TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

EER UND SQL TUTORIAL SE1

Version: 1.0

Autor: Christoph Kozarits

Zuletzt geändert von: Christoph Kozarits

Zuletzt geändert am: 12.10.2003

INHALTSVERZEICHNIS

1. Grundlagen

1.1. Einleitung

Diese Tutorial soll die Grundlagen der Datenmodellierung und der Datenbankabfragesprache SQL vermitteln. Warum sollen Datenbanken überhaupt modelliert werden? Für ein kommerzielles Unternehmen sind die Betriebsdaten, insbesondere deren Korrektheit und Aktualität, von zentraler Bedeutung. Diese Daten sind die Grundlage für alle Aktivitäten und Entscheidungen. Sie reichen von alltäglichen Routinefällen (z.B. Beantwortung einer Kundenanfrage über die Verfügbarkeit eines Artikels) bis zu langfristigen Investitionen. Falsche, unaktuelle oder nicht abrufbare Daten sind der Albtraum des EDV-Managers. Die methodische Modellierung der Datenstrukturen in einem administrativen System legt den Grundstein für nachvollziehbare und übersichtliche Strukturen in der Datenbank.

Die folgenden Kapiteln sollen zeigen, wie grundlegende Strukturierungsformen für die Datenmodellierung im Relationenmodell dargestellt werden können.

Folgende Hilfsmittel werden im Tutorial verwendet:

Diese Box zeigt häufig gemachte Fehler an, und macht auf diverse Unregelmäßigkeiten aufmerksam.

Diese Box gibt Reading Tipps oder Web Tipps, wo selbst nachgeschlagen werden kann oder soll.

Diese Box wird dazu verwendet, Beispiele anzugeben, auch weiterführende Übungen sind darin untergebracht.

2. Das Relationenmodell

2.1. Einleitung

Das Ziel bei der Entwicklung von *konzeptionellen (semantischen) Datenmodellen* war, grundlegende Strukturierungsformen für die Datenmodellierung zur Verfügung zu stellen, die weitgehend den natürlichen, in unserer Sprache auftretenden Begriffsstrukturen entsprechen.
Die semantischen Strukturen, die in der Anwendungsumgebung auftreten, sollen explizit im Datenbankschema erfasst und dargestellt werden. Neben der Struktur der Daten sollte auch deren Semantik im Datenbankschema ausgedruckt werden können.

In diesem Tutorial wird nur auf das Extended-Entity-Relationship-Modell (EER) von Teorey, Yang und Fry eingegangen, das 1986 die Erweiterung des ER (Entity-Relationship-Modell, 1976) von Chen bildete.

Hinweis: Alle im Tutorial verwendeten Beispiele beziehen sich auf eine Tennis-Turnier-Verwaltung. Das gesamte EER dieser Verwaltung wird am Ende dieses Kapitels gezeigt.

2.2. Wichtige Begriffe

Entität:
Eine Entität ist eine persistente (d.h. dauerhaft gespeicherte) Klasse, die Objekte enthält, die mit Objekten anderer Entitäten in Beziehung stehen können. *Stammdaten* sind langlebige Entitäten, die die statische Grundstruktur der Daten einer Anwendung abbilden, *Journaldaten* bilden die laufenden Transaktionen ab, die Stammdaten betreffen (etwa Verkauf). Eine Entität wird üblicherweise mit einem Substantiv (Hauptwort) in der Einzahl benannt (z.B. Kunde).

Attribut:
Attribute sind Charakteristika von Entitäten, die diese beschreiben. Jede Entität kann mehrere Attribute besitzen, deren Werte jeweils ein Objekt der Entität beschreiben. Ein Attribut speichert ähnlich einer Variable Informationen innerhalb einer Entität. Jedes Attribut hat einen eindeutigen Typ und einen gültigen Wertebereich. Unterschieden wird zwischen identifizierenden und beschreibenden Attributen.

Relation:
Eine Relation ist die Beziehung zwischen zwei (oder mehreren) Entitäten. Welche Art von Beziehungen es gibt, wird später im Dokument beschrieben.

Schlüssel (identifizierende Attribute):
Attribute, die ausreichen jedes Objekt einer Entität eindeutig zu beschreiben, werden als Schlüssel bezeichnet (z.B. Matrikelnummer). Ein *zusammengesetzter Schlüssel* besteht aus mehreren Attributen. Ein *Primärschlüssel* ist ein möglichst kurzer (minimaler) Schlüssel, der in der Datenbank zur Identifizierung einer Entität verwendet wird. Ein *Fremdschlüssel* ist ein Attribut, das Schlüssel in einer benachbarten Entität ist und damit die beiden Entitäten miteinander in Relation setzt.

Integrität(sbedingung):
Integritäts- oder auch *Konsistenzbedingungen* sind logische und widerspruchsfreie Bedingungen, denen Attribute und Entitäten in einer konsistenten, d.h. logisch richtigen und sinnvollen, Datenbank genügen müssen. Eine konsistente Datenbank ist im allgemeinen die Grundlage für das Funktionieren jedes Programms, das mit den Daten der Datenbank arbeiten soll. Ein *Konsistenzprüfer* ist ein Programm, das die Konsistenzbedingungen einer gegebenen Datenbank überprüft und etwaige Inkonsistenzen geeignet meldet.

Relationales Datenbanksystem:
Ist eine Datenbankstruktur, zwischen deren Tabellen Beziehungen (Relationen) bestehen, die durch ein EER-Diagramm veranschaulicht werden können. Der Großteil der heute kommerziell verwendeten Datenbanken sind relationale Datenbanken. Ihre Vorteile liegen in der höheren Modularität und der besseren Performance gegenüber klassischen (hierarchischen) Datenbanken.

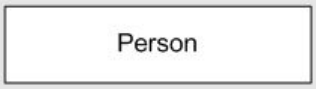
Index:
Ist eine Hilfstabelle, die der Datenbank das schnellere Auffinden eines bestimmten Datensatzes ermöglicht. Der schnellere Suchzugriff wird mit erhöhtem Platzbedarf bezahlt; beim Einfügen von Daten benötigt das Aktualisieren der Indextabellen extra Zeit.

2.3.Grundlegende EER-Konstrukte

2.3.1.Entitäten

Ein Objekt kann ein real existierendes Objekt sein, z.B. ein Gegenstand oder eine Person, aber auch ein ideelles Objekt, z.B. eine Dienstleistung wie eine Bestellung. Objekte mit denselben Eigenschaften werden zu *Entities* (Objekttypen) zusammengefasst. In der Praxis ist der deutsche Begriff Objekttyp kaum gebräuchlich.

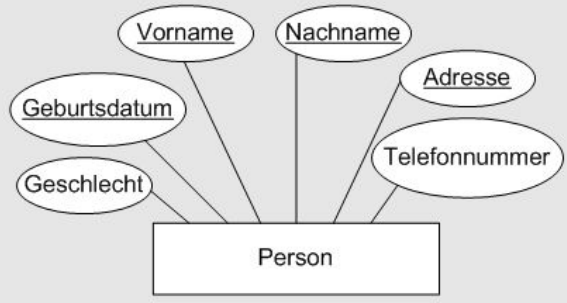
Entities werden im EER durch ein Rechteck dargestellt, wobei die Bezeichnung innerhalb des Rechtecks steht.



Die Entität mit der Bezeichnung Person

2.3.2.Attribute

Attribute haben einen beschreibenden Charakter für Entitäten. Eine bestimmte Ausprägung eines Attributes wird als Wert des Attributs bezeichnet. IM EER-Diagramm werden Attribute durch Ellipsen dargestellt, die mit der Entity, die sie charakterisieren, verbunden sind. Der Name des Attributs befindet sich innerhalb der Ellipse. Im Folgenden werden die Attribute aus Platz- und Übersicht-Gründen jeweils unter die Entität geschrieben.



Die Entität Person hat die Attribute Vorname, Nachname, Adresse, Geburtsdatum, Telefonnummer und Geschlecht. Die unterstrichenen Attribute bilden den Schlüssel. Dabei wird vorausgesetzt, dass es keine Personen gibt, die gleich heißen, das gleiche Geburtsdatum haben und auch noch die gleiche Wohnadresse.
Die zweite Darstellungsweise sieht folgendermaßen aus:

Unterschieden werden zwei Arten von Attributen: identifizierende und beschreibende Attribute. Erstere werden auch als Schlüssel bezeichnet und sind jene Attribute, die einzelne Objekte einer Entity eindeutig kennzeichnen. Beschreibende Attribute oder Nicht-Schlüsselattribute stellen nicht eindeutige Charakteristika der Objekte dar und dienen zur weiteren Beschreibung. Schlüssel können aus einem oder aus mehreren Attributen bestehen. Jede Entity kann mehrere verschiedene Schlüssel haben, im EER gibt es aber immer einen ausgezeichneten Schlüssel (Primärschlüssel), der durch Unterstreichung des Namens dargestellt wird.

Hat eine Entität mehrere Attribute, die einen Schlüssel bilden können, kann nur eines dieser Attribute als Primärschlüssel herangezogen werden.

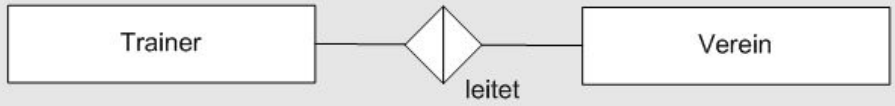
Eine Entität STUDENT hat die Attribute Sozialversicherungsnummer und Matrikelnummer. Beide Attribute bilden einen eindeutigen Schlüssel, jedoch darf nur eines der beiden als Primärschlüssel herangezogen werden.

2.3.3.Relationen

Eine Relation ist die Beziehung zwischen zwei (oder mehreren) Entitäten.
Der Grad einer Beziehung ist die Anzahl der Entities, die in dieser Beziehung miteinander verbunden sind. Eine Beziehung heißt n-äre Beziehung, wenn sie vom Grad n ist. Unäre, binäre und ternäre Beziehungen sind Spezialfälle, wobei unäre und binäre Beziehungen die in der realen Welt am häufigsten vorkommenden sind. Unäre Beziehungen sind Beziehungen einer Entity mit sich selbst und werden auch binär rekursive Beziehungen genannt.
Ternäre Beziehungen, also Beziehungen zwischen drei Entities werden benötigt, wenn binäre Beziehungen den Sachverhalt nicht ausreichend beschreiben können. Kann jedoch eine ternäre Beziehung mit Hilfe von zwei oder drei binären Beziehungen ausgedruckt werden, so ist der Einfachheit und Klarheit wegen die Darstellung durch binäre Beziehungen vorzuziehen.

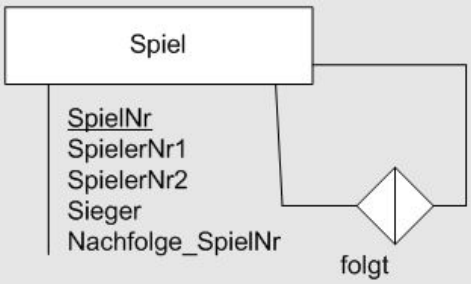
Graphisch werden Beziehungen in Form einer Raute dargestellt, deren Ecken mit den beteiligten Entities verbunden sind. Die Rolle wird über die Raute geschrieben.

Binäre Beziehung:



Ein Trainer leitet einen Verein.

Unäre Beziehung:

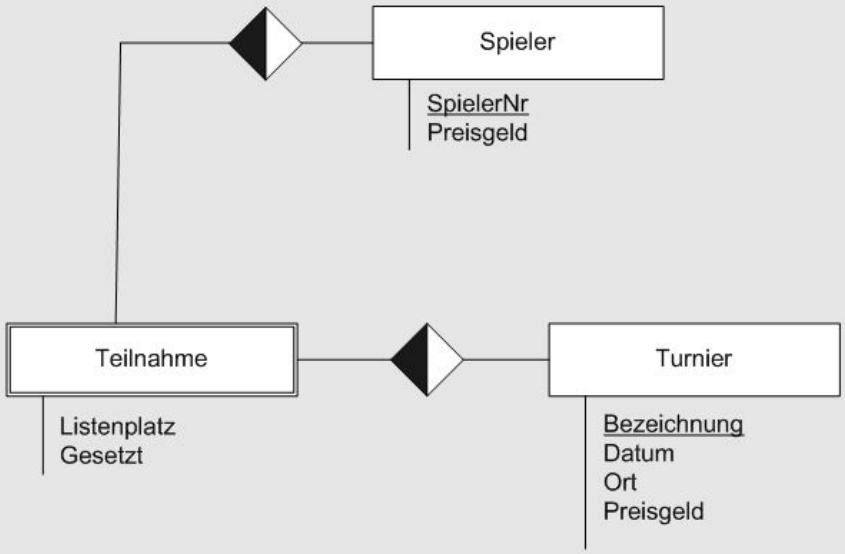


Die Entität Spiel hat eine Beziehung mit sich selber. Jedes Spiel in einem Turnier kann ein Nachfolge-Spiel haben (die Ausnahme bildet eigentlich nur das Finale). Jedes Spiel wird eindeutig durch die SpielNr identifiziert. Sollte es ein Nachfolge-Spiel geben, wird die SpielNr des nachfolgenden Spiels im Attribut Nachfolge_SpielNr gespeichert.

Binäre Beziehungen treten am häufigsten auf. Sollen Baum-Strukturen in einer Datenbank abgelegt werden, müssen unäre Beziehungen verwendet werden.

2.3.4.Weak Entity

Entities, deren Objekte nur mit Hilfe einer anderen Entity oder mehreren anderen Entities, mit der bzw. denen sie in Beziehung stehen, identifiziert werden können, werden Weak Entities genannt. Eine Weak Entity wird durch doppelte Umrahmung dargestellt. Sie hat keine Attribute, die alleine den Schlüssel bilden, obwohl sie dazu beisteuern können.



Die Entität Teilnahme bildet ein Weak-Entity, da sie nur durch Spieler und Turnier identifiziert werden kann. Ein Objekt der Entität Teilnahme ist nur sinnvoll, wenn es ein Objekt Spieler und Turnier gibt.

2.4.Komplexität einer Beziehung

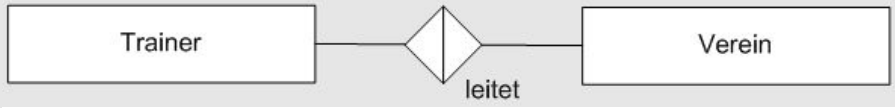
Die Komplexität einer Beziehung zwischen Entities beschreibt, mit wievielen Instanzen der jeweils anderen Entity jede Instanz einer Entity in Beziehung stehen kann. Bei einer Beziehung zwischen zwei Entities gibt es die folgenden Möglichkeiten:

1:1 – Beziehung
1:n – Beziehung
m:n - Beziehung

2.4.1.1:1 – Beziehung

Jedes Objekt der Entität A steht in Beziehung zu genau einem Objekt der Entität B.

Dargestellt wird diese Beziehung durch eine leere Raute.



Ein Trainer leitet genau einen Verein und jeder Verein wird maximal von einem Trainer geleitet.

2.4.2.1:n – Beziehung

Jedes Objekt der Entität A steht in Beziehung zu einem oder mehreren von B, aber jedes Objekt von B nur zu einem von A.

Die 1:n – Relation wird im EER-Diagramm durch eine halbgefüllte Raute dargestellt, wobei die Füllung auf der Seite der n Objekte steht.

Ein Trainer kann mehrere Spieler betreuen. Jeder Spieler kann aber nur von einem Trainer betreut werden.

2.4.3.m:n – Beziehung

Jedes Objekt der Entität A steht in Beziehung zu einem oder mehreren von B und jedes Objekt der Entität B steht in Beziehung zu einem oder mehreren von A.

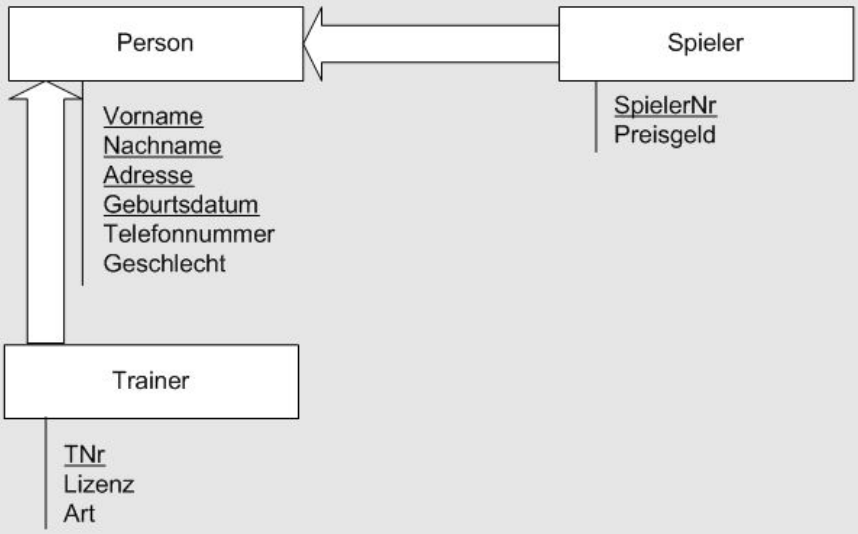
Diese Relation wird durch eine voll gefüllte Raute dargestellt.

Ein Tennis-Turnier kann über mehrere Plätze verfügen und ein Tennisplatz kann bei mehreren Turnieren verwendet werden.

2.5.Generalisierung

Das Konzept der Verallgemeinerung konnte mit den Mitteln des im letzten Abschnitt besprochenen Realtionen nur sehr dürftig dargestellt werden, nämlich als eine spezielle 1:1 - Beziehung zwischen zwei Entities.
Haben zwei oder mehrere Entities sehr viele gleiche Attribute, so können die gemeinsamen Attribute zu einem neuen Entity zusammengefasst werden und die bestehenden Entities davon abgeleitet werden (Vererbung bzw. Spezialisierung).

Im EER-Diagramm wird die Generalisierung durch einen Pfeil mit geschlossener Spitze dargestellt, wobei der Pfeil ausgehend von der Kind-Entitäten in Richtung der Eltern-Entität zeigt.

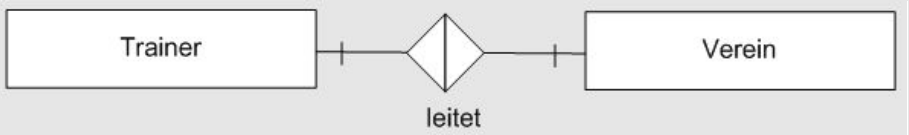


Die Entität Person umfasst alle Attribute, die sowohl Spieler, als auch Trainer benötigen. Diese Attribute werden vom Eltern-Entity Person an die Kinder-Entities Spieler und Trainer vererbt.

2.6.Existenz einer Entity in einer Beziehung

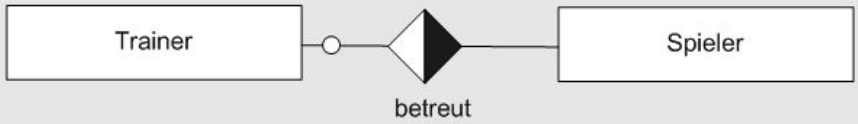
Wie in den letzten Kapiteln beschrieben, gibt die Komplexität einer Beziehung an, mit wievielen Instanzen der anderen Entität jede Instanz der einen Entität in Beziehung stehen kann. Dabei wurde bis jetzt immer von "einer" oder von "vielen" Instanzen gesprochen. Bis jetzt wurde aber keine Möglichkeit gezeigt, bei welcher die Entität mit keiner Instanz der anderen Entität in Beziehung steht. Das Konzept der Existenz einer Entity in einer Beziehung gibt uns die Möglichkeit auch diesen Sachverhalt zu modellieren. Die Existenz einer Entity in einer Beziehung heißt obligatorisch, wenn es zu jeder Instanz dieser Entity mindestens eine Instanz der mit ihr verbundenen Entity gibt, die mit ihr in Beziehung steht. Ist eine solche Instanz nicht unbedingt nötig, so heißt sie optional. Graphisch wird eine optionale Existenz mit einem Kreis auf der Verbindung zur Entity, obligatorische mit einem senkrechten Strich dargestellt.

Zwingende (obligatorische) Relation:



Die obligatorische Beziehung wird mit senkrechten Strichen dargestellt. Jeder Trainer muss mindestens einen Verein leiten (in diesem Fall durch die leere Raute genau einen Verein). Umgekehrt muss jeder Verein von einem Trainer geleitet werden.

Optionale Relation:



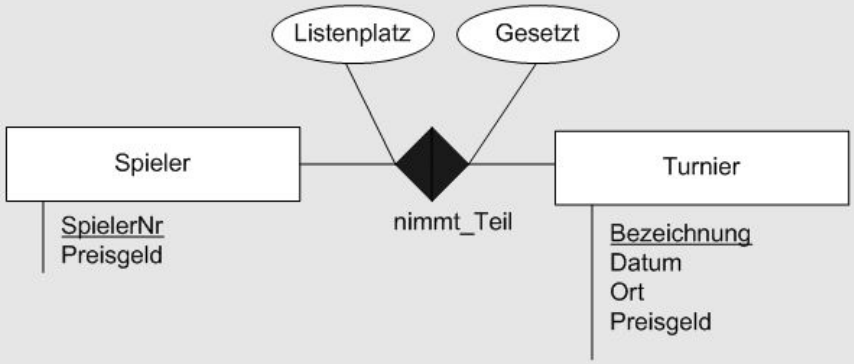
Die optionale Beziehung wird mit einem Kreis dargestellt.
Jeder Spieler kann (optional) von einem Trainer betreut werden. Das heißt, dass nicht jeder Spieler einen Trainer haben muss.

Unbekannt:

In der Beziehung zwischen den Entities Tennisplatz und Turnier wird die Existenz von Instanzen offengelassen.

2.7.Attribute einer Beziehung

Attribute können, wie in den vorigen Kapiteln gezeigt, zu Entities aggregiert werden. Es gibt aber auch die Möglichkeit, Attribute zu Beziehungen zu aggregieren. Attribute werden üblicherweise jedoch nur zu n:m-Beziehungen hinzugefügt, da im Falle von 1:1 - oder 1:n - Beziehungen mindestens auf einer der beiden Seiten der Beziehung ein einziges Objekt steht und damit die Mehrdeutigkeit kein Problem darstellt und dadurch die Attribute zu einem der beiden Entities aggregiert werden können.

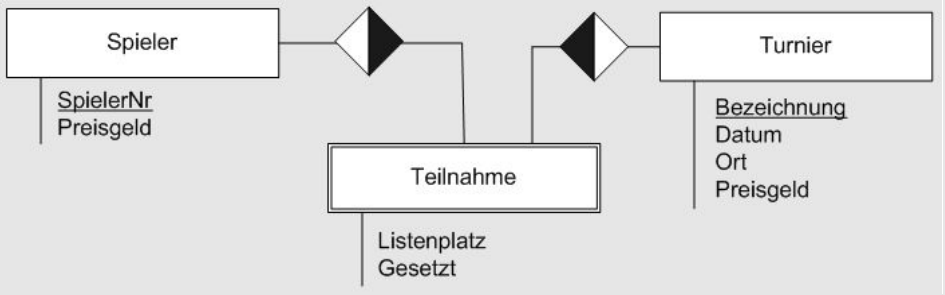


In der Relation zwischen Spieler und Turnier wird festgehalten, ob ein Spieler bei einem bestimmten Turnier gesetzt ist und welchen Listenplatz er bei jedem Turnier hat. Beide Attribute können weder der Entität Spieler noch der Entität Turnier zugeordnet werden, da die Informationen bei jedem Spieler bei jedem Turnier abgelegt werden sollen.

2.7.1.Vergleich zu Weak Entity

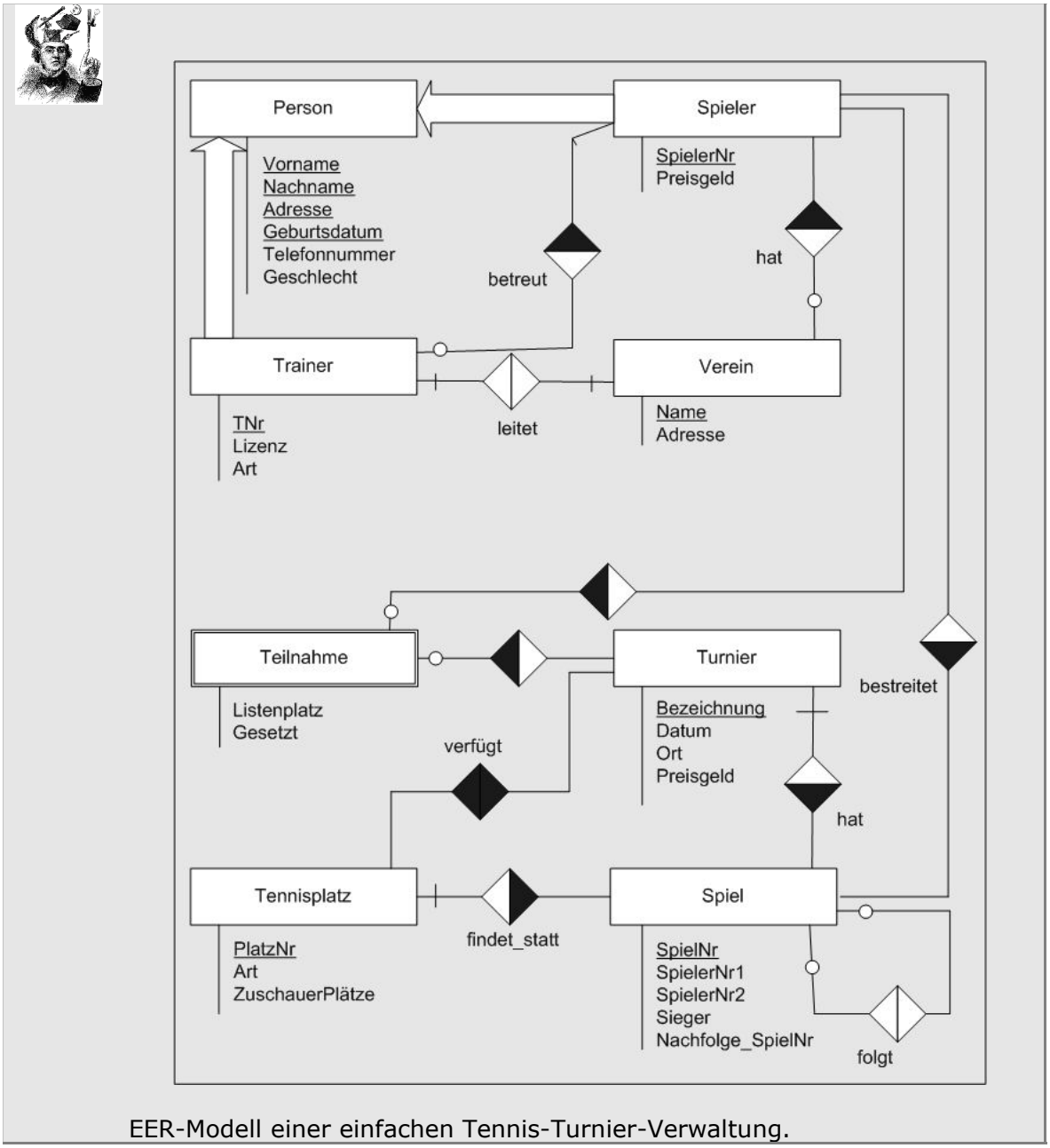
Wenn Attribute zu einer m:n - Beziehungen aggregiert werden, muss überlegt werden, ob nicht an Stelle der Beziehung eine Weak Entity modelliert werden sollte, zu der die Attribute aggregiert werden.

Jede m:n - Beziehung kann auch durch eine Weak Entity, die durch 1:n-Beziehungen mit den beiden sie definierenden Entities verbunden ist, aggregiert werden.



Die Relation zwischen Spieler und Turnier wird durch eine Weak-Entity Teilnahme ersetzt, in welcher die Attribute Listenplatz und Gesetzt festgehalten werden.

2.8.Gesamt-EER



EER-Modell einer einfachen Tennis-Turnier-Verwaltung.

3. Die Datenbanksprache SQL

3.1. Allgemeines

Die relationale Abfragesprache SQL bedeutet Structured Query Language und wurde 1974 von IBM entwickelt und 1986 von ANSI (American National Standards Institute) und 1987 von ISO (International Standard Organisation) als internationaler Standard einer Sprachschnittstelle für relationale Datenbankmanagementsysteme genormt. Nach einer ersten Überarbeitung im Jahr 1989, wo die Sprache erweitert wurde, um referenzielle Integrität und generelle Integritätsbedingungen zu verarbeiten, wurden 1992 in der zweiten Überarbeitung die Datenmanipulation, Schemamanipulation und Datendefinitionssprache erweitert. SQL-Anweisungen werden als Zeichenkette an die Datenbank geschickt und dort verarbeitet. In folgenden Kapiteln werden nur Datenbankabfragen und einfache Datenbankmanipulationen mittels behandelt.

Folgende Syntax wird in den folgenden Abschnitten verwendet.

Schriftart / Symbol	Bedeutung
GROSSBUCHSTABEN	Reservierte Worte und Zeichen.
Kursiv	Variablen, Namen, Typen, Ausdrücke, die eingesetzt werden müssen.
Klammerung durch {}	Wiederholungen; der zwischen den geschwungenen Klammern liegende Ausdruck kann beliebig oft vorkommen.
Klammerung durch []	Option; der zwischen den eckigen Klammern liegende Ausdruck kann höchstens einmal angegeben werden.
<u>unterstrichen</u>	Standardeinstellungen (Default).
Erstes | Zweites	Ein senkrechter Strich gibt eine Entweder-oder-Möglichkeit an, d.h. entweder Erstes oder Zweites.

Tabelle 3.1 Syntaxbeschreibung

3.2. Datentypen

SQL unterstützt unter anderem die folgenden Datentypen:

Datentyp	Wertebereich	Beschreibung
CHAR(n)	n Buchstaben, max. 32767	ein String fixer Länge
VARCHAR(n)	n Buchstaben, max. 32767	ein String variabler Länge, max. n Stellen groß
INTEGER	-2,147,483,648 bis 2,147,483,648	Ganzzahl
SMALLINT	-32768 bis 32767	Ganzzahl
FLOAT	$3.4*10^{-38}$ bis $3.4*10^{38}$	Gleitkommazahl in einfacher Genauigkeit
DOUBLE	$1.7*10^{-308}$ bis $1.7*10^{308}$	Gleitkommazahl in doppelter Genauigkeit
DATE	1 Jan 100 bis 11 Jan 5941	Datumsfeld; beinhaltet auch Uhrzeit

Tabelle 3.2 Datentypen

3.3.Datendefinition

3.3.1.Erzeugen von Tabellen

SQL stellt folgende Anweisung zur Verfügung, um eine Tabelle anzulegen.

CREATE TABLE *tabellenname (attribut1 typ {Optionen} {, attributN typ {Optionen} } { , Zusatzoptionen});*

Als Attribut- und Tabellennamen sind Wörter – bestehend aus alphanumerischen Zeichen und Unterstrich "_" – erlaubt. Als Optionen (eigentlich *Einschränkungen*, engl. *constraints*) kann angegeben werden, ob das Feld auf alle Fälle mit Daten gefüllt werden muß (Option NOT NULL) oder ob es sich bei dem betreffendem Attribut um einen einfachen Primärschlüssel handelt (Option PRIMARY KEY).

In den Zusatzoptionen können zusammengesetzte Schlüssel angegeben werden:

PRIMARY KEY (attribut1,....,attributN).

Auch Referenzen auf andere Tabellen können realisiert werden:

FOREIGN KEY*(schlüssel1)* REFERENCES *tabellenname(schlüssel2)*

Dieser Ausdruck bewirkt, dass eine logische Verknüpfung hergestellt wird um sicherzustellen, dass "*schlüssel1*" in der gerade angelegten Tabelle sich auf den korrespondierenden Schlüssel in der Tabelle "*tabellenname*" bezieht. Die angegebenen Attribute "*schlüssel1*" und "*schlüssel2*" müssen allerdings auch als Schlüssel oder zumindest als Teilschlüssel in der Tabelle "tabellenname" angegeben sein, um sicherzustellen, dass die Referenzierung auch tatsächlich existiert. Allerdings können diese Referenzen nur dazu genützt werden, um sicherzustellen, dass diese elementaren Integritätsbedingungen nicht verletzt werden. Das heißt, dass die *references*-Anweisung nur sicherstellt, dass zu dem angegebenen "schlüssel1" in der Tabelle "tabellenname" auch tatsächlich ein "schlüssel2" mit dem selben Wert existiert.

Weiters können auch Integritätsbedingungen durch CHECK Ausdruck angegeben werden, wobei *Ausdruck* ein logischer Ausdruck, wie etwa *attribut1 < attribut2* oder *attribut1 >= attribut2* ist.

```
CREATE TABLE Person
(Vorname VARCHAR(20) NOT NULL,
Nachname VARCHAR(30) NOT NULL,
Adresse VARCHAR(60) NOT NULL,
Geburtsdatum DATE NOT NULL,
Telefonnummer VARCHAR(30),
Geschlecht INTEGER NOT NULL,
PRIMARY KEY (Vorname, Nachname, Adresse, Geburtsdatum),
CHECK ((Geschlecht>=1) AND (Geschlecht<=2)));

CREATE TABLE Trainer
(TNr INTEGER PRIMARY KEY NOT NULL,
Person_Vorname VARCHAR(20) NOT NULL,
Person_Nachname VARCHAR(30) NOT NULL,
Person_Adresse VARCHAR(60) NOT NULL,
Person_Geburtsdatum DATE NOT NULL,
Lizenz INTEGER,
Art INTEGER NOT NULL,
FOREIGN KEY (Person_Vorname, Person_Nachname, Person_Adresse,
Person_Geburtsdatum) REFERENCES Person(Vorname, Nachname,
Adresse, Geburtsdatum));

CREATE TABLE Verein
(Name VARCHAR(40) PRIMARY KEY NOT NULL,
Trainer_TNr INTEGER NOT NULL,
Adresse VARCHAR(60),
FOREIGN KEY (Trainer_TNr) REFERENCES Trainer(TNr));

CREATE TABLE Spieler
(SpielerNr INTEGER PRIMARY KEY NOT NULL,
Person_Vorname VARCHAR(20) NOT NULL,
Person_Nachname VARCHAR(30) NOT NULL,
Person_Adresse VARCHAR(60) NOT NULL,
Person_Geburtsdatum DATE NOT NULL,
Verein_Name VARCHAR(40),
Preisgeld FLOAT,
FOREIGN KEY (Person_Vorname, Person_Nachname, Person_Adresse,
Person_Geburtsdatum) REFERENCES Person(Vorname, Nachname,
Adresse, Geburtsdatum),
FOREIGN KEY (Verein_Name) REFERENCES Verein(Name));
```

Erläuterung zum Beispiel:

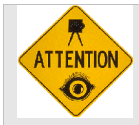
Hier wurde ein Teil der Tabellen erzeugt, die für das Relationenmodell des Tennisturniers aus dem letzten Kapitel notwendig sind (vgl. Kapitel 2.8 Gesamt-EER). Es wurden die Tabellen Person, Trainer, Verein und Spieler erzeugt, welchen den gleichnamigen Entitäten entsprechen. Die erste Tabelle Person hat den zusammengesetzten Primärschlüssel aus den Attributen Vorname, Nachname, Adresse und Geburtsdatum.

Primärschlüssel dürfen nie einen NULL-Wert enthalten.

Das Attribut Geschlecht darf nur die Werte 1 oder 2 annehmen (CHECK-Klausel).

Die Entität Trainer wurde von Person abgeleitet (Generalisierung), was in SQL durch Fremdschlüssel umgesetzt wird. Den Attributen, die den Fremdschlüssel bilden, wurde aus Gründen besserer Lesbarkeit immer der Tabellenname gefolgt von einem Unterstrich der referenzierten Tabelle vorangestellt.

Als nächstes wurde die Tabelle Verein erzeugt, welche mit Trainer durch eine 1:1-Beziehung in Verbindung steht. Dabei erhält die Tabelle Verein den Fremdschlüssel Trainer_TNr.

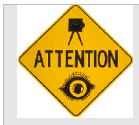


1:1-Relationen können immer auf 2 Arten umgesetzt werden.

Die zweite Möglichkeit im Beispiel wäre in der Tabelle Trainer als Fremdschlüssel das Attribut Name der Tabelle Verein anzulegen. Dabei müsste aber die Tabelle Verein zuerst erzeugt werden.

Die zwingende Beziehung wird durch das NOT NULL beim Fremdschlüssel garantiert.

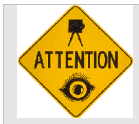
Die Entität Spieler erbt von der Entität Person und hat mit Verein eine 1:n-Beziehung und hat daher auch 2 Fremdschlüssel.



Die Reihenfolge des Erstellens der Tabellen in unserem Beispiel musste genau in dieser Reihenfolge erfolgen, da nur auf Tabellen referenziert werden kann, die es bereits gibt. Die Ausnahme wäre, wenn die 1:1-Relation anders umgesetzt werden würde.

3.3.2.Ändern von Tabellen

Manchmal kann es vorkommen, dass man nachträglich die Struktur einer Tabelle verändern will. SQL stellt dafür den "ALTER TABLE"-Befehl zur Verfügung.



ALTER TABLE *tabellenname*
ADD *attributname1 typ1* |
DROP *attributname1* |
CHANGE *attributname1_alt attributname1_neu typ1_neu* |
MODIFY *attributname1 typ1_neu*
{ , ADD *attributnameN typN* |
DROP *attributnameN* |
CHANGE *attributnameN_alt attributnameN_neu typN_neu* |
MODIFY *attributnameN typN_neu };*

```
ALTER TABLE Person
ADD Email VARCHAR(20),
MODIFY Telefonnummer VARCHAR(40) NOT NULL,
DROP Geschlecht;
```

Das Attribut Email wird der Tabelle Person hinzugefügt, das Attribut Geschlecht wird gelöscht und der Typ des Attributs Telefonnumer wird geändert.

3.3.3.Löschen von Tabellen

Die Anweisung DROP *tabellenname* wird verwendet, um eine Tabelle zu entfernen. Die angegebenen Tabellen mit allen darin enthaltenen Daten werden *vollständig* gelöscht.

DROP TABLE *tabellenname1 {, tabellennameN};*

3.4.Datenmanipulation

3.4.1.Einfügen von Daten

SQL stellt folgende Anweisung zur Verfügung, um Daten in eine Tabelle zu speichern.

INSERT INTO *tabellenname [(attribut1, attribut2, ... attributN*)] VALUES (*wert1, wert2, ... wertN*);

Mit diesem Befehl wird ein neuer Datensatz (ein neuer Tupel bzw. eine neue Zeile in der Tabelle) angelegt und die Attribute *attribut1, attribut2,... attributN* mit den Werten *wert1, wert2, ... wertN* gefüllt. Falls die Werte für alle Attribute bekannt sind und die Werte in der richtigen Reihenfolge angeführt werden, so kann die Angabe der Attribute weggelassen werden. Die richtige Reihenfolge bedeutet hier, daß die Werte für die einzelnen Attribute in genau der Reihenfolge angegeben werden, in der sie beim Anlegen der Datenbank angegeben wurden. Werden neue Attribute zu einer Tabelle hinzugefügt, so werden diese am Ende angefügt.

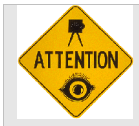
INSERT INTO Person (Vorname, Nachname, Adresse, Geburtsdatum, Telefonnummer, Geschlecht)
VALUES
(“Hans“,“Meier“,“Waldweg 2“,“10.09.1970“,“01/7654321“,1);

entspricht

INSERT INTO Person
VALUES
(“Hans“,“Meier“,“Waldweg 2“,“10.09.1970“,“01/7654321“,1);

3.4.2.Ändern von Daten

Um bestehende Datensätze zu modifizieren verwendet man folgenden Befehl:



UPDATE *tabellenname*
SET *attributname = value {, attributname = value*}
[WHERE *Bedingung*];

Jedem Attribut wird der Wert zugewiesen, der hinter dem Gleichheitszeichen steht. Es werden allerdings alle Tupel bearbeitet falls keine WHERE-Klausel angegeben wird, welche die Änderung auf bestimmte Datensätze beschränkt.

UPDATE Person SET
Vorname = "Hans", Nachname = "Huber"
WHERE
Adresse = "Waldweg 2";

ACHTUNG: Jede Person, die als Adresse "Waldweg 2" hat, wird in "Hans Huber" umbenannt.

3.4.3.Löschen von Daten

Zum Löschen von Datensätzen aus einer Tabelle stellt SQL folgenden Befehl zur Verfügung.

DELETE FROM *tabellenname*
[WHERE *Bedingung*];

Wie beim UPDATE-Befehl kann auch beim Löschen durch die WHERE-Klausel auf bestimmte Datensätze beschränkt werden.

DELETE FROM Person
WHERE
Vorname = "Hans" AND Nachname = "Huber";

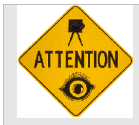
Jede Person, die "Hans Huber" heißt, wird gelöscht.

3.5.Datenabfragen

Die wohl wichtigste Fähigkeit einer Datenbanksprache ist ihre Möglichkeit Daten abzufragen. SQL stellt den SELECT-Befehl zur Verfügung, mit dem *Anfragen* (*Queries*) an die Datenbank geschickt werden können. Die Datenbank antwortet auf diese Anfragen in Form einer Tupel-Menge, also einer Liste von Tupeln (Datensätzen), die den angegebenen Kriterien entsprechen.

3.5.1.Grundkonstruktion einer Abfrage

Allgemein hat eine Abfrage folgendes Aussehen. In den weiteren Abschnitten werden die genauen Abfragemöglichkeiten näher beleuchtet.



SELECT *attributname {, attributname}*
FROM *tabellenname { , tabellenname }*
[WHERE *Bedingung*]
[ORDER BY [ASC | DESC] *attributname*];

Eine solche SQL-Abfrage wird ausgewertet, indem zunächst das kartesische Produkt der in der FROM-Klausel angegebenen Relationen gebildet wird. Daraus werden dann jene Tupel ausgewählt, die die in der WHERE-Klausel angegeben Bedingungen erfüllen. Zuletzt werden die in der SELECT-Klausel angegebenen Attribute dieser Tupel ausgewählt. Sollen alle Attribute ausgewählt werden, wird der Stern (*) verwendet.

ACHTUNG: Für alle weiteren Beispiele wird angenommen, dass die Datenbank mit foglenden Daten befüllt wurde.

```
INSERT INTO Person VALUES
("Hans","Meier","Waldweg 2","10.09.1970","01/7654321",1);
INSERT INTO Person VALUES
("Hans","Meier","Wasserweg 27","31.12.1945","01/2222222",1);
INSERT INTO Person VALUES
("Peter","Schmidt","Hauptstrasse 45","11.02.1982","01/8765432",1);
INSERT INTO Person VALUES
("Christian","Lang","Wiesenweg 30","10.09.1965","01/9876543",1);
INSERT INTO Person VALUES
("Petra","Lang","Wiesenweg 30","08.08.1968","01/9876543",2);
INSERT INTO Person VALUES
("Martin","Schmidt","Kleinweg 15","25.04.1956","01/1234567",1);
INSERT INTO Person VALUES
("Martina","Kurz","Lärchengasse 12","01.11.1977","01/2345678",2);
```

SELECT * FROM Person;

Liefert alle Attribute aller Datensätze in der Tabelle Person.

Vorname	Nachname	Adresse	Geburtsdatum	Telefonnummer	Geschlecht
Hans	Meier	Waldweg 2	10.09.1970	01/7654321	1

Hans	Meier	Wasserweg 27	31.12.1945	01/2222222	1
Peter	Schmidt	Hauptstrasse 45	11.02.1982	01/8765432	1
Christian	Lang	Wiesenweg 30	10.09.1965	01/9876543	1
Petra	Lang	Wiesenweg 30	08.08.1986	01/9876543	2
Martin	Schmidt	Kleinweg 15	25.04.1956	01/1234567	1
Martina	Kurz	Lärchengasse 12	01.11.1977	01/2345678	2

Tabelle 3.5.1a Ergebnis des Beispiels

Mit der ORDER BY-Klausel können die Datensätze in einer bestimmten Reihenfolge sortiert werden (ASC: aufsteigend, DESC: absteigend).

SELECT * FROM Person ORDER BY Vorname;

Liefert alle Attribute aller Datensätze in der Tabelle Person sortiert nach dem Attribut Vorname.

Vorname	Nachname	Adresse	Geburtsdatum	Telefonnummer	Geschlecht
Christian	Lang	Wiesenweg 30	10.09.1965	01/9876543	1
Hans	Meier	Waldweg 2	10.09.1970	01/7654321	1
Hans	Meier	Wasserweg 27	31.12.1945	01/2222222	1
Martin	Schmidt	Kleinweg 15	25.04.1956	01/1234567	1
Martina	Kurz	Lärchengasse 12	01.11.1977	01/2345678	2
Peter	Schmidt	Hauptstrasse 45	11.02.1982	01/8765432	1
Petra	Lang	Wiesenweg 30	08.08.1986	01/9876543	2

Tabelle 3.5.1b Ergebnis des Beispiels

3.5.2.Die Selektion

Die Selektion wird durch die Angabe der Bedingungen in der WHERE-Klausel realisiert. Bei den Bedingungen können beliebige Vergleichsoperatoren (die für den Datentyp definiert sind) verwendet werden.

SELECT * FROM Person WHERE Geschlecht = 1 AND Geburtsdatum>"01.01.1960";

Liefert alle Personen, die männlich sind (Geschlecht = 1) und später als am 01.01.1960 geboren sind.

Vorname	Nachname	Adresse	Geburtsdatum	Telefonnummer	Geschlecht
Hans	Meier	Waldweg 2	10.09.1970	01/7654321	1
Peter	Schmidt	Hauptstrasse 45	11.02.1982	01/8765432	1
Christian	Lang	Wiesenweg 30	10.09.1965	01/9876543	1

Tabelle 3.5.2 Ergebnis des Beispiels

3.5.3.Die Projektion

Die Projektion wurde vorher schon kurz angeschnitten und erfolgt durch Angabe der gewünschten Attribute nach dem SELECT realisiert.

SELECT Vorname, Nachname, Geburtsdatum FROM Person;

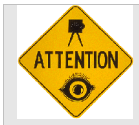
Liefert Vorname, Nachname und Geburtsdatum aller Personen.

Vorname	Nachname	Geburtsdatum
Hans	Meier	10.09.1970
Hans	Meier	31.12.1945
Peter	Schmidt	11.02.1982
Christian	Lang	10.09.1965
Petra	Lang	08.08.1986
Martin	Schmidt	25.04.1956
Martina	Kurz	01.11.1977

Tabelle 3.5.3 Ergebnis des Beispiels

3.5.4.Alias

Nach den grundlegenden Abfragemöglichkeiten in den letzten Kapiteln können nun schon etwas komplexere Abfragen durchgeführt werden.
Daher folgende erweiterte Syntax:



SELECT [ALL | DISTINCT] [Alias.]*attributname* {,[Alias.]*attributname*}
FROM *tabellenname* [Alias] { , *tabellenname* [Alias] }
[WHERE *Bedingung*]
[ORDER BY [ASC | DESC] *attributname*];

Der Unterschied zu unserer ersten Definition liegt nun darin, dass für jeden Tabellennamen ein *Alias* (eine Ersetzung) angeben werden kann. Weiters können wir entscheiden, ob alle doppelt vorkommenden Tupel in der Ergebnismenge auch doppelt angegeben werden (Option ALL) oder ob doppelte Einträge nur einmal ausgegeben werden (Option DISTINCT).

Es sollen alle SpielerNr und Preisgelder der Mitglieder aller Vereine ausgegeben werden, die bereits mehr als 1000 Euro verdient haben. (HINWEIS: Relationen siehe EER im Kapitel 2.8)

SELECT DISTINCT s.SpielerNr, s.Preisgelder, v.Name FROM Spieler s, Verein v WHERE s.Verein_Name = v.Name AND s.Preisgeld>1000;

Hier haben wir nun ein Beispiel für Alias: Der Tabelle "Spieler" wird der Alias "s" zugewiesen und "Verein" erhält den Alias "v". Dieses Aliassystem ist sehr hilfreich, wenn es darum geht, Attribute aus verschiedenen Tabellen zu vergleichen. Zu beachten ist, dass die einzelnen Attribute mittels *Alias.Attributname* angesprochen werden. Doch nun etwas genauer zu unserer Anfrage.
In der WHERE-Klausel vergleichen wir den Namen der Tabelle "Verein" mit dem entsprechenden Feld in der Tabelle "Spieler" (Verein_Name) auf Gleichheit. SQL liefert daraufhin alle in der SELECT-Klausel angegebenen Felder zurück, die zu Tupeln gehören, die die WHERE-Klausel erfüllen. Doppelt auftretende Tupel werden hier nur einmal ausgegeben. Zusätzlich werden nur jene Datensätze ausgegeben, die in der Tabelle Spieler im Attribut "Preisgeld" einen höheren Wert als 1000 enthalten.

3.5.5.Aggregatfunktionen

Um Summen, Maxima und Minima zu bilden können in SQL Aggregatfunktionen verwendet werden.

Aggregatfunktion	Beschreibung
COUNT (attributname)	Anzahl der Records in der Ergebnismenge.
SUM (attributname)	Der Wert des Attributs wird für alle Ergebnistupel summiert.
AVG (attributname)	Der Wert des Attributs wird für alle Ergebnistupel gemittelt.
MAX (attributname)	Das Maximum des angegebenen Attributs in der Ergebnismenge.
MIN (attributname)	Das Minimum des angegebenen Attributs in der Ergebnismenge.

Tabelle 3.5.5 Aggregatfunktionen

Es sollen die Anzahl der Mitglieder des Vereins "Tennis 2000" ausgegeben werden. (HINWEIS: Relationen siehe EER im Kapitel 2.8)

SELECT COUNT(*) FROM Spieler WHERE Verein_Name = "Tennis 2000";

Als Ergebnis dieser Anfrage erhalten wir eine Zahl, nämlich die gesuchte Anzahl von Mitgliedern im Verein "Tennis 2000" (und in der Datenbank erfasst sind).

3.5.6.Gruppierung

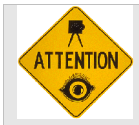
Im letzten Kapitel wurden die Aggregatfunktionen vorgestellt. Was passiert aber, wenn in einer Abfrage Aggregatfunktionen mit anderen Attributen vermischt werden sollen.

SELECT Spieler_SpielerNr, COUNT(*) FROM Teilnahme;

ACHTUNG: Dieses Beispiel liefert einen Fehler.

Versuchen wir jetzt, ohne uns eine konkrete Ergebnismenge anzusehen, zu bestimmen, warum obiges Beispiel nicht durchführbar ist. Als erstes geben wir alle SpielerNr der Spieler aus. Dann sollen wir auch noch die Anzahl der Spieler ausgeben. Wir stellen hier also fest, dass der erste Teil der SELECT-Klausel *mehrere* Datensätze zurückliefern würde, COUNT hingegen genau *eine* Zahl. SQL ist leider nicht in der Lage derartige Ergebnismengen zu generieren. Die obige Anfrage ist daher so *nicht* möglich.

Daher gibt es in SQL de Möglichkeit Gruppierungen zu bilden. Dazu folgende erweiterte Syntax.



SELECT [ALL | DISTINCT] [Alias.]*attributname* {,[Alias.]*attributname*}
FROM *tabellenname* [Alias] { , *tabellenname* [Alias] }
[WHERE *Bedingung*]
[GROUP BY [Alias.]*attributname* {, [Alias.] *attributname*}
[HAVING *Bedingung*]];
[ORDER BY [ASC | DESC] *attributname*];

Die GROUP BY-Klausel hat den Effekt, dass die Ergebnistupel entsprechend ihren Ausprägungen zusammengefasst werden. Mittels der optionalen HAVING-Klausel können die Bedingungen aus der WHERE-Klausel noch erweitert werden.
Mit anderen Worten: Zuerst werden jene Tupel in die Ergebnismenge aufgenommen, die der WHERE-Klausel genügen und dann werden jene Tupel wieder eliminiert, die der HAVING-Klausel nicht genügen. GROUP BY-Ausdrücke können jedes beliebige Attribut der Tabellen in der FROM-Klausel verwenden, egal ob sie in der SELECT-Klausel aufscheinen oder nicht. Fehlt die GROUP BY-Klausel, so wird das Ergebnis als eine einzige Gruppe aufgefasst.

Hier also eine korrekte Version der obigen Anfrage:

```
SELECT Spieler_SpielerNr, COUNT(Spieler_SpielerNr) FROM Teilnahme
GROUP BY Spieler_SpielerNr;
```

Wenn wir nun die Ergebnismenge betrachten, wird für jeden Spieler angegeben, wie oft er in der Tabelle vorkommt (bei wie vielen Turnieren er gestartet ist).
Es gibt allerdings auch noch Regeln für die Verwendung von GROUP BY-Klauseln: Bei Verwendung der GROUP BY-Klausel muss jeder Ausdruck in der SELECT-Klausel entweder eine Konstante, eine Aggregatfunktion (wie etwa SUM, COUNT) oder exakt ein Ausdruck der GROUP BY-Klausel sein.

3.5.7.Mengenoperationen

SQL stellt die Mengenoperationen UNION (Mengenvereinigung) für typkompatible Relationen (d.h. Relationen mit der gleichen Anzahl von. jeweils typkompatiblen Attributen) zur Verfügung.

SQL-Query [UNION SQL-Query];

Es sollen alle Adressen aller Personen und Vereine ausgegeben werden.

```
(SELECT Adresse FROM Person)
UNION
(SELECT Adresse FROM Verein);
```

3.5.8.Teilabfragen

Innerhalb der WHERE-Klausel kann an jeder Stelle, an der ein Wert erwartet wird, dieser auch durch eine SQL-Abfrage berechnet werden. Wird in der Teilabfrage (Sub-Select) auf keine Relation der Hauptabfrage Bezug genommen, so handelt es sich um eine einfache Teilabfrage. Diese wird nur einmal ausgewertet, das Ergebnis dieser Auswertung wird dann in der Hauptabfrage verwendet. Wird in der Teilabfrage auf eine Relation der Hauptabfrage Bezug genommen, so nennen wir das eine korrelierte Teilabfrage. In diesem Fall wird die Teilabfrage für jedes Tupel der Hauptabfrage neu ausgewertet.

Es sollen alle Spieler erfasst werden, die bereits mehr Preisgeld gewonnen haben, als der Spieler „Christian Lang".

```
SELECT * FROM Spieler WHERE Preisgeld >
(SELECT Preisgeld FROM Spieler WHERE Person_Vorname = "Christian"
AND Person_Nachname = "Lang");
```

Wichtige spezielle Formen von Teilbedingungen in der WHERE-Klausel sind:

Vergleichsoperator	Wird wahr, wenn…
EXISTS <Abfrage>	Ergebnismenge von <Abfrage> nicht leer ist.
<Ausdruck> <Operator> ALL <Abfrage>	<Ausdruck> <Operator> allen in <Abfrage> vorkommendem Ausdruck ist.
<Ausdruck> <Operator> ANY <Abfrage>	<Ausdruck> <Operator> einem in <Abfrage> vorkommendem Ausdruck ist.
<Ausdruck> IN <Abfrage>	<Ausdruck> in <Abfrage> vorkommt.
<Ausdruck> NOT IN <Abfrage>	<Ausdruck> nicht in <Abfrage> vorkommt.

Tabelle 3.5.8 Vergleichsoperatoren für Teilabfragen

Es soll überprüft werden, ob irgendein Spieler, der mehr als der Durschnitt aller Spieler an Preisgeld verdient hat, einen Trainer ohne Lizenz hat.

```
SELECT * FROM Trainer WHERE Lizenz = 0 AND TNr =
ANY (SELECT Trainer_TNr FROM Spieler WHERE Preisgeld >
(SELECT AVG(Preisgeld) FROM Spieler));
```

3.5.9.Der Verbund

Wenn benötigte Daten in getrennten Tabellen liegen können diese bei Bedarf über ihre Schlüssel in Zusammenhang gebracht werden. Dieser Vorgang wird als Verbund (Join) bezeichnet. Das geschieht über ein entsprechendes Statement in der WHERE-Klausel.

Es sollen alle Spieler mit allen Turnieren an denen sie teilgenommen haben, ausgegeben werden.

```
SELECT s.SpielerNr, t.Bezeichnung FROM Spieler s,Turnier t, Teilnahme tn
WHERE s.SpielerNr = tn.Spieler_SpielerNr AND t.Bezeichnung =
tn.Turnier_Bezeichnung;
```

SQL kennt aber auch eine elegantere Möglichkeit, die den Zweck der Verknüpfung hervorhebt:

Es sollen alle Spieler mit allen Turnieren an denen sie teilgenommen haben, ausgegeben werden.

```
SELECT s.SpielerNr, t.Bezeichnung FROM
(Spieler s JOIN Teilnahme tn ON s.SpielerNr = tn.Spieler_SpielerNr) JOIN
Turnier t ON tn.Turnier_Bezeichnung = t.Bezeichnung;
```

3.6.Benutzeransichten

SQL kennt auch eine Möglichkeit, um häufig benutzte SELECT-Anweisungen (z.B. "maßgeschneiderte" Ausschnitte aus großen Tabellen für bestimmte Benutzer, Routineabfragen oder Zwischenergebnisse für komplexe Abfragen) abzulegen. Dazu können Ansichten definiert werden, auf die in der Folge wie auf eine Tabelle zugegriffen werden kann (das Einfügen von Daten ist je nach Art der der Ansicht zugrundeliegenden Abfrage im allgemeinen aber nicht mehr möglich). Eine Ansicht (View) wird wie folgt angelegt:

CREATE VIEW *Name* AS *Select-Anweisung*;

Jede zulässige SELECT-Anweisung kann als View definiert werden.

4. Kurze Einführung in PostgreSQL

Hier eine kleine Sammlung von wichtigen und praktischen Befehlen für die Datenbank. Alle Befehle werden wenn nicht ausdrücklich anders beschrieben in der Konsole (Cygwin, Terminal) verwendet.

4.1. Anlegen einer Datenbank

Der Befehl lautet:

createdb [options] dbname [description]

Options:
-D, --location=PATH Alternative place to store the database
-T, --template=TEMPLATE Template database to copy
-E, --encoding=ENCODING Multibyte encoding for the database
-h, --host=HOSTNAME Database server host
-p, --port=PORT Database server port
-U, --username=USERNAME Username to connect as
-W, --password Prompt for password
-e, --echo Show the query being sent to the backend
-q, --quiet Don't write any messages

Um eine Datenbank auf einem Lokalrechner zu erstellen reicht einfach z.B.:

createdb sedatenbank

Wenn man eine Datenbank auf einem anderen Rechner erstellen will ist die Option '-h' bzw '--host=HOSTNAME' von Nöten. Weiters ist ein '-D' nötig um einen Pfadanzugeben wo die Datenbank ihre Dateien ablegen soll.

zB. *createdb -h 10.10.230.222 -D /home/datenbank/data/ sedatenbank*

4.2. Löschen der Datenbank

Der Befehl lautet:

dropdb [options] dbname

Options:
-h, --host=HOSTNAME Database server host
-p, --port=PORT Database server port
-U, --username=USERNAME Username to connect as
-W, --password Prompt for password
-i, --interactive Prompt before deleting anything
-e, --echo Show the query being sent to the backend
-q, --quiet Don't write any messages

Eine lokale Datenbank kann ganz einfach mit folgendem Befehl gelöscht werden:

dropdb sedatenbank

4.3.Anlegen eines Datenbankusers

Der Befehl laut:

createuser [options] [username]

Options:
-d, --createdb User can create new databases
-D, --no-createdb User cannot create databases
-a, --adduser User can add new users
-A, --no-adduser User cannot add new users
-i, --sysid=SYSID Select sysid for new user
-P, --pwprompt Assign a password to new user
-E, --encrypted Encrypt stored password
-N, --unencrypted Do no encrypt stored password
-h, --host=HOSTNAME Database server host
-p, --port=PORT Database server port
-U, --username=USERNAME Username to connect as (not the one to create)
-W, --password Prompt for password to connect
-e, --echo Show the query being sent to the backend
-q, --quiet Don't write any messages

Um einen einfachen Datenbankuser anzulegen reicht der Befehl z.B.:

createuser sestudent

Danach fragt das System ob der Datenbankuser Datenbanken oder andere User anlegen darf. Diese Fragen können verneint werden.

4.4.Dump

4.4.1.Erstellen eines Dumps

Das Erstellen eines Dumps ist auf die Datenbank bezogen, die häufigste Tätigkeit eines Entwicklers. Der Dump ermöglicht das vollkommene reproduzieren einer Datenbank. Der Dump ist ein Textfile das aus SQL-Statements besteht. Er generiert alle Tabellen, Trigger, Funktionen und füllt die Tabellen auch mit Daten.

Der Befehl lautet:

pg_dump [options] dbname

Options:
-a dump only the data, not the schema
-b include large objects in dump
-c clean (drop) schema prior to create
-C include commands to create database in dump
-d dump data as INSERT, rather than COPY, commands
-D dump data as INSERT commands with column names
-f FILENAME output file name
-F {c|t|p} output file format (custom, tar, plain text)
-h HOSTNAME database server host name
-i proceed even when server version mismatches pg_dump version
-n suppress most quotes around identifiers

-N enable most quotes around identifiers
-o include oids in dump
-O do not output \connect commands in plain text format
-p PORT database server port number
-R disable ALL reconnections to the database in plain text format
-s dump only the schema, no data
-S NAME specify the superuser user name to use in plain text format
-t TABLE dump this table only (* for all)
-U NAME connect as specified database user
-v verbose mode
-W force password prompt (should happen automatically)
-x do not dump privileges (grant/revoke)
-X use-set-session-authorization output SET SESSION AUTHORIZATION commands
rather than \connect commands
-Z {0-9} compression level for compressed formats

Die üblich verwendeten Argumente sind '-f' um ein Outfile anzugeben, '-U' um einen gültigen Datenbankuser anzugeben, wenn man mit einer Datenbank im Netz arbeitet '-h'. Anschließend muss natürlich die Datenbank angegeben werden.

Beispielbefehl für Dump der lokalen Datenbank

pg_dump -f dump.out -U sestudent sedatenbank

Beispiel für Dump von einem Rechner im Internet

pg_dump -f dump.out -U sestudent -h 192.123.34.45 sedatenbank
oder
pg_dump -f dump.out -U sestudent -h db.irgenwo.org sedatenbank

4.4.2.Einspielen eines Dumps

Das Einspielen benutzt den Befehl psql. Die einzige Vorraussetzung ist eine bereits existierende Datenbank.

Der Befehl lautet:

psql [options] [dbname [username]]

Options:
-a Echo all input from script
-A Unaligned table output mode (-P format=unaligned)
-c COMMAND Run only single command (SQL or internal) and exit
-d DBNAME Specify database name to connect to (default: hubertba)
-e Echo commands sent to server
-E Display queries that internal commands generate
-f FILENAME Execute commands from file, then exit
-F STRING Set field separator (default: "|") (-P fieldsep=)
-h HOSTNAME Specify database server host (default: local socket)
-H HTML table output mode (-P format=html)
-l List available databases, then exit
-n Disable enhanced command line editing (readline)
-o FILENAME Send query results to file (or |pipe)
-p PORT Specify database server port (default: 5432)
-P VAR[=ARG] Set printing option 'VAR' to 'ARG' (see \pset command)
-q Run quietly (no messages, only query output)
-R STRING Set record separator (default: newline) (-P recordsep=)

-s Single step mode (confirm each query)
-S Single line mode (end of line terminates SQL command)
-t Print rows only (-P tuples_only)
-T TEXT Set HTML table tag attributes (width, border) (-P tableattr=)
-U NAME Specify database user name (default: hubertba)
-v NAME=VALUE Set psql variable 'NAME' to 'VALUE'
-V Show version information and exit
-W Prompt for password (should happen automatically)
-x Turn on expanded table output (-P expanded)
-X Do not read startup file (~/.psqlrc)

Der Beispielbefehl verwendet die Optionen '-f' um alle in dieser Datei stehenden Befehle auszuführen. weiters benötige ich '-d' um die Datenbank anzugeben die ich befüllen möchte. Um eine Datenbank im Netz zu füllen benötige ich natürlich die bereits bei vorhergehenden Befehlen verwendete Option '-h'.

Beispielbefehl um lokale Datenbank zu füllen

psql -f dump.out -d sedatenbank

Beispielbefehl um remote Datenbank zu füllen

psql -f dump.out -h db.irgendwo.org -d sedatenbank

4.5.Datenbank kaputt, was nun?

Hier eine kurze Ablaufbeschreibung was zu tun ist wenn die Datenbank abgestürzt ist und das starten verweigert.

Zuerst muss die gesamte alte Datenbank gelöscht werden.

dropdb sedatenbank

Dann wird die neue Datenbank erstellt.

createdb sedatenbank

Falls ein User benötigt wird, wird dieser angelegt.

createuser sedatenbank

Zum Schluss wird der Dump eingespielt.

psql -f dump.out sedatenbank