegeben, so legt DB2 selbständig Indexe an, um die Einhaltung dieser sicherzustellen. Ein einmal angelegter Index wird dann automatisch bei änderungen an den Inhalten der entsprechenden Tabelle gepflegt. Dies ist natürlich mit einem erhöhten Verwaltungsaufwand verbunden.

Näheres zu Indexen und den zur Realisierung verwendeten Datenstrukturen erfährt man in der Vorlesung **Datenbanken** oder in der begleitenden Literatur.

Das expliziten Anlegen eines Index geht mit dem CREATE INDEX-Statement, das Löschen entsprechend über DROP INDEX. Man benötigt dazu mindestens das INDEX- oder das CONTROL-Recht für die Tabelle, sowie das CREATEIN-Recht im angegebenen Schema. Der Benutzer, der einen Index anlegt, erhält auf diesem das CONTROL-Recht.

über das Schlüsselwort UNIQUE kann man bei der Erstellung eines Index angeben, dass keine Tupel in allen Werten der angegebenen Attribute übereinstimmen dürfen. Sollten zum Zeitpunkt der Indexgenerierung Tupel in der Tabelle dagegen verstoßen, so wird eine Fehlermeldung ausgegeben und der Index nicht angelegt.

Es ist nicht möglich, zwei gleiche Indexe anzulegen. Dabei darf ein Index 16 Attribute umfassen und die Attribute dürfen zusammen nicht länger als 1024 Bytes sein. Die Ordnung ASC bzw. DESC gibt an, ob die Indexeinträge in aufsteigender bzw. absteigender Reihenfolge angelegt werden. Nur wenn UNIQUE

angegeben wurde, können über die INCLUDE-Klausel weitere Attribute in den Index aufgenommen werden, für die aber keine Schlüsselbedingung gelten soll.

Beispiel:

```
CREATE INDEX adressen
   ON anwender (adresse)
```

Indexe werden nicht explizit aufgerufen, sondern vom Datenbank-Managementsystem implizit bei der Bearbeitung von Anfragen herangezogen.

14.6 Verändern von Tabellendefinitionen

Zum ändern einer Tabellendefinition sind das ALTER- oder CONTROL-Recht an der zu ändernden Tabelle, das ALTERIN-Recht für das Schema der entsprechenden Tabelle oder ein übergeordnetes Recht nötig. Um eine Fremdschlüsselbeziehung einzuführen oder zu entfernen benötigt man für die referenzierte Tabelle zusätzlich noch das REFERENCES-, das CONTROL-, das DBADM- oder das SYSADM-Recht. Entsprechend muß beim Löschen einer Schlüsselbedingung für alle die zugehörigen Attribute referenzierenden Tabellen die anfangs genannten Rechte bestehen.

Geändert wird eine Tabellendefinition durch ein ALTER TABLE-Statement:

```
ALTER TABLE example.employee
    ALTER name SET DATA TYPE VARCHAR (100)
    ADD hasemail CHAR(1)
    DROP CONSTRAINT yearsal
```

Das Beispiel demonstriert die drei weesentlichen änderungsmöglichkeiten in einem ALTER TABLE-Statement für die (hypothetische) Tabelle EMPLOYEE im Schema EXAMPLE:

- eine Attributdefinition wird geändert

  über die ALTER-Klausel können VARCHAR Attribute vergrößert (aber nicht verkleinert) werden, außerdem kann durch diese Klausel der Ausdruck zur Erzeugung von automatisch generierten Attributwerten verändert werden.

- ein neues Attribut wird hinzugefügt

  Bei der Verwendung der ADD-Klausel gilt das gleiche wie beim Erstellen einer neuen Tabelle. Wird eine neue Schlüsselbedingung hinzugefügt, und existiert noch kein Index hierfür, so wird ein neuer Index angelegt. Die ADD COLUMN-Klausel wird stets zuerst ausgeführt.

- ein benannter CONSTRAINT wird aus der Tabelle gelöscht

  Man kann keine Attribute löschen und keine Constrants oder Checks, die direkt und ohne Bezeichner bei einer Attributdefinition angegeben wurden.

15 Pflege von Tabelleninhalten

15.1 Einfügen von Tupeln

Das Einfügen von neuen Tupeln in eine bestehende Tabelle geschieht mit dem INSERT-Statement. Die Werte der einzufügenden Tupel müssen in Bezug auf Datentyp und Länge zu den jeweiligen Attributen passen. Außerdem müssen die einzufügenden Tupel alle Constraints (Schlüsselbedingungen, Fremdschlüssel und Checks) erfüllen. Wird in eine Sicht eingefügt, so muß diese das zulassen.

Wird beim INSERT-Statement für ein Attribut kein Wert eingetragen, so wird (falls vorhanden) der Defaultwert oder der NULL-Wert eigetragen. Für jedes Attribut, das keinen Defaultwert besitzt und keine NULL-Werte zulässt, muß ein Wert angegeben werden.

```
INSERT INTO category (cat_id, cat_name)
VALUES
     (4, 'Fiction'), (9, 'Poetry')

INSERT INTO werke
    SELECT titel, year
    FROM work
```

15.2 Löschen von Tupeln

Das Löschen von Tupeln geschieht mit dem DELETE-Statement. Dieses löscht alle Tupel aus der aktuellen Sicht, welche die Suchbedingung erfüllen. Man benötigt hierzu das DELETE-Recht oder das CONTROL-Recht auf der Tabelle/Sicht, das DBADM- oder das SYSADM-Recht. Tritt bei der Löschung eines Tupels ein Fehler aus, so bleiben **alle** Löschungen dieses Statements unwirksam.

Wird keine Suchbedingung angegeben, so werden **alle** Tupel gelöscht.

```
DELETE FROM anwender
       WHERE nick='Hase'

DELETE FROM work
       WHERE titel NOT LIKE '%wizard%of%'
         OR year>2000
```

15.3 Ändern von Tupeln

Das Ändern von Tupeln geschieht mit dem UPDATE-Statement. Analog zum DELETE-Statement wird über eine Suchbedingung angegeben, welche Tupel der angegebenen Sicht/Tabelle verändert werden sollen. Wiederum müssen die nötigen Rechte gegeben sein: das UPDATE-Recht auf allen zu verändernden Attributen, das UPDATE-Recht auf der Tabelle/Sicht, das CONTROL-Recht auf der Tabelle/Sicht, das DBADM-Recht oder das SYSADM-Recht.

```
UPDATE enthaelt
  SET sprache = UCASE(sprache)

UPDATE anwender
  SET adresse =
    (SELECT adresse FROM anwender
       WHERE name='Mouse' AND vorname='Minnie')
  WHERE name='Mouse' AND vorname='Mickey'
```

16 Füllen von Tabellen

16.1 Export und Import

Aus einer bestehenden Tabelle kann man Daten in eine Datei exportieren und später diese Daten in eine andere Tabelle wieder einfügen. Dabei benutzt man das EXPORT-Statement.

Beim Export kann die erzeugte Datei entweder eine Textdatei mit Begrenzern sein, aus denen in einer anderen Anwendung wieder eine Tabelle erzeugt werden kann, oder aber eine Datei im IXF (*integrated exchange format*). Letzteres Format kann zusätzliche Informationen über das Schema der Tabelle aufnehmen.

Wenn eine Tabelle exportiert wird, gibt man zusätzlich zur Exportdatei ein SELECT-Statement an. Das einfachste SELECT-Statement

```
SELECT * FROM schema.tabelle
```

exportiert eine vollständige Tabelle mit allen Zeilen und Spalten. Zwei beispielhafte Export-Befehle mit Auswahl bestimmter Spalten und Zeilen der Tabelle sehen etwa so aus:

```
EXPORT TO werke.ixf OF ixf
  SELECT title,year
  FROM work
  WHERE titel like 'B%'

EXPORT TO myfile.del OF del
  modified by chardel'' coldel;
  SELECT cat_ID, cat_name
  FROM category
```

Entsprechend kann eine exportierte Datei auch wieder importiert werden. Am einfachsten geht dies für zuvor als IXF Dateien exportierte Tabellen. Es ist sinnvoll, alle benötigten Daten bereits beim Export in einer Tabelle zu sammeln. Zum Importieren kann man das IMPORT-Statement benutzen.

```
IMPORT FROM werke.ixf OF ixf
  INSERT INTO works
```

Eine andere Möglichkeit ist das mächtigere und komfortablere LOAD-Statement, das im nächsten Abschnitt behandelt wird.

16.2 Load

Das Einspoolen von Daten in eine Tabelle aus einer beliebigen Textdatei kann mit den Kommandos IMPORT oder LOAD geschehen. Dabei ist der Dateiname anzugeben, der Dateityp (ASC = Textdatei ohne Begrenzer, DEL = Textdatei mit Begrenzer, IXF), die Logdatei und zusätzliche Optionen. Die genaue Syntax und die möglichen Optionen sind im Handbuch beschrieben.

Beim Einspoolen von Textdateien (nicht IXF-Dateien) kann man folgende Importmodi benutzen:

`INSERT`: Hinzufügen der neuen Tupel

`INSERT_UPDATE`: Hinzufügen der neuen Tupel, bzw. Ändern alter Tupel mit gleichem Primärschlüssel

`REPLACE`: Löschen aller alten Tupel, Einfügen der neuen Tupel

IXF-Dateien enthalten zusätzliche Schema-Informationen. Hier gibt es noch die Modi `REPLACE_CREATE` und `CREATE`, mit denen wenn nötig eine Tabelle angelegt wird. Zum Einspoolen von Daten in eine Tabelle müssen die nötigen Rechte gesetzt sein.

Angenommen, es existiert eine Tabelle `category` mit den Spalten `cat_id INTEGER NOT NULL`, `cat_name VARCHAR(40) NOT NULL` und `top_cat INTEGER`, sowie eine Textdatei mit Kategorienamen, Kategorienummer und Oberkategorie von Kategorien, jeweils durch ein ‚|' getrennt. Dann würde das folgende LOAD-Statement aus jeder Zeile der Textdatei das 2., 1. und 3. Element auswählen und mit diesen Daten die Tabelle füllen:

```
LOAD FROM "textdatei" OF DEL MODIFIED BY COLDEL|
     METHOD P (2,1,3)
     MESSAGES "logfile"
     INSERT INTO category
       (cat_id, cat_name, top_cat)
```

Es werden also bei diesem Import alle Werte jeder Zeile in vertauschter Reihenfolge hergenommen (um sie der Reihenfolge der Tabellenattribute anzupassen). Durch `INSERT` werden die Daten in die Tabelle hinzugefügt, statt die bestehenden Daten zu überschreiben.

16.2.1 Tabellenintegrität

Die `LOAD`-Operation kann beim Füllen einer Ziel-Tabelle mit generierten Spalten die Tabelle in den `CHECK PENDING` Status versetzen. Um die Integrität generierter Spalten wieder zu herzustellen, muss `SET INTEGRITY` ausgeführt werden.

```
SET INTEGRITY FOR tablename IMMEDIATE CHECKED FORCE GENERATED
```

Falls nach dem Importiewren bei der Überprüfung der Tabellen fehlerhafte Zeilen gefunden werden, müssen diese entfernt werden, bevor man mit der Tabelle weiterarbeiten kann. Solche fehlerhafte Zeilen können beim Importieren entstehen, wenn die importierten Daten die Integritätsbedingungen verletzen. Über-

prüfen kann man die Integrität durch den folgenden DB2-Befehl, der die erste Verletzung auswirft:

```
SET INTEGRITY FOR tablename IMMEDIATE CHECKED
```

Gibt es durch den Import Integritätsfehler in der Tabelle, kann man diese entweder leeren (z.B. Löschen und Neuanlegen oder durch Entfernen aller Tupel) oder reparieren. Will man die Tabelle reparieren ist dazu eine Fehlertabelle nötig, in welche die Zeilen verschoben werden, welche die Integritätsbedingungen verletzen. Diese Tabelle muss den gleichen Aufbau besitzen, wie die eigentliche Tabelle, was sich etwa durch

```
CREATE TABLE exception_table LIKE table
```

erreichen lässt, allerdings darf die Fehlertabelle **keine** Constraints oder generierten Werte besitzen! Eventuell mitkopierte Constraints (referentielle Constraints, Checks oder Schlüssel) müssen gegebenenfalls zunächst gelöscht werden, bevor die Fehlertabelle benutzt werden kann. Zusätzlich kann die Fehlertabelle zwei weitere Spalten enthalten, von denen in der ersten der Zeitstempel festgehalten wird (Datentyp TIMESTAMP) und in der zweiten der Name des durch das Tupel verletzten Integritätsbedingung (Datentype CLOB(32K)).

Nun kann man `SET INTEGRITY` mit der Fehlertabelle aufrufen:

```
SET INTEGRITY FOR tablename IMMEDIATE CHECKED
    FOR EXCEPTION IN tablename USE exception_table
```

Der Befehl `SET INTEGRITY` erlaubt auch das völlige Ausschalten von Integritätsüberprüfungen. Das ist zum Beispiel nützlich, wenn man neue Constraints hinzufügen möchte wie im folgenden Beispiel:

```
SET INTEGRITY FOR table OFF;
ALTER TABLE table ADD CHECK (something < other);
ALTER TABLE table ADD FOREIGN KEY (attribute) REFERENCES other_table;
SET INTEGRITY FOR table IMMEDIATE CHECKED;
```

16.3 Optionen für IDENTITY-Spalten

Falls in der Zieltabelle `IDENTITY`-Spalten definiert sind, dann kann man bei `IMPORT`- und `LOAD`-Befehlen explizit angeben, wie diese Spalten zu behandeln sind.

Mögliche Optionen sind:

- `IDENTITYMISSING` – Diese Option besagt, dass die Quelldaten keine eigene `IDENTITY`-Spalte enthalten. Das System generiert also beim Einfügen automatisch passende, neue Werte.

  ```
  IMPORT FROM werke.ixf OF ixf
    MODIFIED BY IDENTITYMISSING
    INSERT INTO works
  ```

- `IDENTITYIGNORE` – Bei dieser Option sind zwar in den Quelldaten IDs enthalten, diese werden jedoch ignoriert und durch neue, vom System generierte ersetzt.

- `IDENTITYOVERRIDE` – Hier werden die IDs aus den Quelldaten übernommen, selbst wenn sie in der Zieltabelle schon vorhanden sind. Da so unter Umständen vorhandene Daten ersetzt oder modifiziert werden können, ist diese Option nur beim `LOAD`-Befehl verfügbar.

```
LOAD FROM "textdatei" OF DEL
  MODIFIED BY IDENTITYOVERRIDE
    INSERT INTO works
       (titel,year)
```

Weitere Informationen zu `IDENTITY`-Spalten und ihren Zusammenhang mit `IMPORT`, `EXPORT` und `LOAD` findet Ihr im Netz unter:

`http://www.ibm.com/developerworks/db2/library/techarticle/0205pilaka/0205pilaka2.html`

17 Vordefinierte Funktionen

Bei der Arbeit mit IBM DB2-SQL stehen eine Reihe von vordefinierten Funktionen zur Verfügung, mit denen Werte manipuliert werden können. Diese teilen sich in zwei Arten:

Skalare Funktionen: Auswertung einer Liste von skalaren Parametern mit Rückgabe eines skalaren Wertes; werden in Ausdrücken benutzt

Aggregatfunktionen: Anwendung auf Spalten einer Gruppe bzw. einer Relation mit Rückgabe eines skalaren Wertes; werden z.B. in Anfragen benutzt

Man kann auch benutzerdefinierte Funktionen der beiden genannten Arten, sowie Tabellenfunktionen, die ganze Relationen zurückgeben, erstellen. Diese können dann genau wie die Systemfunktionen benutzt werden. Bei allen Funktionen ist darauf zu achten, dass die von ihnen zurückgegebenen Datentypen von den Parametertypen abhängen. Viele der Funktionen sind überladen, d.h. für unterschiedliche Datentypen definiert.

17.1 Skalare Funktionen

Die wichtigsten skalaren Funktionen seien hier genannt, zur Definition und ausführlichen Beschreibung der Funktionen sei auf das Handbuch oder die Online-Dokumentation verwiesen.

`http://publib.boulder.ibm.com/infocenter/db2luw/v9/`

Typkonvertierung: BIGINT, BLOB, CHAR, CLOB, DATE, DBCLOB, DECIMAL, DREF, DOUBLE, FLOAT, GRAPHIC, INTEGER, LONG, LONG_VARCHAR, LONG_VARGRAPHIC, REAL, SMALLINT, TIME, TIMESTAMP, VARCHAR, VARGRAPHIC

Mathematik: ABS, ACOS, ASIN, ATAN, CEIL, COS, COT, DEGREES, EXP, FLOOR, LN, LOG, LOG10, MOD, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQRT, TAN, TRUNC

Stringmanipulation: ASCII, CHR, CONCAT, DIFFERENCE, DIGITS, HEX, INSERT, LCASE, LEFT, LENGTH, LOCATE, LTRIM, POSSTR, REPEAT, REPLACE, RIGHT, RTRIM, SOUNDEX, SPACE, SUBSTR, UCASE, TRANSLATE

Datumsmanipulation: DAY, DAYNAME, DAYOFWEEK, DAYOFYEAR, DAYS, HOUR, JULIAN_DAY, MICROSECOND, MIDNIGHT_SECONDS, MINUTE, MONTH, MONTHNAME, QUARTER, SECOND, TIMESTAMP_ISO, TIMESTAMPDIFF, WEEK, YEAR

System: EVENT_MON_STATE, NODENUMBER, PARTITION, RAISE_ERROR, TABLE_NAME, TABLE_SCHEMA, TYPE_ID, TYPE_NAME, TYPE_SCHEMA

Sonstige: COALESCE, GENERATE_UNIQUE, NULLIF, VALUE

18 Ausdrücke

Ausdrücke werden in SELECT-Klauseln und in Suchbedingungen benutzt, um Werte darzustellen. Sie setzen sich aus einem oder mehreren *Operanden*, Klammern und unären oder binären *Operatoren* zusammen: + (Summe, positiver Wert), - (Differenz, negativer Wert), * (Produkt), / (Quotient) und || (Stringkonkatenation).

Als Operanden sind erlaubt:

Attributnamen: möglicherweise qualifiziert durch Tabellennamen oder Tupelvariablen

Konstanten: ganze Zahlen (12, -10), Dezimalzahlen (2.7, -3.24), Fließkommazahlen (-12E3, 3.2E-12), Zeichenketten ('hello'), Datumswerte ('12/25/1998','25.12.1998','1998-12-25'), Zeitwerte ('13.50.00', '13:50', 1:50 PM'), Zeitstempel ('2001-01-05-12.00.00.000000')

Funktionen

Zeitdauerangaben: bei arithmethischen Operationen mit Datums-, Zeit- oder Zeitstempelwerten können Zeitdauern benutzt werden, z.B. 5 DAYS, 1 HOUR (MONTH/S, DAY/S, HOUR/S, SECOND/S, MICROSECOND/S)

Registerwerte: Hierunter fallen

CURRENT DATE: aktuelles Datum

CURRENT TIME: aktuelle Uhrzeit

CURRENT TIMESTAMP: aktueller Zeitstempel

CURRENT TIMEZONE: aktuelle Zeitzone

CURRENT NODE: aktuelle Nodegruppennummer

CURRENT SCHEMA: aktuelles Schema

CURRENT SERVER: Servername

USER: Benutzername

```
SELECT title
FROM book
WHERE YEAR(pubyear) > 1980
```

Typumwandlungen: Explizite Umwandlung in einen bestimmten Typ, z.B. CAST (NULL AS VARCHAR(20)) oder CAST(Gehalt*1.2345 AS DECIMAL(9,2))

Subqueries: liefert eine Anfrage immer einen skalaren Wert zurück, dann kann man diese als Operand benutzen

Fallunterscheidungen: z.B.

```
CASE format
    WHEN 'Hardcover' THEN 'Gebundenes Buch'
    WHEN 'Mass Market Paperback'
         THEN 'Kleinformatiges Taschenbuch'
    WHEN 'Paperback' THEN 'Taschenbuch'
END

CASE
    WHEN YEAR(pubyear) > 1945 THEN 'Nach WK II'
    WHEN YEAR(pubyear) > 2008 THEN 'Huch!?'
ELSE 'Vor dem Ende des 2. WK'
END
```

Die Bedingungen werden der Reihe nach ausgewertet. Ergebnis ist der erste als wahr evaluierende Fall, sonst der Wert in der ELSE-Klausel. Gibt es keine ELSE-Klausel so wird als Defaultwert NULL genommen. Der Ergebnisausdruck muß die Bestimmung des Datentyps des Gesamtausdrucks zulassen, insbesondere müssen die Ergebnistypen der einzelnen Ausdrücke kompatibel und dürfen nicht alle NULL sein.

19 Prädikate

Prädikate sind logische Tests, die in Suchbedingungen verwendet werden können. Sie können als Werte *true*, *false* und *unknown* annehmen. Die bekannten Prädikate sind:

- Vergleichsprädikate: =, <>, <, >, <=, >= (Vergleiche mit NULL-Werten ergeben *unknown*)
- NOT als Verneinung eines Prädikates
- BETWEEN, z.B. runtime BETWEEN 1 AND 3
- NULL, z.B. dateofdeath IS NOT NULL

- LIKE, Zeichenkettenvergleich mit Platzhaltern _ für ein Zeichen oder % für beliebig viele: name LIKE 'Schr_der%' würde z.B. auf 'Schröder', 'Schrader' oder 'Schrederbein' passen
- EXISTS, Test auf leere Menge
- IN, Test auf Vorkommen in einer Kollektion von Werten

Es ist möglich, Vergleichsprädikate durch SOME, ANY oder ALL zu quantifizieren und dadurch ein einzelnes Tupel mit einer Menge von Tupeln zu vergleichen. Beispiele:

```
category IN ('Fiction','Mythology','Non-Fiction','Poetry')

('2002,'Reaper Man') = ANY (SELECT year(pubyear),
                                        title
                                  FROM book)

EXISTS (SELECT *
        FROM book
        WHERE pubyear < 2000)

1000000000 > ALL (SELECT bevölkerung FROM land)
```

Transaktionen

Als neues Konzept sollen nun zusätzlich Transaktionen betrachtet werden. Eine Transaktion ist eine Zusammenfassung von Datenbank-Operationen, im Extremfall von genau einer Operation. Transaktionen in Datenbanken sind charakterisiert durch die sogenannten **ACID**-Eigenschaften. Das bedeutet, eine Transaktion soll den vier folgenden Eigenschaften genügen:

A tomicity (Atomarität),

C onsistency (Konsistenz),

I solation, und

D urability (Dauerhaftigkeit)

Eine Transaktion soll also entweder als Ganzes oder gar nicht wirksam werden; die korrekte Ausführung einer Transaktion überführt die Datenbank von einem konsistenten Zustand in einen anderen; jede Transaktion soll unbeeinflußt von anderen Transaktionen ablaufen; und die Änderungen einer wirksam gewordenen Transaktion dürfen nicht mehr verloren gehen.

In SQL gibt es zur Unterstützung von Transaktionen verschiedene sprachliche Konstrukte. Im Allgemeinen haben wir DB2 bisher so benutzt, dass alle Anweisungen automatisch implizit als Transaktionen behandelt wirden. Dies wird durch die Option `-c` (*auto commit*) beim Aufruf des *command line processors* (CLPs) erreicht. Alle Änderungen eines jedes SQL-Statements werden **sofort** durchgeschrieben und wirksam.

Ruft man den CLP mit `+c` auf, dann wird das *auto commit* deaktiviert. Nun gilt Folgendes:

- Jeder lesende oder schreibende Zugriff auf die Datenbank beginnt implizit eine neue Transaktion.
- Alle folgenden Lese- oder Schreibvorgänge gehören zu dieser Transaktion.
- Alle eventuellen Änderungen innerhalb dieser Transaktion gelten zunächst einmal nur **vorläufig**.
- Der Befehl `commit` schreibt die Änderungen in die Datenbank durch. Zu diesem Zeitpunkt erst müssen Integritätsbedingungen eingehalten werden. Zwischendurch kann also auch eine Integritätsbedingung verletzt werden.
- Commit macht die von den folgenden Befehlen getätigten Änderungen wirksam: ALTER, COMMENT ON, CREATE, DELETE, DROP, GRANT, INSERT, LOCK TABLE, REVOKE, SET INTEGRITY, SET transition variable und UPDATE.
- Solange diese Transaktion nicht durchgeschrieben wurde, nimmt der Befehl `rollback` alle Änderungen seit dem letzten **commit** als Ganzes zurück.

Im Mehrbenutzerbetrieb auf einer Datenbank, etwa wenn gleichzeitig mehrere Benutzer über eine Webseite auf die Datenbank zugreifen und Daten ändern oder einfügen, kann es durch die Konkurrenz zweier Transaktionen zu Problemen oder Anomalien kommen. Typisches Beispiel sind gleichzeitige Änderungen an einer Tabelle durch zwei unabhängige Benutzer:

Phantomtupel: Innerhalb einer Transaktion erhält man auf die gleiche Anfrage bei der zweiten Ausführung zusätzliche Tupel.

Nichtwiederholbarkeit: Das Lesen eines Tupels liefert innerhalb einer Transaktion unterschiedliche Ergebnisse.

“Schmutziges” Lesen: Lesen eines noch nicht durchgeschriebenen Tupels.

Verlorene Änderungen: Bei zwei gleichzeitigen Änderungen wird eine durch die andere überschrieben und geht verloren.

Daher kann man in DB2-SQL zum einen den Isolationsgrad setzen, mit dem man arbeitet, indem man **vor** Aufnahme der Verbindung mit einer Datenbank den Befehl CHANGE ISOLATION benutzt. Es gibt vier Isolationsgrade, die unterschiedlich gegen die genannten Anomalien schützen:

	RR	RS	CS	UR
Phantomtupel	nein	ja	ja	ja
Nichtwiederholb.	nein	nein	ja	ja
Schmutziges Lesen	nein	nein	nein	ja
Verlorenes Update	nein	nein	nein	nein

Außerdem lassen sich explizit Tabellen oder eine gesamte Datenbank sperren, indem man den Befehl LOCK TABLE benutzt, bzw. die Verbindung mit der Datenbank unter Angabe des Sperrmodus herstellt. SHARE-Sperren sind der Standard und bedeuten, dass niemand anderes eine EXCLUSIVE-Sperre anfordern kann. Beispiele:

```
LOCK TABLE land IN EXCLUSIVE MODE;

CONNECT TO almanach IN SHARE MODE USER username;
```

20 Webanwendungen in Java mit JSPs

20.1 Idee und Hintergrund

Eine Webanwendung (Webapp) ist ein Programm, das serverseitig läuft und mit dem Benutzer über HTTP kommuniziert. Häufig wird die Kommunikation benutzerseitig über einen Webbrowser durchgeführt, der Anfragen an die Webapp sendet, indem er bestimmte URLs aufruft, und die Antworten der Webapp als HTML-Seiten anzeigt.

Wenn man in Java implementieren will, schreibt man i.d.R. Code, der am Ende in einem Servlet-Container, wie z.B. Apache Tomcat, läuft. Dazu erweitert man eine Klasse `HttpServlet` und berechnet die Antwort, die dem Benutzer übermittelt wird, als Aneinanderreihung von Strings:

Listing 1: beispiel.java

```
1 StringBuffer out = new StringBuffer();
2 out.append("<html>");
3 out.append("<head>").append("<title>Titel</title>").append("</head>");
4 out.append("<body>");
5 out.append("<h1>überschrift</h1>");
6 String result = calculate();
7 out.append("<p>Ergebnis: ").append(result).append("</p>");
8 out.append("</body>");
9 out.append("</html>");
```

Diese Methode hat mehrere Nachteile. Einer ist, dass das HTML in kleinen Fragmenten zwischen sehr viel Java-Code zerstreut und damit der Zusammenhang der einzelnen HTML-Teile verloren geht. Ein anderer Nachteil ist, dass die Arbeitsteilung zwischen Java-Entwickler und Designer oder Frontend-Entwickler (der oft allein für den HTML-Code zuständig ist) schlecht unterstützt ist: Wenn der Frontend-Entwickler sein HTML abgegeben hat, muss der Java-Entwickler die Seite mühevoll zerschneiden, um sie in den Java-Code zu integrieren. Wenn danach eine Änderung am HTML durchgeführt werden muss (der Kunde möchte z.B. eine andere Farbe), ist es teilweise sehr schwierig, die Änderung im Java-HTML-Mix nachzuvollziehen und der Code muss evtl. komplett neu geschrieben werden.

Um diese Nachteile (und die relativ aufwändige Definition von Servlets über XML-Dateien) zu relativieren, wurden Java Server Pages (JSP) entwickelt. Die Grundidee ist, Java-Code in HTML-Code einzubetten. Obiges Beispiel sieht in JSP aus wie folgt:

Listing 2: beispiel.jsp

```
1 <html>
2 <head><title>Titel</title></head>
3 <body>
```

```
4 <h1>überschrift</h1>
5 <p>Ergebnis:<%=calculate()%></p>
6 </body>
7 </html>
```

Eine solche JSP-Datei wird direkt im Servlet-Container hinterlegt und bei Aufruf durch einen Benutzer automatisch in Java-Code umgewandelt, übersetzt und als Servlet verwendet, ohne dass der Benutzer oder der Entwickler davon etwas merkt.

20.2 Webapps schreiben

20.2.1 JSP

Beim ersten Aufruf einer JSP-Seite wird aus dem Inhalt der `.jsp`-Datei in einem ersten Schritt (automatisch) ein Servlet erzeugt. Das ist eine Java-Klasse, die eine Oberklasse um die entsprechende Logik erweitert. Die Logik wird in eine Methode `doGet()` eingefügt. Dazu gehört auch das Zusammensetzen der Ausgabe.

Aus obigem Test-JSP ergibt sich vereinfacht folgende Servlet-Klasse:

Listing 3: beispiel.java

```
1  package jsp_servlet;
2  import java.util.*;
3  /* ... */
4
5  /* 1 */
6  class _myservlet
7  implements javax.servlet.Servlet, javax.servlet.jsp.HttpJspPage {
8  /* 2 */
9  public void _jspService(
10   javax.servlet.http.HttpServletRequest request,
11   javax.servlet.http.HttpServletResponse response )
12   throws javax.servlet.ServletException, java.io.IOException
13   {
14     javax.servlet.jsp.JspWriter out = pageContext.getOut();
15     HttpSession session = request.getSession( true );
16     try {
17       /* 3 */
18       out.print( "Hallo, Welt." );
19     } catch ( Exception _exception ) {
20       /* ... */
21     }
22   }
23 }
```

Mithilfe verschiedener JSP-Tags kann man Code an unterschiedliche Stellen dieser Servlet-Klasse stellen. Der JSP-Standard kennt ein paar Tags, die besondere Funktionen ausführen.

<%@...%> Tags dieser Sorte erzeugen Code, der an der Stelle `/* 1 */` im Beispiel-Code eingefügt wird. Allerdings ist die Syntax festgelegt. Ein Beispiel für den Import von Packages ist `<%@ page import="java.sql.*,com.*" %>`.

<%!...%> Der Inhalt dieser Tags wird in den Klassen-Body eingefügt, gleichberechtigt mit z.B. Objektvariablen an der Stelle `/* 2 */`. Hier kann man z.B. Methoden definieren.

<%...%> Der Inhalt dieser Tags wird in die Methode eingefügt, die eine HTTP-Abfrage bearbeitet – an der Stelle `/* 3 */`, aber ohne `out.print()`. Der Inhalt sind Java-Statements, die durch Semikolon („;“) beendet werden müssen. Beispiel: `<% int i = 1;%>`.

<%=...%> Der Inhalt dieser Tags wird an der entsprechenden Stelle in den Ausgabestrom geschrieben – bei Objekten über den impliziten Aufruf der `toString()`-Methode. Im Beispiel ist das bei `out.print("Hallo, Welt.");`. Da hier nur ein Ausdruck stehen darf, endet der Inhalt dieser Tags *nicht* mit Semikolon. Beispiel: `<%=i%>`.

<%--...--%> Dieses Tag umschließt einen Kommentar, der nicht in die (HTML-)Ausgabe geschrieben wird.

20.2.2 Servlets

Standardmäßig kommunizieren Browser und Server im Web über das HTTP-Protokoll. Dabei sendet der Browser Befehle an den Server, um z.B. Dokumente anzufordern oder abzuspeichern. Für das Praktikum konzentrieren wir uns auf die HTTP-Operationen **GET** und **POST**. GET wird vom Browser verwendet, um ein Dokument anzufordern. POST, um Daten an den Server zu senden.

Ein Servlet wird über eine vordefinierte URL aufgerufen. Die entsprechende Klasse im Server stellt Methoden bereit, um auf HTTP-Operationen vom Browser zu antworten. Die Mindestanforderung an eine Servlet-Klasse ist, dass sie Implementierungen sowohl für GET als auch für POST bereit stellt.

Der folgende Programmcode demonstriert die Implementierung eines Servlets, welches den Text „Willkommen!“ ausgibt. Aufgerufen werden kann es über den Pfad **/HelloServlet** (z.B. http://localhost:8080/ServletTest/**HelloServlet**).

Listing 4: HelloServlet.java

```
1  import java.io.IOException;
2  import javax.servlet.ServletException;
3  import javax.servlet.annotation.WebServlet;
4  import javax.servlet.http.HttpServlet;
5  import javax.servlet.http.HttpServletRequest;
6  import javax.servlet.http.HttpServletResponse;
7
8  @WebServlet("/HelloServlet")
9  public class HelloServlet extends HttpServlet {
10
11   private static final long serialVersionUID = 1L;
12
13   public HelloServlet() {
14       super();
15   }
16
17   protected void doGet(HttpServletRequest request, HttpServletResponse
18                        response) throws ServletException, IOException {
19       response.getWriter().append("Willkommen!");
```

```
20   }
21
22
23   protected void doPost(HttpServletRequest request, HttpServletResponse
24                         response) throws ServletException, IOException {
25       doGet(request, response);
26   }
27 }
```

20.3 Datenbankzugriff mit Java: JDBC

20.3.1 Einführung

Zum Zugriff auf die DB2-Datenbank benutzen wir JDBC (*Java DataBase Connectivity*). JDBC ist eine einheitliche Datenbankschnittstelle für Java, die es für den Anwendungsentwickler transparent macht, ob hinter dieser Schnittstelle auf eine DB2-, Oracle-, MySQL-, oder sonst eine Datenbank zugegriffen wird. Die Verbindung wird durch einen Treiber hergestellt, der erst zur Laufzeit in das Programm eingebunden wird.

Unter JDBC greift man über eine URL auf die Datenbank zu, deren Aufbau folgendermassen aussieht: `jdbc:subprotocol:[//host:port/]database`. Host und Port sind also optional. Das genaue Format für die Verbindung zu den im Praktikum verwendeten DB2-Instanzen wird im Abschnitt 20.4 vorgestellt.

Die eigentliche Verbindung wird über die Klasse `DriverManager` aufgebaut, spezifiziert durch das Interface `Connection`. Ein Programm kann dabei auch Verbindungen zu mehreren Datenbanken offenhalten, wobei jede Verbindung durch ein Objekt realisiert wird, welches das Interface `Connection` implementiert. Dieses bietet unter anderem folgende Methoden:

- `createStatement()`: erzeugt Objekte, die das Interface `Statement` implementieren
- `createPreparedStatement(String)`: erzeugt Objekte, die das Interface `PreparedStatement` implementieren
- `commit()`: schreibt Transaktionen durch
- `rollback()`: nimmt Transaktionen zurück
- `setAutoCommit(boolean)`: setzt die Eigenschaft `autoCommit` für Transaktionen auf wahr oder falsch
- `close()`: schließt eine offene Verbindung

SQL-Anweisungen werden über das Interface `Statement` und `PreparedStatement` benutzt. Die Schnittstellen definieren eine Reihe von Methoden, die es erlauben, Anweisungen auf der Datenbank auszuführen:

- `executeQuery(String)`: führt ein SQL-Statement aus (wird als Paramater angeben) und liefert ein Ergebnis zurück
- `executeQuery()`: führt ein SQL-Statement aus und liefert ein Ergebnis zurück

- `executeUpdate(String)`: führt INSERT, UPDATE, DELETE aus, dabei ist die Rückgabe die Anzahl der betroffenen Tupel oder nichts

Im Interface `ResultSet` werden die Funktionalitäten zur Weiterverarbeitung von Anfrageergebnissen definiert. Auf Objekten, die dieses Interface implementieren, stehen u.a. folgende Methoden zur Verfügung:

- `next()`: geht zum ersten bzw. nächsten Tupel des Ergebnisses
- `getMetaData()`: liefert ein Objekt zurück, welches das Interface `ResultSetMetaData` implementiert, hierüber können Informationen über den Aufbau des Ergebnisses ausgelesen werden
- `findColumn(String)`: findet zu einem Attributnamen die Position im Ergebnistupel
- `get<Typ>(int)` oder `get<Typ>(String)`: lesen die (benannte) Spalte des Ergebnistupels aus und liefern ein Objekt vom angegeben Typ zurück

Die verschiedenen in SQL bekannten Datentypen werden auf Java-Klassen abgebildet:

SQL-Typ	**Java-Typ**
CHAR, VARCHAR, LONG VARCHAR	String
DECIMAL, NUMERIC	java.math.BigDecimal
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
DOUBLE, FLOAT	double
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.TimeStamp

20.3.2 Parameter in SQL-Anweisungen

Sollen in einer SQL-Anweisung Parameter verwendet werden, so muss unbedingt ein `PreparedStatement` statt eines „herkömmlichen“ `Statement`s verwendet werden. Beispielsweise kann mit der SQL-Anfrage `SELECT name FROM user WHERE login = 'meier'` der Name des Benutzers `meier` ermittelt werden. Der Wert für das Login könnte etwa von einem Benutzer stammen, der sein Login in ein HTML-Formular eingetragen hat. Dieser Wert darf nicht ohne weiteres für eine SQL-Anfrage verwendet werden, da die Eingabe des Benutzers selbst SQL-Bestandteile enthalten könnte (sog. SQL Injection). Ein Fehler der häufig begangen wird ist, dass die Eingabe des Benutzers ohne zusätzliche Überprüfungen für das Erzeugen einer SQL-Anweisung verwendet wird (z.B. durch einfache Konkatenation von Zeichenketten). Stattdessen bieten sich `PreparedStatement`s an, die das Überprüfen und Escapen von Parametern automatisch durchführen. Im Folgenden wird ein Beispiel für den fehlerhaften und korrekten Umgang mit Parametern gegeben:

Listing 5: Fehlerhafte Behandlung von Parametern in SQL-Anweisungen

```
1 import java.sql.Connection;
```

```
2  import java.sql.DriverManager;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6
7  public class Sample {
8
9      public static void main(String[] args) {
10         ...
11         try {
12             ...
13             Statement st = db2Conn.createStatement();
14             String myQuery = "SELECT name FROM user " +
15                 "WHERE login = '" + loginFromForm + "'";
16             ResultSet resultSet = st.executeQuery(myQuery);
17             ...
18         }
19         catch (SQLException e) {
20             e.printStackTrace();
21         } finally {
22             // Resourcen schließen
23             ...
24         }
25     }
26 }
```

Listing 6: Korrekte Behandlung von Parametern in SQL-Anweisungen

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6
7  public class Sample {
8
9      public static void main(String[] args) {
10         ...
11         try {
12             ...
13             String myQuery = "SELECT name FROM user WHERE login = ?";
14             PreparedStatement st = db2Conn.prepareStatement(myQuery);
15             st.setString(1, loginFromForm);
16             ResultSet resultSet = st.executeQuery();
17             ...
18         }
19         catch (SQLException e) {
20             e.printStackTrace();
21         } finally {
22             // Resourcen schließen
23             ...
24         }
25     }
26 }
```

20.3.3 Schließen von DB-Resourcen

Verbindungen zur Datenbank müssen mit JDBC explizit geschlossen werden, wenn diese nicht mehr verwendet werden. Dies geschieht in der Regel in einem

`finally`-Block, der sich einem `try`-Block anschließt, der den Datenbank-Code umschließt. Da das Schließen z.B. von DB-Verbindungen häufig durchgeführt werden muss, empfiehlt es sich, den entsprechenden Code in eine Hilfsmethode auszulagern. Das nachfolgende Beispiel zeigt, wie DB-Resourcen korrekt geschlossen werden. Werden Resourcen nicht richtig geschlossen oder werden Resourcen an der falschen Stelle geschlossen kann dies zu Fehlern führen (etwa Verbindungabbruch wegen zu vielen offenen Verbindungen zur Datenbank).

Listing 7: Korrektes Schließen von DB-Resourcen

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6
7  public class Sample {
8
9      public static void main(String[] args) {
10         Connection db2Conn = null;
11         try {
12             b2Conn = DriverManager.getConnection(...);
13             ...
14             String myQuery = "SELECT name FROM user WHERE login = ?";
15             PreparedStatement st = db2Conn.prepareStatement(myQuery);
16             st.setString(1, loginFromForm);
17             ResultSet resultSet = st.executeQuery();
18             ...
19         }
20         catch (SQLException e) {
21             e.printStackTrace();
22         } finally {
23             // Resourcen schließen
24             if (db2Conn != null) {
25                 try {
26                     db2Conn.close();
27                     } catch (IOException e) {
28                         e.printStackTrace();
29                     }
30             }
31         }
32     }
33 }
```

20.3.4 Transaktionen

Auf die korrekte Verwendung von Transaktionen ist ebenfalls zu achten. Standardmäßig ist bei Verbindungen zur Datenbank die Eigenschaft `autoCommit` auf `true` gesetzt. Das heißt, dass nach jeder SQL-Anweisung die Transaktion ein `commit` durchführt. Nun kann es aber notwendig sein, die Steuerung der Transaktionen manuell zu übernehmen, da mehrere SQL-Anweisungen in der selben Transaktion laufen sollen. Ein Beispiel hierfür ist etwa das Einfügen von Daten in zwei Tabellen. Schlägt das Einfügen der Daten in der zweiten Tabelle fehl, so würden sich inkonsistente Daten in der Datenbank befinden, da die Daten für die erste Tabelle schon in der Datenbank sind (die Transaktion führt ein automatisches `commit` nach der ersten SQL-Anweisung durch). Um dieses Pro-

blem zu umgehen, muss sichergestellt sein, dass beide SQL-Anweisungen in der selben Transaktion laufen, die erst ein `commit` durchführt, wenn beide Anweisungen erfolgreich ausgeführt wurden. Im Fehlerfall sollte die Transaktion ein `rollback` durchführen. Die nachfolgenden zwei Beispiele zeigen die fehlerhafte bzw. die korrekte Verwendung von Transaktionen mit JDBC.

Listing 8: Fehlerhafte Benutzung von Transaktionen

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6
7  public class Sample {
8
9      public static void main(String[] args) {
10         User user = ...;
11         Connection db2Conn = null;
12         try {
13             db2Conn = DriverManager.getConnection(...);
14             ...
15             final String sql1 = "INSERT INTO user VALUES (?,?,?,?)";
16             PreparedStatement st = db2Conn.prepareStatement(sql1);
17             st.setString(1, user.getName());
18             st.setInt(2, ...);
19             ...
20             st.execute();
21             ...
22             final String sql2 = "INSERT INTO user_roles VALUES (?,?)";
23             PreparedStatement st2 = db2Conn.prepareStatement(sql2);
24             st2.setString(1, ...);
25             ...
26             st2.execute();
27             ...
28         }
29         catch (SQLException e) {
30             e.printStackTrace();
31         } finally {
32             // Resourcen schließen
33             if (db2Conn != null) {
34                try {
35                    db2Conn.close();
36                    } catch (IOException e) {
37                       e.printStackTrace();
38                    }
39             }
40         }
41     }
42 }
```

Listing 9: Korrekte Benutzung von Transaktionen

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6
7  public class Sample {
8
9      public static void main(String[] args) {
```

```
10          User user = ...;
11          Connection db2Conn = null;
12          try {
13              db2Conn = DriverManager.getConnection(...);
14              db2Conn.setAutoCommit(false);
15              ...
16              final String sql2 = "INSERT INTO user VALUES (?,?,?,?)";
17              PreparedStatement st = db2Conn.prepareStatement(sql1);
18              st.setString(1, user.getName());
19              ...
20              st.execute();
21              ...
22              final String sql2 = "INSERT INTO user_roles VALUES (?,?)";
23              PreparedStatement st2 = db2Conn.prepareStatement(sql2);
24              st2.setString(1, ...);
25              ...
26              st2.execute();
27              ...
28              // Transaktion führt Commit durch
29              db2.commit();
30          }
31          catch (SQLException e) {
32              // Rollback der Transaktion bei Fehler
33              db2Conn.rollback();
34              e.printStackTrace();
35          } finally {
36              // Resourcen schließen
37              if (db2Conn != null) {
38                 try {
39                     db2Conn.close();
40                     } catch (IOException e) {
41                        e.printStackTrace();
42                     }
43              }
44          }
45      }
46  }
```

20.4 Verbindung mit DB2

Beispiel-Code für das Aufbauen einer Verbindung zur lokalen DB2 mit Abfrage von Daten ist im folgenden Listing angegeben. Damit Java den DB2-Treiber findet, muss `db2jcc.jar` im Webapp-Verzeichnis unter `WEB-INF/lib` vorhanden sein. Die Datei findet Ihr im Verzeichnis `~/sqllib/java/` – dabei steht ~ für das Home-Verzeichnis des Benutzers.

In dem untigen Code bauen wir eine Datenbankverbindung zur Datenbank `mondial` auf und lesen aus der Datenbank die Hauptstadt von Deutschland aus.

Listing 10: connection.java

```
1  <%@ page import="java.sql.*" %>
2  <%!
3  private String getCapital(String country) {
4    String out="";
5    try {
6      Class.forName("com.ibm.db2.jcc.DB2Driver");
```

```
7       Connection connection =
8         DriverManager.getConnection("jdbc:db2:mondial");
9
10      String stStr = "SELECT capital " +
11        "FROM dbmaster.country WHERE name=?";
12      PreparedStatement stmt = connection.prepareStatement(stStr);
13      stmt.setString(1, country);
14      ResultSet rs = stmt.executeQuery();
15
16      StringBuffer outb = new StringBuffer();
17      while (rs.next()) {
18        String name = rs.getString("capital");
19        outb.append(name).append("<br/>");
20      }
21      out = outb.toString();
22
23    } catch (Exception e) {
24      e.printStackTrace();
25      return "#fail";
26    }
27    return out;
28  }
29  %>
30  <h1>Hauptstadt</h1>
31  <%= getCapital("Germany") %>
```

Achtet bitte darauf, dass bei den PreparedStatements keine Anführungszeichen um die Platzhalter gehören. Also nicht `WHERE name='?'`, sondern `WHERE name=?`. SQL-Statements mittels String-Operationen zusammenzubauen ist keine gute Idee.

20.5 HTML-Formulare

Um Benutzereingaben aufzunehmen und an ein Servlet oder eine JSP zu übergeben, können HTML-Formularelemente zum Einsatz kommen. Beim Abschicken des Formulars wird dann das Servlet `test` aufgerufen und der Inhalt der Formularelemente per POST oder GET übergeben. Mehr zu HTML und HTML-Formularen findet man z.B. unter

- `http://www.w3.org/TR/html4/` oder
- `http://de.selfhtml.org/`.

Das folgende Beispiel für ein HTML-Formular, das ein Servlet aufruft, zeigt verschiedene Formularelemente, die genutzt werden können:

Listing 11: form.html

```
1   <form enctype="multipart/form-data" action="test" method="post">
2     <table>
3       <tr><td colspan="2">Beispiel</td></tr>
4       <tr>
5         <td><b>Matrikelnummer:</b></td>
6         <td><input type="text" name="matrikel" size="60"/></td>
7       </tr>
8       <tr>
9         <td><b>Aufgabe 1:</b></td>
10        <td><textarea name="aufgabe_1" rows="15" cols="60"></textarea>
```

```
11              <input type="reset"/>
12          </td>
13        </tr>
14      </table>
15      <input type="hidden" name="woche" value="1"/>
16      <input type="submit" value="abschicken"/>
17  </form>
```

- `input` vom Typ `text` erlaubt die Eingabe eines kurzen Textes
- `input` vom Typ `hidden` übergibt zusätzliche Werte, ohne dass diese dem Benutzer angezeigt werden
- `input` vom Typ `reset` setzt ein Formular zurück
- `input` vom Typ `submit` schickt das Formular ab
- `textarea` erlaubt die Eingabe längerer, mehrzeiliger Texte

Jede Eingabe, die durch eine Java-Web-Anwendung ausgelesen werden soll, benötigt einen Namen. Im HTML-Element `form` wird dann das aufzurufende Skript und die Methode (POST oder GET) angegeben.

20.6 Auslesen von POST/GET-Parametern in Java und JSP

In Java-Servlets (und damit auch in JSP-Seiten) sind einige Objekte implizit vordefiniert, die Zugriff auf relevante Daten liefern. Die genaue Beschreibung der folgenden Objekte findet Ihr in der JavaEE-API-Dokumentation.

20.6.1 `request`

Im `request`-Objekt der Klasse `HttpServletRequest` gibt es ein paar interessante Methoden für Zugriffe auf Daten, die in direkter Verbindung mit dem aktuellen Request stehen – das ist im Grunde alles, was der Webbrowser beim letzten Klick auf einen Link o.ä. übertragen hat. Insbesondere die Parameter können mit der Methode `getParameter(String name)` ausgelesen werden:

Listing 12: parameter.jsp

```
1  <%
2    String country = request.getParameter("country");
3    if (country == null) {
4      out.println("");
5    }
6    else {
7      out.println("Country: "+country+"<br/>");
8    }
9  %>
```

Weiterhin gibt es Methoden, um auf Bestandteile der Query zuzugreifen, Cookies auszulesen, etc.

20.6.2 `session`

HTTP ist ein Zustandsloses Protokoll: ein Client sendet eine Anfrage, der Server sendet eine Antwort und damit endet der gesamte Kontext. Die nächste Anfrage hat für den Server auf HTTP-Ebene keine Verbindung zur vorhergehenden. Wenn man das möchte, z.B. weil ein Benutzer sich nur ein mal einloggen soll und bei jedem nachfolgenden Seitenaufruf dennoch richtig erkannt werden soll (incl. Warenkorb, etc.), muss man mit Sessions arbeiten. Das Session-Handling funktioniert mit Java-Servlets automatisch. Zur Unterstützung gibt es das `session`-Objekt vom Typ `HttpSession`. Prominenteste Methode dieser Klasse ist `getId()`, die die ID der Session als String liefert. Die Session-ID identifiziert eine laufende Session eindeutig.

A Tipps zum Debugging

Manchmal funktionieren Programme nicht direkt beim ersten Anlauf und man muss debuggen. Beim Debuggen von Web-Applikationen gibt es ein paar unangenehme Effekte, von denen einige durch Caching verursacht werden. Sagen wir, Euer Code funktioniert nicht und Ihr habt Debug-Ausgaben eingefügt. Aber aus irgendeinem Grund werden auch die Debug-Ausgaben nicht im Browser ausgegeben. Grund dafür kann Caching sein – entweder auf Seiten des Servers oder auf Browser-Seite. Um auf Nummer Sicher zu gehen, solltet Ihr den Browser-Cache löschen und Euren Server (Apache oder Tomcat) beenden und neu starten.

Wenn Ihr sicher gehen wollt, dass Ihr mit einer bestimmten Version Eurer Webapp arbeitet (z.B. mit der, in der Ihr eine Änderung zum Testen durchgeführt habt) und nicht mit gecachten Seiten, könnt Ihr an passender Stelle (z.B. in einem h1-Tag) temporär einen Zähler einfügen, den Ihr bei Änderungen hochzählt. Wenn der geänderte Code vom Server übernommen und im Browser angezeigt wird, seht Ihr dann am Zähler, dass die Änderung wirksam geworden ist. Beispiel: Im ersten Versuch sieht Euer Code so aus:

```
<h1>Hauptstadt</h1>
<%= getCapital("Germany") %>
```

Nach Änderung des Ländernamens z.B. auf „USA“ kommt immer noch „Berlin“ als Ausgabe zurück. Um festzustellen, dass der Code richtg übersetzt und übernommen wurde, ändert Ihr die Überschrift:

```
<h1>Hauptstadt 1</h1>
<%= getCapital("Germany") %>
```

Nun sollte im Browser die Überschrift „Hauptstadt 1“ lauten. Wenn nicht, habt Ihr ein Problem mit einem Cache.

B Übersicht über die benutzten Befehle

?	Hilfe
? *command*	Hilfe zu einem Befehl
start dbm	Datenbankmanager starten
stop dbm	Datenbankmanager stoppen
get dbm configuration	Konfiguration des Datenbankmanagers anzeigen
catalog tcpip node *name*	entfernten Datenbankserver als lokalen Knoten *name* katalogisieren
list node directory	Verzeichnis aller katalogisierten Knoten anzeigen
uncatalog node *name*	katalogisierten Knoten *name* aus dem Verzeichnis entfernen
catalog database *name* as *alias*	Datenbank *name* unter lokalem Alias *alias* katalogisieren
list database directory	Verzeichnis aller katalogisierter Datenbanken anzeigen
uncatalog database *alias*	als *alias* katalogisierte Datenbank aus dem Verzeichnis entfernen
create databse *name*	neue Datenbank *name* erstellen
drop database *name*	Datenbank *name* löschen
connect to *name*	Verbinden mit Datenbank *name*
terminate	Verbindung mit Datenbank beenden
set schema *name*	das aktuelle Schema auf *name* setzen
get authorizations	die Privilegien auf der aktuellen Datenbank anzeigen
grant *privilege* on database to public/group *name*/user *name*	allen, einer Gruppe oder einem Benutzer auf der aktuellen Datenbank ein Privileg verleihen
revoke *privilege* on database to public/group *name*/user *name*	ein Privileg auf der aktuellen Datenbank wieder zurücknehmen
create schema *name*	erstelle ein neues Schema (einen Namensraum für Datenbankobjekte)
drop schema *name* restrict	lösche ein existierendes Schema
create table *name*	erstelle eine neue Tabelle
drop table *name*	lösche eine existierende Tabelle
alter table *name*	ändere die Definiation einer existierenden Tabelle
create sequence *name* as integer	erstelle eine Sequenz mit Auto-Inkrement
alter sequence *name* restart	setzt eine Sequenz wieder auf den Startwert zurück
create index *name* on *table* ...	erzeuge einen neuen Index auf einer Tabelle
drop index *name* ...	lösche den Trigger *name*
describe indexes for table *name*	zeige die Indexe auf der Tabelle *name* an
list tables for schema *name*	alle Tabellen für das Schema *name* anzeigen
describe table *name*	den Aufbau und die Attribute der Tabelle Name anzeigen

export to *file* of *type*	exportiere Daten in eine Datei
import from *file* of *type*	importiere Daten aus einer Datei
load from *file* of *type*	importiere Daten mit Festlegung der Importmethode
delete from *name* where *condition*	lösche Tupel aus einer Tabelle auf die *Condition* zutrifft
update *name* set *assignment* where *condition*	ändere den Inhalt von Tupeln auf die *Condition* zutrifft
insert into *name* values	füge direkt neue Werte in eine existierende Tabelle ein
insert into *name statement*	füge das Ergebnis eines *Statements* als neue Werte in eine existierende Tabelle ein