Hochschule für Technik und Wirtschaft Berlin

University of Applied Sciences

Datenbanken

Vorlesungsskript für das 3. Semester

Studiengang Internationale Medieninformatik

2. Grundlagen von SQL

Dozent: M. Sc. Burak Boyaci

Version: 14.10.2025

Wintersemester 25/26

2

Die Sprachelemente von SQL

Kapitelübersicht

SQL (Structured Query Language) ist eine Programmiersprache zur Definition von Datenstrukturen und zur Datenverwaltung relationaler Datenbanken. Es erschien erstmals 1974 und wurde 1986 ein Standard des ANSI, der Normbehörde der USA, sowie ein Jahr später der ISO. Die 2020 aktuelle Version ist SQL:2016.[1] Wir werden in diesem Kapitel eine Einführung in diese Programmiersprache kennenlernen.

2.1 * Informatischer Überblick

SQL ist eine „deklarative" Programmiersprache, d.h. im Gegensatz zu „imperativen" Sprachen wie Java, C oder Visual Basic wird nicht der Kontrollfluss, also der Ablauf des Programms programmiert, sondern nur „ergebnisorientiert" die Logik und die Funktionalität des Programms.
„Im Gegensatz zur imperativen Programmierung, bei der das *Wie* im Vordergrund steht, fragt man in der deklarativen Programmierung nach dem *Was*, das berechnet werden soll."[2]

Ferner ist SQL „Turing-vollständig", d.h. eine Programmiersprache, die logische WHILE Schleifenermöglicht,alsoSchleifen, diegrundsätzlich unendlichoftdurchlaufen werdenkönnen. Damit können alle berechenbaren Funktionen implementiert werden.[3]

SQL hat im Wesentlichen die Merkmale einer Interpretersprache, d.h. der Quelltext wird zur Laufzeit direkt übersetzt und ausgeführt. Die meisten RDBMS allerdings übersetzen den Quelltext intern in einen „Syntaxbaum" oder einen „Query Execution Plan", der als temporäre Datei zunächst gespeichert und dann ausgeführt wird (also ähnlich dem Bytecode von Java oder

[1] https://en.wikipedia.org/wiki/SQL
[2] https://de.wikipedia.org/wiki/Deklarative_Programmierung
[3] https://stackoverflow.com/questions/900055/, [Win]

C#).[4] In diesen Fällen ist SQL also eigentlich auch eine Compilersprache, allerdings ist die kompilierte Datei nur DBMS-intern ausführbar.

2.2 Reservierte Wörter

Jede Programmiersprache besitzt ein festes „Vokabular“ von sogenannten *reservierten Wörtern*. Sie haben eine festgelegte Bedeutung und dürfen nur nach bestimmten Grammatikregeln, der *Syntax*, kombiniert werden. Die wichtigsten reservierten Wörter von SQL, die auch auf fast allen Datenbanksystemen funktionieren, sind in Tabelle 2.1 aufgeführt. Sie umfasst ebenso

Wichtige reservierte Wörter in SQL

ABS	ADD	ALL	ALTER	AND	ANY	AS	AVG	BETWEEN	BY
CHECK	COLLATE	CONCAT	CONSTRAINT	COUNT	CREATE	DATE	DOMAIN	DOUBLE	DECIMAL
DELETE	DESC	DISTINCT	DROP	EXCEPT	EXISTS	FOREIGN	FROM	GRANT	GROUP
HAVING	IN	INNER	INSERT	INTEGER	INTERSECT	INTO	IS	JOIN	LEFT
LIKE	MAX	MIN	NOT	NULL	OF	ON	OR	ORDER	OUTER
POSITION	POWER	PRIMARY	RECURSIVE	REFERENCES	RESTRICT	REVOKE	RIGHT	ROUND	SELECT
SET	SMALLINT	SQRT	START	STDDEV	SUBSTRING	SUM	TABLE	TIME	TIMESTAMP
UNION	UNIQUE	UPDATE	USER	VALUES	VARCHAR	VARIANCE	VIEW	WHERE	WITH

Literale		**Besondere Zeichen**
NULL	0, 1, -2 | 1.2, 3.1E8 'Abc'	-| * | +, , /, % | =, !=, <>, >, <, >=, <=

Tabelle 2.1: Wichtige reservierte Wörter und Literale von SQL

Literale, also Zahlen und Zeichenfolgen, die einen festen Wert darstellen. Die in Apostrophe eingeklammerten Zeichenfolgen werden *Strings* genannt. Daneben gibt es Zeichenfolgen mit besonderer Bedeutung wie Wildcards, Kommentarmarkierungen oder Rechenoperationen.

SQL ist nicht schreibungssensitiv *(not case sensitive)*, d.h. es unterscheidet nicht zwischen Groß- und Kleinschreibung. (Bei einigen RDBMS, z.B. MariaDB auf Linux, wird die Schreibung von Tabellennamen allerdings sehr wohl unterschieden.) In diesem Skript wird die Konvention verwendet, reservierte Wörter in SQL in Großbuchstaben zu schreiben.

Obwohl SQL standardisiert ist, existieren viele Dialekte für die verschiedenen Datenbanksysteme, so dass ein einmal geschriebener SQL-Quelltext nicht unbedingt auf allen Datenbanken läuft. Der zentrale Grund ist, dass die reservierten Wörter zu einem gewissen Teil nicht identisch sind. Allein schon die Anzahl der reservierten Wörter variiert zwischen den verschiedenen Datenbanksystemen:

Anzahl reservierter Wörter

OpenOffice Base*	Microsoft Access	MariaDB* / MySQL	PostgreSQL*	Azure SQL / SQL-Server	Oracle
61	252	238	93	185	110

* Open Source / freie Software

Übrigens zum Vergleich: Übliche (imperative) Programmiersprachen haben deutlich weniger reservierte Wörter, z.B. hat Java 49, Python nur 30.

2.3 Kommentare

Wie in jeder Programmiersprache gibt es auch in SQL die Möglichkeit, Kommentare einzufügen, um anderen Programmierer*innen (und sich selbst) Hinweise zum Zweck oder zur Funktions-

[4] https://www.quora.com/Is-SQL-interpreted-or-compiled

weise des Programms zu geben. Kommentare werden von dem Interpreter ignoriert und daher nicht ausgeführt. Es gibt zwei Arten von Kommentaren in SQL, den einzeiligen Kommentar,

```
--Dies ist ein Kommentar
```

der durch zwei Minuszeichen für den nachfolgenden Rest der Zeile markiert wird, und der mehrzeilige (oder Block-) Kommentar,

```
/* Dies ist ein Kommentar,
der über mehrere Zeilen geht
 */
```

dessen Bereich mit /* geöffnet und mit */ geschlossen wird.

2.4 Datentypen

Datentypen sind die elementaren Speichereinheiten einer Programmiersprache, die jeweils einen bestimmten Speicherbereich zugewiesen bekommen (vgl. auch Abbildung 1.1 auf Seite 7). Datentypen legen die Wertebereiche dieser Einheiten fest. Es gibt in der Regel drei Kategorien von Datentypen, Zeichen *(character)*, Ganzzahlen *(integer)* und Gleitkommazahlen *(float, double)*. Aus der Aneinanderreihung von Zeichen entsteht Text *(string)* als ein zusammengesetzter Datentyp. Vgl. dazu die Hierarchie der Speicherkonzepte in Abbildung 1.1.

In SQL werden je nach RDBMS unterschiedliche Datentypen festgelegt. Aber gängige und in fast allen Dialekten verwendete sind diejenigen in Tabelle 2.2. Die Tabelle ist in die vier Bereiche

Datentyp	Speicher	Beschreibung
`varchar(n)`	n Byte	Text (String) mit Maximallänge n; wird mit Apostrophs `'Abc'` gekennzeichnet
`text`	x Byte	Text beliebiger Länge; wird mit Apostrophs `'Abc'` gekennzeichnet
`smallint`	2 Byte	Ganzzahl im Bereich von −32 768 bis 32 767 ($= -2^{15}$ bis $2^{15} - 1$)
`int`	4 Byte	Ganzzahl im Bereich von −2 147 483 648 bis 2 147 483 647 ($= -2^{31}$ bis $2^{31} - 1$)
`bigint`	8 Byte	Ganzzahl im Bereich von −9 223 372 036 854 775 808 bis 9 223 372 036 854 775 807 ($= -2^{63}$ bis $2^{63} - 1$)
`float`	4 Byte	Gleitkommazahl im Bereich von `1.0E-38` bis `3E+38` ($= 2^{-126}$ bis 2^{128}), auf bis zu 8 Dezimalstellen genau
`double`	8 Byte	Gleitkommazahl mit einem Betrag von `2E-308` bis `2E+308` ($= 2^{-1022}$ bis 2^{1024}) auf bis zu 16 Dezimalstellen genau
`decimal(p,s)`	$f(p)$ Byte*	Festkommazahl mit maximal p Stellen, davon s Nachkommastellen; Restriktionen: $p \leqq 38$ bei SQL Server ($p \leqq 65$ bei MariaDB) und $0 \leqq s \leqq p$
`date`	3 Byte	Datum, je nach RDBMS im Format `'yyyy-mm-dd'` oder `#mm/dd/yyyy#`
`time`	4 Byte	Zeit, je nach RDBMS meist im Format HH:MM:SS
`timestamp`	8 Byte	Datum und Uhrzeit im Format `'yyyy-mm-dd hh:mm:ss'` (SQL Server: `datetime`)

Tabelle 2.2: Gängige Datentypen in SQL. * Erläuterung: $f(p) = 1 + 4 \cdot \left\lceil \frac{p}{9{,}5} \right\rceil$

Strings, Ganzzahlen, Gleitkommazahlen und Datum/Zeit unterteilt. Zeichen und Text werden in SQL mit Apostroph 'Abc' gekennzeichnet, Kommazahlen mit einem Punkt als Dezimaltrennzeichen. Siehe dazu auch die Übersicht über die Literale in der obigen Tabelle 2.1. Besonderheiten beispielsweise von MS Access sind, dass dort der Datentyp **Integer** nur 2 Byte hat und der Datentyp **Long Integer**4 Byte (wie hier **int**); ferner heißt der Datentyp **float** dort **Single**. [5] Für weitere Information zu den Datentypen seien die Dokumentationen

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

[5] https://support.office.com/en-us/article/30ad644f-946c-442e-8bd2-be067361987c; im Widerspruch dazu jedoch https://docs.microsoft.com/en-us/office/client-developer/access/desktop-database-reference/ equivalent-ansi-sql-data-types

und

https://www.w3schools.com/sql/sql_datatypes.asp.

empfohlen.

2.5 Anweisungen

Ein SQL-Programm besteht aus einer oder mehreren Anweisungen, auch Befehle oder Statements genannt. Zwei Anweisungen werden in SQL mit einem Semikolon getrennt. Für die letzte auszuführende Anweisung darf es weggelassen werden.
SQL ist, anders als der Name vielleicht vermuten lässt, keine reine Abfragesprache für Daten, sondern kann zusätzlich sowohl die Struktur der Datenbank als auch Zugriffsrechte verwalten. Entsprechend ist der Befehlssatz von SQL in drei Teilbereiche gegliedert, die DCL, die DDL und die DML. Die *Data Control Language (DCL)* ist der Sprachteil von SQL, der für die Verwaltung von Zugriffsrechten auf die Datenbank zuständig ist. Wir werden uns in einem späteren Kapitel im Zusammenhang mit Zugriffsrechten die DCL anschauen.

Data Definition Language (DDL) Mit der Data Definition Language können eine Datenbank und die Struktur ihrer Tabellen verwaltet werden. Konkret können Datenbanken, Tabellen und User angelegt, verändert und gelöscht werden. Dazu stehen die Ausdrücke CREATE, ALTER und DROP zur Verfügung, die kombiniert werden können mit den Ausdrücken DATABASE, TABLE und USER:

$$\left\{\begin{array}{c}\texttt{CREATE}\\ \texttt{ALTER}\\ \texttt{DROP}\end{array}\right\} + \left\{\begin{array}{c}\texttt{DATABASE}\\ \texttt{TABLE}\\ \texttt{USER}\end{array}\right\} + \textit{Name} + \textit{[Zusatzoptionen]};\qquad (2.1)$$

Der Relationentyp wird in SQL definiert, indem die Spalten mit ihrem Datentyp aufgeführt werden:

```
CREATE TABLE tabelle (
   spalte_1 datentyp_1,
   spalte_2 datentyp_2,
     ...
   spalte_n datentyp_n
);
```

Auf diese Weise wird also der Relationentyp

Tabelle (datentyp_1, …, datentyp_n)

definiert. Der Relationentyp einer Tabelle wird oft auch *Schema* oder *Struktur* der Tabelle genannt. In den meisten Datenbanksystemen kann man den Zeichensatz festlegen, in dem die Werte in Spalten mit Textformaten gespeichert werden. Zu empfehlen ist hier der Zeichensatz UTF-8:

```
CREATE TABLE tabelle (
   spalte_1 datentyp_1,
     ...
) DEFAULT CHARSET=utf8;
```

Dieser Zeichensatz codiert den Unicode und ermöglicht die Buchstaben aller Verkehrssprachen der Welt sowie zahlreiche technische und mathematische Symbole. Aus Gründen der Leserlichkeit werden wir diesen Zusatz in Zukunft nicht weiter berücksichtigen.
Durch **ALTER** können Tabellen um Spalten ergänzt werden, mit **DROP** wird eine Tabelle vollständig gelöscht.

```
DROP TABLE tabelle (
  spalte_1 datentyp_1,

  ... );
```

```
ALTER TABLE tabelle
ADD column_name datatype;
```

Data Manipulation Language (DML) Mit der Data Manipulation Language (DML) werden Anweisungen für die Verwaltung der Daten selbst bereitgestellt, d.h. man kann damit die Datensätze anlegen, lesen, aktualisieren und löschen. Diese vier zentralen Funktionen auf Daten werden mit dem Kürzel CRUD (für create, read, update, delete) zusammengefasst. Beispielsweise werden in eine existierende Tabelle Datensätze eingefügt, indem der Ausdruck **INSERT** verwendet wird:

```
INSERT INTO tabelle
  (spalte_1,...,spalte_n)
  VALUES
 (wert_11,...,wert_1n),
  ...
  (wert_m1,...,wert_mn);
```

Hiermit werden *m* Datensätze mit ihren Werten für die *n* Spalten eingefügt. Wichtig ist, dass die im ersten Klammerpaar angegebene Reihenfolge der Spalten der Reihenfolge der Werte entsprechen muss. Bei manchen RDBMS kann man pro INSERT nur einen Datensatz einfügen, z.B. bei Oracle. Will man alle Spalten der Tabelle füllen, kann man die Klammern mit den Spaltennamen vor **VALUES** auch weglassen. Dazu muss allerdings die genaue Anzahl und Anordnung der Spalten in dem Tabellenschema eingehalten werden.
Daten können in SQL mit **SELECT** gelesen werden. Wir werden die Anweisung im Laufe dieses Skripts genauer kennenlernen. Die einfachste Variante lautet wie folgt:

```
SELECT * FROM tabelle
```

Sie zeigt die gesamte Tabelle mit dem Namen tabelle an, d.h. all ihre Spalten und Datensätze. Die Ausdrücke **UPDATE** zum Ändern eines Datensatzes und **DELETE** zum Löschen von Datensätzen dagegen werden wir zu einem späteren Zeitpunkt kennenlernen.

Beispiel 2.1. Wir werden in diesem Beispiel als erste Anwendung der betrachteten SQL-Anweisungen die Realisierung des Relationstyps Full House aus Beispiel 1.2 als

Tabelle betrachten:

```
CREATE TABLE full_house (
  farbe varchar(5), karte
  varchar(5)
);
```

Mit dem Befehl

```
INSERT INTO full_house (farbe, karte) VALUES
  ('Kreuz','Ass'),
  ('Pik','Ass'),
  ('Herz','König'),
  ('Herz','Ass'),
  ('Karo','König');
```

werden die Daten darin gespeichert. Die Tabelle können wir uns dann mit der Anweisung

```
SELECT * FROM full_house;
```

ausgeben lassen:

farbe	karte
Kreuz	Ass
Pik	Ass
Herz	König
Herz	Ass
Karo	König

2.6 Constraints

In Datenbanken ist es oft sinnvoll bestimmte Regeln für einzelne Spalten relationaler Datenbanken zu definieren. Constraints übernehmen diese Aufgabe. Es kann beispielsweise sinnvoll sein für einen Eintrag in eine Datenbanktabelle keine Nullwerte zuzulassen. Folgende Constraints sind möglich:

- NOT NULL: die Spalte innerhalb der Datenbank darf nicht leer sein (= NULL)
- UNIQUE: jeder Eintrag innerhalb einer Spalte muss einzigartig sein (darf sich nicht doppeln)
- PRIMARY KEY: Kombination aus NOT NULL und UNIQUE – ist ein eindeutiger Schlüssel zur Identifikation einer Zeile innerhalb einer Datenbank
- FOREIGN KEY: stellt die Verbindung zwischen zwei Tabellen her
- CHECK: stell sicher, dass eine bestimmte Bedingung erfüllt wird
- DEFAULT: vergibt einer Spalte einen Standardwert, wenn nichts anderes definiert wird

2.7 Funktionen

In SQL sind einige Standardfunktionen definiert, die als Parameter geeignete Attributwerte der einzelnen Datensätze einer Tabelle erhalten können und einen Wert zurückliefern. Sie werden oft auch *Single-Row-Funktionen* genannt, da sie auf einzelnen Datensätzen einsetzbar sind. Einige Beispiele sind die folgenden mathematischen und string bearbeitenden Funktionen:

Elementare mathematische Funktionen:

ABS(x)	$\lvert x \rvert$
POWER(x, y)	x^y
ROUND(x, n)	x auf n Nachkommastellen runden
SQRT(x)	$\sqrt{x}$

Wichtige String Funktionen:

CONCAT('s1', ..., 'sn')	Hängt die Strings s1 bis sn aneinander
POSITION('teil'**IN**'String')	Position des Teilstrings im Text; 0, wenn nicht enthalten
SUBSTRING('Text', *x* [, *y*])	Teilstring ab Position *x* [optional nur bis Position *x* + *y*]
TRIM(' Text ')	entfernt führende und hängende

Hierbeistellen allevon Apostrophe eingeklammerten Zeichenfolgen einen beliebigen Stringdar, die Variablen *x*, *y* und *n* dagegen Zahlen. Die Zeichenfolgen und Zahlen können Konstanten sein oder Attributwerte eines Datensatzes annehmen. Dabei ist zu beachten, dass in Funktion mit mehreren Parametern nur Attributwerte aus demselben Datensatz eingesetzt werden können.

Beispiel 2.2. Wenden wir auf unsere Tabelle full_house in Beispiel 2.1 die Anweisung

```
SELECT SUBSTRING (farbe, 1, 2) FROM full_house;
```

an, so werden für von der Farbe jedes der 5 Datensätze wegen *x* = 1 und *y* = 2 als Konstanten die ersten zwei Buchstaben angezeigt:

SUBSTRING(farbe, 1, 2)
Kr
Pi
He
He
Ka

Das Ergebnis ist also eine neue Tabelle mit nur einer Spalte, aber fünf Datensätzen. Wir können auch mehrere Attributwerte eines Datensatzes miteinander kombinieren. Beispielsweise erhalten wir mit

```
SELECT CONCAT ('{', farbe, '-' , karte, '}') FROM full_house;
```

die Aneinanderreihung („Konkatenation“) der Farbeund Kartejedes Datensatzes, abgeschlossen und verbunden mit konstanten Zeichen:

CONCAT(’{’, farbe, ’ ’, karte, ’}’)
{Kreuz Ass}
{Pik Ass}
{Herz König}
{Herz Ass}
{Karo König}

Wir können darüber hinaus Funktionen ineinander verschachteln.

Zwischenfrage 2.3. Betrachten wir die Tabelle full_houseaus dem obigen Beispiel 2.1 und die folgende Anweisung mit verschachtelten Funktionen:

```
SELECT CONCAT (SUBSTRING(farbe,1,1), '-', SUBSTRING(karte,1,1)) FROM full_house;
```

Was ist die Ausgabe?

3

Die SELECT-Anweisung

Kapitelübersicht

3.1 Einfache SELECT-Anweisungen

Die wichtigste und mächtigste Anweisung in SQL ist **SELECT**. Sie dient der Abfrage auf die in einer Tabelle gespeicherten Daten. Eine solche Abfrage kann beliebig komplex werden. Beginnen wir mit dem Einfachen, um zunächst die grundsätzliche Funktionsweise zu verstehen. Die **SELECT**-Anweisung ist eine Auswahl der Daten von Spalten einer Tabelle, also eine *Selektion* von Daten. Die Syntax lautet:

```
SELECT spalte_1, .. , spalte_n FROM tabelle;
```

Die Spalten (spalte_1,..... , spalte_n) müssen dabei eine Auswahl der Spalten der betrachteten Tabelle sein. Die Ausgabe eines SELECTS ist eine Tabelle mit den Spalten spalte_1, ... , spalte_n und aus den Datensätzen der Tabelle besteht, in der die nicht ausgewählten Spalten also weggelassen sind:

spalte $_1$	. . .	spalte $_n$
$wert_{1,1}$	. . .	$wert_{1,n}$
.		.
$wert_{m,1}$	. . .	$wert_{m,n}$

Diese Ausgabe heißt *Ergebnismenge*. Die Reihenfolge der Spalten der Ergebnismenge ist dabei durch die Reihenfolge der Spaltenauswahl im SELECT festgelegt. Will man alle Spalten

der Tabelle auswählen, so kann man den Stern $*$ als Wildcard verwenden:

```
SELECT * FROM tabelle;
```

Beispiel 3.1. Betrachten wir unsere Tabelle full_house in Beispiel 2.1. Die Ergebnisse dreier möglicher SELECT-Anweisungen sind dann beispielsweise jeweils:

SELECT farbe, karte **FROM** full_house;

farbe	karte
Kreuz	Ass
Pik	Ass
Herz	König
Herz	Ass
Karo	König

SELECT farbe **FROM** full_house;

farbe
Kreuz
Pik
Herz
Herz
Karo

SELECT karte **FROM** full_house;

karte
Ass
Ass
König
Ass
König

Die erste Tabellenausgabe hätte man ebenso mit dem Befehl

```
SELECT * FROM full_house;
```

erreicht, da der Relationstyp full_house(farbe, karte) die Attribute farbe und karte genau in dieser Reihenfolge besitzt.

Bemerkung 3.2. mathematisch betrachtet bildet eine Selektion

SELECT $A_{j_1}, A_{j_2}, \ldots, A_{j_k}$ **FROM** R;

die Tabelle $R(A_1, \ldots, A_n)$ mit n Attributen auf eine Ergebnismenge ab, die eine neue Tabelle $R(A_{j_1}, \ldots, A_{j_k})$ mit nur k Attributen ist, wobei $k \leq n$ und $j_1, \ldots, j_k \in \{1, \ldots, n\}$ gilt:

A_1	A_2	A_3	$\cdots$	A_n
x_{11}	x_{12}	x_{13}	$\cdots$	x_{1n}
$\vdots$	$\vdots$	$\vdots$		$\vdots$
x_{i1}	x_{i2}	x_{i3}	$\cdots$	x_{in}
$\vdots$	$\vdots$	$\vdots$		$\vdots$
x_{m1}	x_{m2}	x_{m3}	$\cdots$	x_{mn}

$\xrightarrow{\text{SELECT}}$

A_{j_1}	A_{j_2}	$\cdots$	A_{j_k}
x_{1j_1}	x_{1j_2}	$\cdots$	x_{1j_k}
$\vdots$	$\vdots$		$\vdots$
x_{ij_1}	x_{ij_2}	$\cdots$	x_{ij_k}
$\vdots$	$\vdots$		$\vdots$
x_{mj_1}	x_{mj_2}	$\cdots$	x_{mj_k}

(3.1)

Ein **SELECT** „projiziert“ also eine Tabelle auf weniger (oder gleich viel) Spalten. Die Ergebnismenge kann jedoch doppelte Zeilen (Dubletten) enthalten. Um dies zu verhindern, kann man das reservierte Wort **DISTINCT** verwenden:

SELECT DISTINCT $A_{j_1}, A_{j_2}, \ldots, A_{j_k}$ **FROM** R;

Ist die ursprüngliche Tabelle eine Relation, so haben wir mit **SELECT DISTINCT** also eine Abbildung von einer Relation $R(A_1, \ldots, A_n)$ mit n Attributen auf eine neue Relation $R'(A_{j_1}, \ldots, A_{j_k})$ mit k Attributen, aber nun mit nur noch r statt m Tupeln („Datensätzen“):

R

A_1	A_2	A_3	$\cdots$	A_n
x_{11}	x_{12}	x_{13}	$\cdots$	x_{1n}
$\vdots$	$\vdots$	$\vdots$		$\vdots$
x_{i1}	x_{i2}	x_{i3}	$\cdots$	x_{in}
$\vdots$	$\vdots$	$\vdots$		$\vdots$
x_{m1}	x_{m2}	x_{m3}	$\cdots$	x_{mn}

$\xrightarrow{\text{SELECT DISTINCT}}$

R'

A_{j_1}	A_{j_2}	$\cdots$	A_{j_k}
$x_{q_1 j_1}$	$x_{q_1 j_2}$	$\cdots$	$x_{q_1 j_k}$
$x_{q_2 j_1}$	$x_{q_2 j_2}$	$\cdots$	$x_{q_2 j_k}$
$\vdots$	$\vdots$		$\vdots$
$x_{q_r j_1}$	$x_{q_r j_2}$	$\cdots$	$x_{q_r j_k}$

(3.2)

$(q_1, \ldots, q_r \in \{1, \ldots, m\})$. Ein **SELECT** projiziert also eine Tabelle auf weniger Spalten, ein **SELECT DISTINCT** auf weniger Spalten *und* weniger Zeilen. Daher spricht man in der Theorie der Relationen bei der Selektion mit **DISTINCT** auch von einer *Projektion* der Relation R auf die Relation $R^{[}$, d.h. in Symbolen

$$p : R \to R^{[}. \tag{3.3}$$

Sie ist bezüglich der Mengen der Relationen „abgeschlossen“. [7]

Beispiel 3.3. Betrachten wir wieder unser Beispiel 2.1, so erhalten wir mit SELECT DISTINCT die folgende Ergebnismenge:

```
SELECT DISTINCT karte FROM full_house;
```

karte
Ass
König

Im Gegensatz zur dritten Ergebnismenge aus Beispiel 3.1 erhalten wir nun also eine Ergebnismenge ohne doppelte Einträge, d.h. mit Bemerkung 3.2 eineneue Relation.

Beispiel 3.4. Betrachten wir die folgende Tabelle, die verschiedene Comic-Alben speichert.

```
--Tabellenstruktur:
CREATE TABLE alben (
   titel varchar(50),
   reihe varchar(50),
   band smallint,
   preis decimal(4,2),
   jahr smallint
);
--Daten:
INSERT INTO alben (titel, reihe, band, preis, jahr) VALUES
   ('Asterix, der Gallier'      , 'Asterix', 1, 2.80, 1968),
   ('Asterix und Kleopatra'   , 'Asterix', 2, 2.80, 1968),
   ('Gespenster Geschichten' , 'Gespenster Geschichten' , 1, 1.20, 1974),
   ('Die Trabantenstadt' , 'Asterix', 17, 3.80, 1974),
   ('Der große Graben' , 'Asterix', 25, 5.00, 1980),
```

[7] Piepmeyer (2011):§4.1.

alben

titel	reihe	band	preis	jahr
Asterix, der Gallier	Asterix	1	2.80	1968
Asterix und Kleopatra	Asterix	2	2.80	1968
Gespenster Geschichten	Gespenster Geschichten	1	1.20	1974
Die Trabantenstadt	Asterix	17	3.80	1974
Der große Graben	Asterix	25	5.00	1980
Das Kriminalmuseum	Franka	1	8.80	1985
Das Meisterwerk	Franka	2	8.80	1986

Tabelle 3.1: Beispieldaten der Tabelle alben. Vgl. Piepmeyer (2011: S. 172)

Die Auswahl der Spalten reihe, preisliefert dann, jeweils ohne und mit **DISTINCT**, die

folgenden Ausgaben:

```
SELECT reihe, preis FROM alben;
```

reihe	preis
Asterix	2.80
Asterix	2.80
Gespenster Geschichten	1.20
Asterix	3.80
Asterix	5.00
Franka	8.80
Franka	8.80

```
SELECT DISTINCT reihe, preis FROM alben;
```

reihe	preis
Asterix	2.80
Gespenster Geschichten	1.20
Asterix	3.80
Asterix	5.00
Franka	8.80

Hier ergibt also nur die Selektion mit DISTINCT eine Relation.

Bemerkung 3.7. * Da die Anweisung **SELECT DISTINCT** mit den Vereinbarungen aus Bemerkung 1.3 eine Relation auf eine Relation abbildet, nennt man sie „abgeschlossen“ bezüglich Relationen.[8]

[8]vgl. Piepmeyer(2011):§4.2.

[9]vgl. Piepmeyer (2011):S. 88ff.

3.2 Alias-Spalten mit AS

Mit einer SELECT-Abfrage kann man nicht nur bestehende Spalten einer Tabelle auswählen, sondern auch völlig neue Spalten aus anderen Spalten oder mit Funktionen berechnete Werte erstellen. Der Name solcher Spalten heißt *Alias* und wird mit dem reservierten Wort **AS** vergeben:

```
SELECT ... ausdruck AS alias ... FROM...
```

Hierbei ist ausdruck ein aus Spaltenwerten bestimmter Wert oder das Ergebnis einer Funktion. Das Wort **AS** kann auch entfallen. (Es sollte aber m.E. aus Gründen der Lesbarkeit einer Abfrage immer verwendet werden.)

Beispiel 3.8. Wollen wir uns für jedes unserer Comic-Alben die Mehrwertsteuer und die Nettopreise berechnen und anzeigen lassen, so können wir dazu Aliasse verwenden:

```
SELECT titel,
       preis AS brutto, ROUND(preis/1.19, 2)
       AS netto,
       ROUND(0.19/1.19*preis, 2) AS mwst
FROM alben;
```

Diese Abfrage ergibt dann:

titel	brutto	netto	mwst
Asterix, der Gallier	2.80	2.35	0.45
Asterix und Kleopatra	2.80	2.35	0.45
Gespenster Geschichten	1.20	1.01	0.19
Die Trabantenstadt	3.80	3.19	0.61
Der große Graben	5.00	4.2	0.8
Das Kriminalmuseum	8.80	7.39	1.41
Das Meisterwerk	8.80	7.39	1.41

Ebenso wie für Spaltennamen können wir auch für Tabellennamen Aliasse verwenden. Zum Beispiel könnten wir die Abfrage aus Beispiel 3.7 äquivalent formulieren als:

```
SELECT a.titel,
       a.preis AS brutto, ROUND(a.preis/1.19, 2)
       AS netto, ROUND(0.19/1.19*a.preis, 2)
       AS mwst
FROM alben AS a;
```

Hier wird also die Tabelle albenmit dem Alias aversehen, das dann innerhalb der Abfrage als gültiger Tabellenname verwendet werden kann, wie hier als Referenzen a.titel und a.preis für die Spalten titel und preis in der Tabelle alben. In diesem Beispiel mit nur einer Tabelle macht ein Alias für Tabellen keinen Sinn. Für das Arbeiten mit mehreren Tabellen können Aliasse die Schreibarbeit bei Abfragen erleichtern.

Zwischenfrage 3.9. Informatisch ist die Abfrage in Beispiel 3.7 zwar völlig korrekt, aber leider ist die Mehrwertsteuer auf Bücher ermäßigt auf 7 %. Wie muss die Abfrage also lauten, um sich die Titel und die wirtschaftlich korrekten Mehrwertsteuerbeträge anzeigen zu lassen?

3.3 Auswahlfilter: Die WHERE-Klausel

Mit der Konstruktion

```
SELECT spalte_1, ..., spalte_n FROM tabelle WHERE bedingung;
```

können wir eine Auswahl von Spalten einer Tabelle treffen, wobei nur Zeilen berücksichtigt werden, die der nach dem reservierten Wort **WHERE** Bedingung genügen. Die Bedingung wird auch WHERE-Klausel genannt und filtert also die gewünschten Datensätze aus allen Datensätzen der Tabelle. In der Terminologie der Relationenalgebra heißt die Bedingung *Prädikat*.[10] Meist wird die WHERE-Klausel mit einem Vergleich zu bestimmten Werten gebildet, d.h. mit einem der folgenden *Vergleichsoperatoren* gebildet:

=	<>, !=	>	<	>=	<=
gleich	ungleich	echt größer	echt kleiner	größer gleich	kleiner gleich

(3.4)

In einigen RDBMS funktioniert der Operator != nicht, hier muss der Operator <> verwendet werden!

Beispiel 3.10. Betrachten wir unsere Full House Tabelle aus Beispiel 2.1. (a) Wie finden wir heraus, welche Karten in dem Full House haben die Farbe Herz? (b) Wie lauten Farbe und Karte der Spielkarten des Full House, die nicht die Farbe Herz haben?

```
SELECT karte FROM full_house
WHERE farbe = 'Herz';
```

karte
Ass
König

```
SELECT farbe, karte FROM full_house
WHERE farbe <> 'Herz';
```

farbe	karte
Kreuz	Ass
Pik	Ass
Karo	König

[10] Piepmeyer (2011):§§4.4, 4.7.

Die zweite WHERE-Klausel hätten wir auch mit !=statt <>versehen können.

Beispiel 3.11. Wie finden wir in unserer Comics-Datenbank aus Beispiel 3.4 diejenigen Alben heraus, für die weniger als 1 € Mehrwertsteuer anfallen? Wir können dazu die Abfrage aus Beispiel 3.7 modifizieren und mit der WHERE-Klausel die geeigneten Datensätze filtern:

```
SELECT titel, ROUND(0.19/1.19*preis, 2) AS mwst FROM alben
WHERE 0.19/1.19*preis < 1;
```

(Leider ermöglicht SQL nicht die Verwendung von Aliassen desselben SELECTs in der WHERE-Klausel!) Die Abfrage liefert:

titel	mwst
Asterix, der Gallier	0.45
Asterix und Kleopatra	0.45
Gespenster Geschichten	0.19
Die Trabantenstadt	0.61
Der große Graben	0.8

In der klassischen Logik ist eine Bedingung eine logische Aussage, die wahr oder falsch sein kann. Die WHERE-Klausel filtert in diesem Sinne nur diejenigen Datensätze, für die diese Aussage wahr ist. In SQL können wir, wie in der Aussagenlogik und somit auch in anderen Programmiersprachen bekannt, mehrere Aussagen logisch miteinander verknüpfen. Die mathematische Grundlage dafür ist die Boole'sche Algebra.

Die wichtigsten dafür notwendigen logischen Operatoren sind **NOT**, **AND** und **OR**, also die logische Verneinung, das logische ODER und das logische UND.

Beispiel 3.12. Wie finden wir in unserer Comics-Datenbank aus Beispiel 3.4 diejenigen Alben heraus, für die weniger als 1 € Mehrwertsteuer anfallen, aber nach 1975 erschienen sind? Wir müssen dazu nur die WHERE-Klausel aus Beispiel 3.10 mit **AND** um eine weitere Filterbedingung erweitern:

```
SELECT titel FROM alben WHERE 0.19/1.19*preis < 1 AND jahr > 1975;
```

Die Abfrage liefert in unserem Fall nur eine einzige Zeile:

titel
Der große Graben

wie wir durch Vergleich der Tabelle 3.1 erkennen können.

Wollen wir Zahlen oder Datumswerte einer Spalte x aus einem Wertebereich $x \in$ [wert_1, wert_2] filtern, so können wir dies mit einer AND-Verknüpfung der beiden Ungleichungen x S wert_1 und x S wert_2erreichen:

```
SELECT ... FROM tabelle WHERE x >= wert_1 AND x <=wert_2
```

Wir können in SQL auf Wertebereiche von Zahlen oder Datumsangaben auch den Ausdruck **BETWEEN** ... **AND** ... verwenden:

```
SELECT ... FROM tabelle WHERE x BETWEEN wert_1 ANDwert_2
```

Hierbei muss wert_1kleiner gleich wert_2sein. Näheres siehe z.B. unter https://www.w3schools.com/sql/sql_between.asp.

Beispiel 3.13. Wie finden wir in unserer Comics-Datenbank aus Beispiel 3.4 diejenigen Alben heraus, die von 1970 bis 1980 erschienen sind? Dazu nutzen wir den Ausdruck **between:**

```
SELECT titel FROM alben WHERE jahr BETWEEN 1970 AND1980;
```

Die Abfrage liefert in unserem Fall drei Datensätze:

titel
Gespenster Geschichten
Die Trabantenstadt
Der große Graben

wie in Tabelle 3.1 abzulesen ist.

Wir werden die logischen Operatoren von SQL später genauer untersuchen. Nur so viel sei hier schon verraten: SQL bietet mehr als die klassische Logik!

3.4 Textsuche mit LIKE

Der Operator **LIKE** dient in Suchbedingungen zum Abfragen von Textmustern über Schablonen und Wildcards. Die Syntax lautet:

```
... WHERE ausdruck LIKE textmuster ...
```

Hierbei ist ausdruck ein Spaltenname oder ein String 'abc', und textmuster ein String mit der Wildcard %oder der Schablone _. Durch den Ausdruck werden diejenigen Datensätze gefiltert, diedemMusterentsprechen. Diefolgenden Beispiele sollendie WirkungvonLIKEverdeutlichen:

Fragestellung	SQL-Abfrage
Welche Albentitel beginnen mit ‘D’?	**SELECT** titel **FROM** alben **WHERE** titel **LIKE** **'D%'**;
Welche Albentitel enden mit ‘n’?	**SELECT** titel **FROM** alben **WHERE** titel **LIKE** **'%n'**;
Welche Albentitel enthalten ein ‘n’?	**SELECT** titel **FROM** alben **WHERE** titel **LIKE** **'%n%'**;
Welche Albentitel haben als 3. Buchstaben ein ‘r’?	**SELECT** titel **FROM** alben **WHERE** titel **LIKE** **'_r%'**;

Die Wildcard beinhaltet insbesondere den leeren String. Die Überprüfung auf Treffer ist für die meisten RDBMS *case insensitive*, unterscheidet also nicht zwischen Groß- und Kleinschreibung der Buchstaben. Beachte: Bei MS Access ist die Wildcard durch * und die Schablone durch "‚?‘ gegeben: [11]

Bezeichnung	Art der Treffer	SQL-Standard	Access
Wildcard	mehrere beliebige Zeichen	%	*
Schablone	einzelnes beliebiges Zeichen	_	?

(3.5)

Vgl. https://www.w3schools.com/sql/sql_like.asp. Die Verneinung der **LIKE**-Operators ist **NOT LIKE**.

[11] https://support.office.com/de-de/article/b2f7ef03-9085-4ffb-9829-eef18358e931

Beispiel 3.14. Um herauszufinden, welche Albentitel einer Reihe angehören, die ein ‘x’ enthält, können wir den LIKE-Operator verwenden. Um das Gegenteil herauszufinden, welche Albentitel einer Reihe angehören, die kein ‘x’ enthält, kann man ihn mit **NOT** kombinieren:

```
SELECT titel FROM alben
WHERE reihe LIKE'%x%'
;
```

titel
Asterix, der Gallier
Asterix und Kleopatra
Die Trabantenstadt
Der große Graben

```
SELECT titel FROM alben
WHERE reihe NOTLIKE'%x%';
```

titel
Gespenster Geschichten
Das Kriminalmuseum
Das Meisterwerk

Insbesondere wird also mit % auch der leere String als Zeichen erfasst, so dass hier das ‘x’ als Endbuchstabe berücksichtigt wird. Verwenden wir eine MS Access Datenbank, so müssen wir stattdessen mit `LIKE'*x*'` die geeignete Wildcard verwenden.

Wollen wir uns alle Titel anzeigen lassen, deren dritter Buchstabe ein ‘s’ ist, so lautet die Abfrage

```
SELECT titel FROM alben WHERE titel LIKE '_s%';
```

(also mit zwei Unterstrichen!) Sie liefert die Ergebnismenge

titel
Gespenster Geschichten
Das Kriminalmuseum
Das Meisterwerk

3.5 Sortierung mit ORDER BY

Häufig ist es erwünscht, sich die Ergebnismenge einer Datenbankabfrage sortiert anzeigen zu lassen. In SQL gibt es dazu die Komponente **ORDER BY**, die an das Ende der zu sortierenden **SELECT**-Anweisung angefügt wird:

```
SELECT spalte_1, ..., spalte_n FROM tabelle ORDER BY spalte_j1, ..., spalte_jk;
```

Hinter **BY** werden als Sortierkriterium die Spalten aufgelistet, nach denen sortiert werden soll. Das bedeutet, dass zunächst nach Spalte spalte_j1 sortiert wird, falls Einträge den gleichen Wert für spalte_j1 haben, wird nach der nächsten Spalte spalte_j2 der Liste sortiert, usw., bis zur letzten Spalte spalte_jkder Liste. Die Spalten dieser Liste müssen dabei nicht unbedingt auch in der **SELECT**-Liste enthalten sein.

Beispiel 3.15. Wollen wir uns die Reihen der Comics aus Beispiel 3.4 sortiert anzeigen und danach, bei Alben der gleichen Reihe, nach dem Jahr, so schreiben wir folgende Anweisung:

```
SELECT reihe, titel, jahr FROM alben ORDER BY reihe, jahr;
```

Die Ergebnismenge lautet dann

reihe	titel	jahr
Asterix	Asterix, der Gallier	1968
Asterix	Asterix und Kleopatra	1968
Asterix	Die Trabantenstadt	1974
Asterix	Der große Graben	1980
Franka	Das Kriminalmuseum	1985
Franka	Das Meisterwerk	1986
Gespenster Geschichten	Gespenster Geschichten	1974

Wie erwartet erfolgt die Sortierung nach Strings alphabetisch, nach Zahlen numerisch.

Wie wir an dem obigen Beispiel erkennen, sortiert SQL die Daten in aufsteigender Reihenfolge. Wollen wir dagegen für ein (oder mehrere) Spalten eine Sortierung in *absteigender* Reihenfolge, so müssen wir hinter die betreffende Spalte das reservierte Wort **DESC** schreiben, also

```
SELECT ... FROM tabelle ORDER BY ..., spalte_ji DESC,...
```

Um ausdrücklich zu betonen, dass aufsteigend sortiert werden soll, kann man den Ausdruck **ORDER BY** auch durch **ASC** für *ascending* abschließen. (Strenggenommen ist ASC aber überflüssig.)

Beispiel 3.16. Wollen wir uns die Reihen der Comics aus Beispiel 3.4 sortiert anzeigen und danach, bei gleicher Reihe, nach dem Jahr, so erreichen wir dies mit der folgenden Anweisung:

```
SELECT reihe, titel, jahr FROM alben ORDER BY reihe, jahr DESC;
```

Die Ergebnismenge lautet dann

reihe	titel	jahr
Asterix	Der große Graben	1980
Asterix	Die Trabantenstadt	1974
Asterix	Asterix und Kleopatra	1968
Asterix	Asterix, der Gallier	1968
Franka	Das Meisterwerk	1986
Franka	Das Kriminalmuseum	1985
Gespenster Geschichten	Gespenster Geschichten	1974

Die Sortierung nach der Reihe bleibt also alphabetisch aufsteigend, die Sortierung bei gleicher Reihe nach Jahren dagegen absteigend. □

Bemerkung 3.17. *(Kollation)* Eigentlich ist ja klar, was „alphabetische Sortierung“ bedeutet, nicht wahr? Bei genauerem Hinsehen gibt es da aber viele Mehrdeutigkeiten. Wasist mit Groß- und Kleinbuchstaben? Ist ‘a’ vor oder nach ‘A’oder ist die Schreibung groß/klein egal? Wasist mit Umlauten? Wie verhält sich also zum Beispiel ‘Ä’zu ‘A’?Sind Ziffern vor Buchstaben einzuordnen? Was ist mit Buchstaben aus dem türkischen, griechischen, chinesischen, arabischen oder hebräischen Alphabet?

Die genaue Sortierungsregel für Buchstaben und Zeichen in Einträgen einer Datenbank heißt *Kollation*. Sie kann bei vielen RDBMS insgesamt oder individuell für einzelne Tabellen konfiguriert werden. In MS Access zum Beispiel wird die Kollation über die allgemeinen Einstellungen konfiguriert, im SQL-Standard ist sie bei der Erstellung einer Datenbank mit dem reservierten Wort **COLLATE** möglich. Um möglichst viele Schriftzeichen zu erlauben, wird üblicherweiseder Zeichensatz Unicode verwendet, undhierdiespeichereffiziente Variante UTF-8. In MariaDB und MySQL kann man z.B. mit

```
CREATE DATABASE datenbank DEFAULT CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

eine gesamte Datenbank so konfigurieren, dass UTF-8 als Zeichensatz und eine *case insensitive* Sortierung eingerichtet wird. In PostgreSQL lautet der entsprechende Befehl

```
CREATE DATABASE datenbank ENCODING= 'UTF8'LC_COLLATE = 'und-x-icu'
```

Hier können auch länderspezifische Kodierungen festgelegt werden, z.B. für einen deutschen Schriftsatz mit der Option LC