

SQL (Structured Query Language)



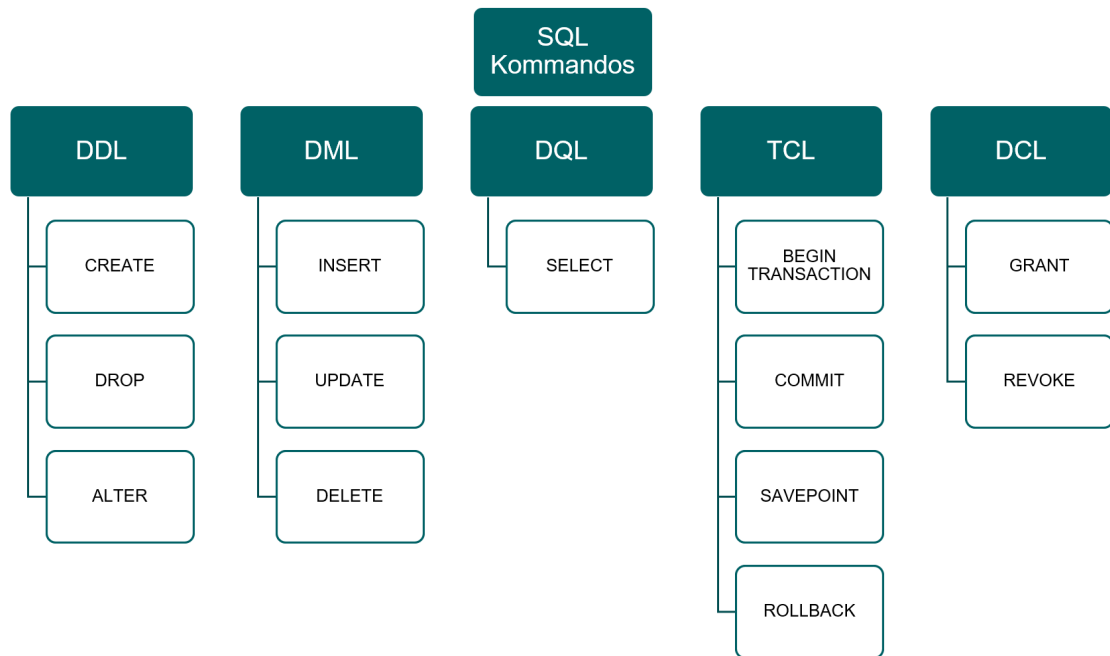
Relationenalgebra und Relationenkalkül sind formale Spezialsprachen für Anfragen. SQL wurde als praxisnähere Sprache 1974 im IBM Almaden Research Centre entwickelt. SQL ist eine Mischung von relationaler Algebra und relationalem Kalkül, plus Zusätze (z.B. arithmetische Operatoren oder Aggregatfunktionen). **SQL Tabellen sind Multimengen von Tupeln – d.h. diese können Duplikate enthalten. Siehe auch das Schlüsselwort DISTINCT.**

Teilsprachen von SQL

Die Teilsprachen von SQL, die wir behandeln werden, sind:

- **Data Definition Language (DDL):** Schemadefinitionen.
- **Data Manipulation Language (DML):** Einfügen, Ändern, Löschen und Anfragen von Daten.
- **Data Query Language (DQL):** Abfragen.
- **Transaction Control Language (TCL):** Transaktionsbehandlung.
- **Data Control Language (DCL):** Rechtebehandlung.

Wir werden uns auf diese Kommandos konzentrieren.



SQL Historie

Jahr	Name	Alias	Kommentare
1986	SQL-86	SQL-87	Erste Formalisierung durch ANSI
1989	SQL-89	FIPS 127-1	Geringfügige Überarbeitung, die Integritätsbeschränkungen hinzufügte, und als FIPS 127-1 (Federal Information Processing Standard) übernommen wurde
1992	SQL-92	SQL2, FIPS 127-2	Hauptüberarbeitung (ISO 9075), Entry Level SQL-92 als FIPS 127-2 übernommen
1999	SQL: 1999	SQL3	Hinzufügen von regulären Ausdrücken, rekursiven Abfragen (z. B. Transitivität), Trigger, Unterstützung für prozedurale und Kontrollflussanweisungen, nichtskalare Typen (Arrays) und einige objektorientierte Features (z. B. strukturierte Typen), Unterstützung für die Einbettung von SQL in Java (SQL/OLB) und umgekehrt (SQL/JRT)
2003	SQL: 2003		Einführung von XML-Funktionen (SQL/XML), Fensterfunktionen, standardisierte Sequenzen und Spalten mit automatisch generierten Werten (einschließlich Identitätsspalten)
2006	SQL: 2006		ISO/IEC 9075-14:2006 definiert die Verwendung von SQL mit XML. Import und Speicherung von XML-Daten in einer SQL-Datenbank, deren Verarbeitung innerhalb der Datenbank und die Veröffentlichung von XML- und herkömmlichen SQL-Daten in XML-Form. Darüber hinaus können Anwendungen Abfragen mit XQuery, der XML-Abfragesprache des World Wide Web Consortium (W3C), in ihren SQL-Code integrieren, um gleichzeitig auf normale SQL-Daten und XML-Dokumente zuzugreifen.

Jahr	Name	Alias	Kommentare
2008	SQL: 2008		Legalisiert ORDER BY außerhalb von Cursordefinitionen. Fügt INSTEAD OF-Triggers, TRUNCATE-Anweisung und FETCH-Klausel hinzu
2011	SQL: 2011		Fügt zeitbezogene Daten (PERIOD FOR) hinzu. Verbesserungen für Fensterfunktionen und FETCH-Klausel.
2016	SQL: 2016		Fügt Zeilenmusterabgleich, polymorphe Tabellenfunktionen, JSON hinzu
2019	SQL: 2019		Fügt Teil 15, mehrdimensionale Arrays (MDarray-Typ und Operatoren) hinzu

Achtung!

Für die Übungsblätter und die Klausuren wird nur standard-konformes SQL verwendet und erwartet, so wie es in dieser Vorlesung vorgestellt wird!

Nicht verwendet, benötigt und zulässig sind SQL-Erweiterungen wie z.B. MySQL, MS SQL, SAP, oder von anderen Herstellern.

SQL Wertebereiche

Die von SQL unterstützten Wertebereiche (domains) sind endliche Unterbereiche der ganzen und der reellen Zahlen, sowie andere Datentypen, um verschiedene Arten von Daten in der Datenbank zu speichern. Hier sind einige der häufig verwendeten und sinnvollen Standardwertebereiche:

Numerische Typen

- **NUMBER** : Ein numerischer Wert mit variabler Präzision und Skala.
- **DECIMAL(p,s)** : Ein numerischer Wert mit einer festen Anzahl von Nachkommastellen, wobei *p* die Gesamtzahl der Ziffern und *s* die Anzahl der Nachkommastellen angibt.
- **INTEGER** : Ein ganzzahliger Wert.
- **SMALLINT** : Ein kleiner ganzzahliger Wert, der weniger Speicherplatz als ein **INTEGER** benötigt.
- **FLOAT** : Ein numerischer Wert mit Gleitkommadarstellung.
- **REAL** : Ein numerischer Wert mit Gleitkommadarstellung und weniger Genauigkeit als **FLOAT**.
- **DOUBLE PRECISION** : Ein numerischer Wert mit Gleitkommadarstellung und höherer Genauigkeit als **FLOAT**.

Zeichenketten-Typen

- **CHAR(n)** : Eine Zeichenkette fester Länge, wobei *n* die Anzahl der Zeichen angibt.
- **VARCHAR(n)** : Eine Zeichenkette variabler Länge, wobei *n* die maximale Anzahl der

Zeichen angibt.

Weitere Typen

- DATE : Ein Datumswert, der Jahr, Monat und Tag enthält.
- BOOLEAN : Ein Wahrheitswert, der entweder TRUE , FALSE oder NULL sein kann.
- CURRENCY : Ein Währungswert, der zur Speicherung von Geldbeträgen verwendet wird.
- ... (Es gibt weitere spezialisierte Datentypen je nach Datenbankmanagementsystem.)

Jeder dieser Wertebereiche kann verwendet werden, um die Art der Daten, die in einer Spalte gespeichert werden, genauer zu definieren. Die Wahl des richtigen Wertebereichs ist wichtig, um die Integrität der Daten und die Effizienz der Datenbank zu gewährleisten. SQL bietet Möglichkeiten bei Bedarf eigene Wertebereiche zu definieren, die wir aber in der Vorlesung nicht behandeln.

Die Beispieldatenbank aus der ER Modellierung

Das folgende Schema haben wir in der ER Modellierung schon entwickelt und kennengelernt:

Relationenschemata (ohne Wertebereiche):

- Student(MatrNr, Name, Semester)
- Professor(PersNr, Name, Rang, Raum)
- Vorlesung(VorlNr, Titel, SWS, gelesenVon)
- Assistent(PersNr, Name, Fachgebiet, Boss)
- Voraussetzen(Vorgänger, Nachfolger)
- Hört(MatrNr, VorlNr)
- Prüft(MatrNr, VorlNr, PersNr, Note)

Interrelationale Abhängigkeiten:

- Vorlesung[gelesenVon] $\subseteq$ Professor[PersNr]
- Assistent[Boss] $\subseteq$ Professor[PersNr]
- Voraussetzen[Vorgänger] $\subseteq$ Vorlesung[VorlNr]
- Voraussetzen[Nachfolger] $\subseteq$ Vorlesung[VorlNr]
- Hört[MatrNr] $\subseteq$ Student[MatrNr]
- Hört[VorlNr] $\subseteq$ Vorlesung[VorlNr]
- Prüft[MatrNr] $\subseteq$ Student[MatrNr]
- Prüft[VorlNr] $\subseteq$ Vorlesung[VorlNr]
- Prüft[PersNr] $\subseteq$ Professor[PersNr]
- Assistent[PersNr] $\cap$ Professor[PersNr] = $\emptyset$

Beispiel für eine Datenbank

Professor			
PersNr	Name	Rang	Raum
2125	Sokrates	W3	226
2126	Russel	W3	232
2127	Kopernikus	W2	310
2133	Popper	W2	52
2134	Augustinus	W2	309
2136	Curie	W3	36
2137	Kant	W3	7

Student		
MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Vorlesung			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

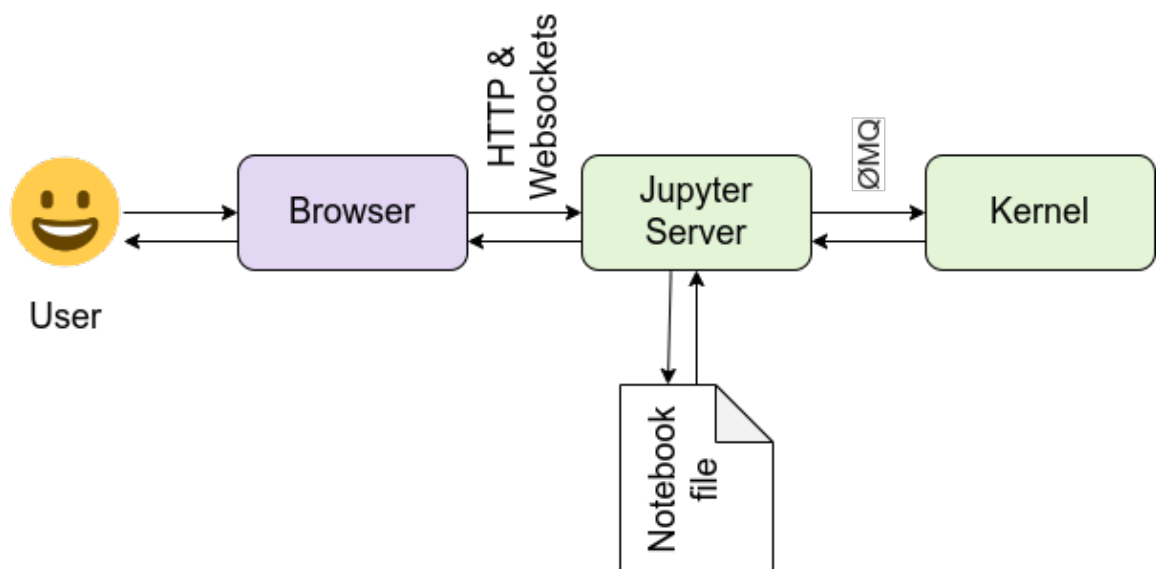
hört			
MatrNr	VorlNr		
26120	5001		
27550	5001		
27550	4052		
28106	5041		
28106	5052		
28106	5216		
28106	5259		
29120	5001		
29120	5041		
29120	5049		
29555	5022		
25403	5022		

voraussetzen	
Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

Assistent			
PerslNr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2126

prüft			
MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1,0
25403	5041	2125	2,0

Python und SQL



<https://docs.jupyter.org/en/latest/projects/architecture/content-architecture.html>

"IPython will treat any line whose first character is a % as a special call to a 'magic' function. These allow you to control the behavior of IPython itself, plus a lot of system-type features. They are all prefixed with a % character, but parameters are given without parentheses or quote." (<https://ipython.readthedocs.io/en/stable/interactive/reference.html#magic>)

`%load_ext sql` ruft die Funktion `load_extension` auf. Diese lädt das Erweiterungsmodul welches die Funktion `%sql` realisiert. Die iPython-Dokumentation erklärt, wie man Magics implementiert.

`%sql sqlite:///beispieldb` verbindet die SQLITE engine zum Python Kernel and macht die Datenbank mit dem Namen `beispieldb` zugänglich. Unter anderem mit `%%sql` können dann in Code-Zellen SQL-Kommandos ausgeführt werden. In den Übung werden wir kann es vorkommen, dass wir die magic-Notation nicht verwenden, da die SQL statements automatisch auf Korrektheit geprüft werden sollen und daher als Python-Variableninhalt benötigt werden.

In []:

```
In [1]: # Set-up
%load_ext sql
%sql sqlite:///SQL-DDL-db.db
```

```
In [2]: %%sql
DROP TABLE if EXISTS Student;
DROP TABLE if EXISTS Professor;
DROP TABLE if EXISTS Vorlesung;
DROP TABLE if EXISTS Prüft;
```

```
* sqlite:///SQL-DDL-db.db
```

```
Done.
```

```
Done.
```

```
Done.
```

```
Done.
```

```
Out[2]: []
```

SQL Data Definition Language (DDL)

In diesem Jupyter Notebook werden wir die Data Definition Language (DDL) von SQL besprechen. DDL ist ein Teil von SQL und wird verwendet, um die Struktur von Datenbanktabellen und -beziehungen zu definieren.

Es gibt drei Haupt-DDL-Anweisungen:

1. CREATE TABLE: zum Erstellen einer neuen Tabelle
2. ALTER TABLE: zum Ändern einer bestehenden Tabelle
3. DROP TABLE: zum Löschen einer Tabelle

Wir werden das folgende Schema als Beispiel verwenden:

Eine Relation anlegen: CREATE TABLE

CREATE TABLE ist eine SQL-Anweisung, die zum Erstellen einer neuen Tabelle in einer Datenbank verwendet wird. Mit dieser Anweisung können Sie die Struktur der Tabelle, ihre Spalten und ihre Datentypen, sowie Primär- und Fremdschlüsselbeziehungen angeben.

Syntax

Die grundlegende Syntax für die CREATE TABLE -Anweisung lautet wie folgt:

```
CREATE TABLE table_name (
    column_name1 data_type1 constraints1,
    column_name2 data_type2 constraints2,
    ...
    column_nameN data_typeN constraintsN
```

```
);
```

Dabei sind:

- **'table_name'**: Der Name der Tabelle, die erstellt wird.
- **'column_name'**: Der Name der Spalte in der Tabelle.
- **'data_type'**: Der Datentyp der Spalte, z.B. INTEGER, VARCHAR, DATE, etc.
- **'constraints'**: Optional, zusätzliche Einschränkungen oder Regeln, die für die Spalte gelten, z.B. NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, etc.

Beispielrelation: Student(MatrNr, Name, Semester)

```
In [3]: %%sql
CREATE TABLE Student
( MatrNr INTEGER PRIMARY KEY,
  Name VARCHAR(255),
  Semester INTEGER
);
```

```
* sqlite:///SQL-DDL-db.db
```

Done.

```
Out[3]: []
```

Jetzt sind Sie dran: Angenommen wir wollen einen Daten über Professoren, wie zum Beispiel den W3-Professor Stefan Decker mit der Personalnummer 1001 und der Büronummer 1234 zu der Datenbank hinzufügen.

Erstellen Sie dafür eine Tabelle für das folgende Schema:

- Professor(PersNr, Name, Rang, Raum)

```
In [4]: %%sql
CREATE TABLE Professor
( PersNr INTEGER PRIMARY KEY,
  Name VARCHAR(255),
  Rang VARCHAR(255),
  Raum Integer
);
```

```
* sqlite:///SQL-DDL-db.db
```

Done.

```
Out[4]: []
```

Optionen für PRIMARY KEYS

Die PRIMARY KEY Einschränkung ist äquivalent zu „NOT NULL UNIQUE“

Verschiedene Möglichkeiten zur Festlegung eines Primary Keys

1. Mit einer separaten Zeile für die PRIMARY KEY Definition - auch für zusammengesetzte Schlüssel:

```
CREATE TABLE Student
( MatrNr INTEGER,
  Name VARCHAR(255),
  Semester INTEGER,
  PRIMARY KEY (MatrNr)
);
```

2. Mit der PRIMARY KEY Definition direkt neben dem Attribut:

```
CREATE TABLE Student
( MatrNr INTEGER PRIMARY KEY,
  Name VARCHAR(255),
  Semester INTEGER
);
```

3. Mit einem benannten CONSTRAINT für die PRIMARY KEY Definition:

```
CREATE TABLE Student
( MatrNr INTEGER,
  Name VARCHAR(255),
  Semester INTEGER,
  CONSTRAINT pk_student PRIMARY KEY (MatrNr)
);
```

Foreign Key

Ein **Fremdschlüssel** (Foreign Key) ist eine Spalte oder eine Gruppe von Spalten in einer Tabelle, die sich auf den **Primärschlüssel einer anderen Tabelle** beziehen. Fremdschlüssel werden verwendet, um Beziehungen zwischen Tabellen herzustellen und die referentielle Integrität der Daten sicherzustellen.

Beispiel: Vorlesung Relation

Im Folgenden finden Sie ein Beispiel, das zeigt, wie Fremdschlüssel verwendet werden, um eine Beziehung zwischen den Tabellen `Professor` und `Vorlesung` herzustellen. Wir wollen Daten wie eine Vorlesung "DBIS", die die Nummer 2413 hat, und von einem Professor mit der Personalnummer 1001 gelesen wird, in die Datenbank eintragen. Dazu müssen wir das folgende Schema der Vorlesungsrelation als SQL Tabelle realisieren:

- Vorlesung(VorlNr, Titel, SWS, gelesenVon)

und wir haben die folgende interrelationale Abhängigkeit:

- Vorlesung[gelesenVon] $\subseteq$ Professor[PersNr]

```
In [5]: %%sql
CREATE TABLE Vorlesung (
  VorlNr INTEGER PRIMARY KEY,
  Titel VARCHAR(255),
```

```
SWS INTEGER,
gelesenVon INTEGER,
FOREIGN KEY (gelesenVon) REFERENCES Professor(PersNr)
);
```

```
* sqlite:///SQL-DDL-db.db
```

```
Done.
```

```
Out[5]: []
```

In diesem Beispiel haben wir die folgende Tabelle:

- **Vorlesung:** Eine Tabelle, die Informationen über Vorlesungen enthält. Die Tabelle hat einen Primärschlüssel `VorlNr`, der die eindeutige Identifikationsnummer einer Vorlesung darstellt.

Die `Vorlesung` -Tabelle hat eine Spalte `gelesenVon`, die angibt, welcher Professor die Vorlesung hält. Um eine Beziehung zwischen der `Vorlesung` -Tabelle und der `Professor` -Tabelle herzustellen, fügen wir der `CREATE TABLE` -Anweisung für die `Vorlesung` -Tabelle die Zeile `FOREIGN KEY (gelesenVon) REFERENCES Professor(PersNr)` hinzu. Dies bedeutet, dass der Wert in der Spalte `gelesenVon` der `Vorlesung` -Tabelle auf den Primärschlüssel `PersNr` in der `Professor` -Tabelle verweisen muss.

Durch die Verwendung des Fremdschlüssels können wir sicherstellen, dass für jede Vorlesung in der `Vorlesung` -Tabelle immer ein gültiger Professor in der `Professor` -Tabelle existiert, der die Vorlesung hält. Dadurch wird die referentielle Integrität der Daten gewahrt.

Wichtig: **FOREIGN KEY** impliziert NICHT **PRIMARY KEY**.

Jetzt sind Sie wieder dran:

Die `Prüft` Tabelle hat das folgende Schema:

- `Prüft( MatrNr, VorlNr, PersNr, Note )`

sowie die folgenden interrelationalen Abhängigkeiten.

- `Prüft[MatrNr]  $\subseteq$  Student[MatrNr]`
- `Prüft[VorlNr]  $\subseteq$  Vorlesung[VorlNr]`
- `Prüft[PersNr]  $\subseteq$  Professor[PersNr]`

Wie sieht die SQL Tabellendefinition aus?

```
In [6]: %%sql
```

```
CREATE TABLE Prüft (
    MatrNr Integer,
    VorlNr INTEGER,
    PersNr Integer,
    Note VARCHAR (255),
    PRIMARY KEY(MatrNr,VorlNr,PersNr),
```

```

FOREIGN KEY (MatrNr) REFERENCES Student(MatNr),
FOREIGN KEY (VorlNr) REFERENCES Vorlesung(VorlNr),
FOREIGN KEY (PersNr) REFERENCES Professor(PersNr)
);

```

```

* sqlite:///SQL-DDL-db.db
(sqlite3.OperationalError) table Prüft already exists
[SQL: CREATE TABLE Prüft (
    MatrNr Integer,
    VorlNr INTEGER,
    PersNr Integer,
    Note VARCHAR (255),
    PRIMARY KEY(MatNr,VorlNr,PersNr),
    FOREIGN KEY (MatrNr) REFERENCES Student(MatNr),
    FOREIGN KEY (VorlNr) REFERENCES Vorlesung(VorlNr),
    FOREIGN KEY (PersNr) REFERENCES Professor(PersNr)
);]
(Background on this error at: https://sqlalche.me/e/20/e3q8)

```

Anmerkung: Es kann sein dass viele real existierende Legacy-Systeme mit Nicht-ASCII Zeichen in Tabellennamen nicht umgehen können!

Relationen ändern und löschen: ALTER TABLE und DROP TABLE

`ALTER TABLE` ist ein SQL-Befehl, der zum Ändern der Struktur einer vorhandenen Tabelle verwendet wird. Mit `ALTER TABLE` können Sie Spalten hinzufügen, löschen oder ändern sowie Einschränkungen hinzufügen oder entfernen.

ALTER TABLE

Syntax zum Hinzufügen Und Löschen einer Spalte:

```

ALTER TABLE table_name
ADD column_name data_type [constraint];

```

```

ALTER TABLE table_name
DROP COLUMN column_name;

```

Syntax zum Ändern einer Spalte:

```

ALTER TABLE table_name
ALTER COLUMN column_name new_data_type;

```

Syntax zum Hinzufügen und Löschen einer Einschränkung:

```

ALTER TABLE table_name
ADD CONSTRAINT constraint_name constraint_type (column_name);

```

```

ALTER TABLE table_name
DROP CONSTRAINT constraint_name;

```

DROP TABLE

DROP TABLE ist ein SQL-Befehl, der zum Löschen einer vorhandenen Tabelle aus der Datenbank verwendet wird. Wenn Sie eine Tabelle löschen, werden alle Daten in der Tabelle gelöscht und die Tabellenstruktur entfernt.

DROP TABLE table_name;

Beispiel:

DROP TABLE Student;

Dieses Beispiel löscht die Student-Tabelle und alle darin enthaltenen Daten. Die IF EXISTS-Bedingung im DROP TABLE-Statement ist eine optionale Klausel, die verwendet wird, um sicherzustellen, dass keine Fehler auftreten, wenn die angegebene Tabelle nicht existiert. Wenn die IF EXISTS-Klausel verwendet wird, prüft das Datenbanksystem, ob die angegebene Tabelle existiert. Wenn die Tabelle existiert, wird sie gelöscht. Wenn die Tabelle jedoch nicht existiert, wird die Anweisung einfach ignoriert und es wird keine Fehlermeldung angezeigt. Dies kann nützlich sein, wenn Sie Skripte ausführen, bei denen die Existenz einer Tabelle nicht garantiert ist. Beispiel:

DROP TABLE IF EXISTS Student;

In diesem Beispiel wird die Tabelle Student gelöscht, wenn sie existiert. Wenn sie nicht existiert, wird keine Fehlermeldung angezeigt und das Skript wird fortgesetzt. Mit diesen Befehlen können Sie die Struktur und Daten Ihrer Datenbank verwalten und an neue Anforderungen anpassen. Achten Sie darauf, dass das Löschen von Spalten oder das Ändern von Datentypen bestehende Daten verlieren oder beschädigen kann.

Sichten anlegen: CREATE VIEW

In relationalen Systemen werden Sichten als abgeleitete Relationen aufgefasst, die über Anfragen definiert sind. Hier überlappen sich die Data Definition Language und die Data Query Language von SQL.

Syntax:

CREATE VIEW <Sichtname> [(<Attributname>{, <Attributname>})] **AS**
<Subquery>

***Beispiel**

CREATE VIEW Langzeit_Studierende **AS**
 SELECT * FROM Student **WHERE** Semester **>=** 10;

In diesem Beispiel erstellen wir eine Sicht namens Langzeit_Studierende, die alle Studenten enthält, deren 'Semester' größer oder gleich 10 ist. Wie genau das SELECT -Statement aussieht werden wir bei der DQL besprechen.

Sichten löschen

DROP VIEW <Sicht-Name>;

Sichten ausführen Jede Anfrage auf eine Sicht führt diese auch aus. Sichten können in Abfragen verwendet werden, als ob sie normale Tabellen wären.

Indexe in SQL

Ein Index in SQL ist eine Datenbankstruktur, die die Suche nach Daten in einer Tabelle effizienter gestaltet. Indexe werden verwendet, um die Abfrageleistung zu verbessern, insbesondere bei komplexen und großen Datenmengen. Je nach DBMS und Datentyp unterschiedliche Realisierung, Häufig durch B+ Bäume realisiert (Details dazu in später in der Vorlesung).

Index erstellen

Um einen Index zu erstellen, verwenden Sie das `CREATE INDEX`-Statement. Die Syntax zum Erstellen eines Index ist:

```
CREATE [UNIQUE] INDEX <Indexname>  
ON <Relationenname> (<Attributname> [<Ordnung>] {, <Attributname>  
[<Ordnung>] })
```

Dabei:

- `[UNIQUE]` : Optional. Wenn `UNIQUE` angegeben ist, wird ein eindeutiger Index erstellt, der sicherstellt, dass keine zwei Zeilen denselben Indexwert haben.
- `<Indexname>` : Der Name des Index, den Sie erstellen möchten.
- `<Relationenname>` : Der Name der Tabelle, für die der Index erstellt werden soll.
- `<Attributname>` : Der Name der Spalte, auf der der Index basieren soll. Sie können mehrere Spalten angeben, um einen zusammengesetzten Index zu erstellen.
- `[<Ordnung>]` : Optional. Gibt die Sortierreihenfolge für den Index an. `ASC` für aufsteigende Sortierung (Standard) und `DESC` für absteigende Sortierung.

Beispiel

```
CREATE UNIQUE INDEX idx_student_semester  
ON Student (Semester ASC);
```

In diesem Beispiel erstellen wir einen Index namens `idx_student_semester` auf der `Student`-Tabelle, der auf der Spalte `Semester` basiert. Dies kann dazu beitragen, die Abfrageleistung zu verbessern, wenn nach Studenten in einem bestimmten Semester gesucht wird.

Index löschen

Um einen Index zu löschen, verwenden Sie das `DROP INDEX`-Statement. Die Syntax zum Löschen eines Index ist:

```
DROP INDEX index_name;
```

Beispiel:

```
DROP INDEX idx_student_matnr;
```

Index-Typen Es gibt verschiedene Typen von Indexen, die in SQL verwendet werden können:

- Unique Index: Ein Unique Index stellt sicher, dass keine zwei Zeilen in der Tabelle denselben Indexwert haben. Dies ist hilfreich, um die Eindeutigkeit der Daten sicherzustellen.
- Clustered Index: Ein Clustered Index bestimmt die physikalische Reihenfolge der Daten in einer Tabelle. Jede Tabelle kann nur einen Clustered Index haben.
- Non-Clustered Index: Ein Non-Clustered Index ist ein Index, der unabhängig von der physikalischen Reihenfolge der Daten in der Tabelle ist. Eine Tabelle kann mehrere Non-Clustered Indexe haben.

Die genaue Syntax hängt von der verwendeten Datenbank ab. Beachten Sie, dass die Verwendung von Indexen zwar die Abfrageleistung verbessern kann, aber auch einige Nachteile hat. Das Hinzufügen von Indexen kann den Speicherplatzbedarf erhöhen und die Leistung von Schreibvorgängen (Einfügen, Aktualisieren, Löschen) beeinträchtigen. Daher ist es wichtig, den richtigen Index für die jeweilige Anwendung sorgfältig auszuwählen.

In [7]:

`%%sql``PRAGMA table_list``* sqlite:///SQL-DDL-db.db`

Done.

Out[7]:

schema	name	type	ncol	wr	strict
main	Prüft	table	4	0	0
main	Professor	table	4	0	0
main	Student	table	3	0	0
main	sqlite_schema	table	5	0	0
temp	sqlite_temp_schema	table	5	0	0

In []:

```
In [1]: # Set-up
%load_ext sql
%sql sqlite:///SQL-DML-db.db
```

```
In [2]: %%sql
DROP TABLE IF EXISTS Student;
DROP TABLE IF EXISTS Vorlesung;
DROP TABLE IF EXISTS Professor;

CREATE TABLE Student (
    MatrNr INTEGER PRIMARY KEY,
    Name VARCHAR(255),
    Semester INTEGER
);

CREATE TABLE Professor (
    PersNr INTEGER PRIMARY KEY,
    Name VARCHAR(255),
    Rang VARCHAR(255),
    Raum INTEGER
);

CREATE TABLE Vorlesung (
    VorlNr INTEGER PRIMARY KEY,
    Titel VARCHAR(255),
    SWS INTEGER,
    gelesenVon INTEGER,
    FOREIGN KEY (gelesenVon) REFERENCES Professor(PersNr)
);
-- SQLite does not enforce foreign key constraints by default - the following PRAGMA
PRAGMA foreign_keys = 1
```

```
* sqlite:///SQL-DML-db.db
```

```
Done.
```

```
Done.
```

```
Done.
```

```
Done.
```

```
Done.
```

```
Done.
```

```
Done.
```

```
Out[2]: []
```

SQL Data Manipulation Language (DML)

Wir lernen nun die grundlegenden SQL-DML-Befehle kennen, die zum Einfügen, Aktualisieren und Löschen von Daten in einer relationalen Datenbank verwendet werden. Die vier Haupt-DML-Befehle sind:

1. INSERT
2. UPDATE

3. DELETE

Manchmal wird SELECT ebenfalls als Teil DML angenommen - wir besprechen SELECT aber separat als Data Query Language.

INSERT

Mit dem INSERT-Befehl können wir neue Zeilen (Tupel) in eine Tabelle einfügen. Die Syntax lautet:

```
INSERT INTO <tabelle> (spalte1, spalte2, ...) VALUES (wert1, wert2, ...);
```

Beispiel: Angenommen, wir haben eine Tabelle "Studenten" mit den Spalten "MatNr", "Name" und "Semester". Um einen neuen Studenten einzufügen, verwenden wir den folgenden Befehl:

```
In [4]: %%sql
INSERT INTO Student (MatrNr, Name, Semester) VALUES (4711, 'Max Mustermann', 3);

* sqlite:///SQL-DML-db.db
1 rows affected.
```

```
Out[4]: []
```

Was passiert wenn wir Daten eintragen ohne darüber nachzudenken?

```
In [5]: %%sql
INSERT INTO Vorlesung(VorlNr, Titel, SWS, gelesenVon)
VALUES (2413, "DBIS", 4, 1001)

* sqlite:///SQL-DML-db.db
(sqlite3.IntegrityError) FOREIGN KEY constraint failed
[SQL: INSERT INTO Vorlesung(VorlNr, Titel, SWS, gelesenVon)
VALUES (2413, "DBIS", 4, 1001)]
(Background on this error at: https://sqlalche.me/e/20/gkpj)
```

Was bedeutet der Fehler? Tragen wir zur zunächst den Professor mit der Personalnummer 1001 ein und versuchen es nochmal.

```
In [6]: %%sql

INSERT INTO Professor(PersNr, Name, Rang, Raum)
VALUES (1001, "Stefan Decker", "W3", 1234)

* sqlite:///SQL-DML-db.db
1 rows affected.
```

```
Out[6]: []
```

```
In [7]: %%sql
INSERT INTO Vorlesung(VorlNr, Titel, SWS, gelesenVon)
VALUES (2413, "DBIS", 4, 1001)

* sqlite:///SQL-DML-db.db
1 rows affected.
```

Out[7]: []

```
In [8]: %%sql
INSERT INTO Vorlesung(VorlNr, Titel, SWS, gelesenVon)
VALUES (2414, "MALO", 4, 1001)

* sqlite:///SQL-DML-db.db
1 rows affected.
```

Out[8]: []

UPDATE

Mit dem UPDATE-Befehl können wir vorhandene Daten in einer Tabelle ändern. Die Syntax lautet:

```
UPDATE <tabelle> SET spalte1 = wert1, spalte2 = wert2, ... WHERE
<bedingung>;
```

Bemerkung: Was als Bedingung in der WHERE Klausel stehen kann wird bei der DQL ausführlicher besprochen.

Beispiel: Um das Semester eines Studenten mit der MatrNr 4711 auf 4 zu ändern, verwenden wir den folgenden Befehl:

```
In [9]: %%sql
UPDATE Student SET Semester = Semester + 1 WHERE MatrNr = 4711;

* sqlite:///SQL-DML-db.db
1 rows affected.
```

Out[9]: []

DELETE

Mit dem DELETE-Befehl können wir Zeilen aus einer Tabelle löschen. Die Syntax lautet:

```
DELETE FROM <tabelle> WHERE <bedingung>;
```

Beispiel: Lösche das Tupel von der Tabelle Student bei dem die Matrikelnummer den Wert 4711 hat. Die möglichen Bedingungen werden bei der DQL ausführlich besprochen.

```
In [10]: %%sql
DELETE FROM Student WHERE MatrNr = 4711;

* sqlite:///SQL-DML-db.db
1 rows affected.
```

Out[10]: []

Was passiert wenn der eingetragene Professor gelöscht wird?

```
In [11]: %%sql
```

```
DELETE FROM Professor WHERE PersNr = 1001;
```

```
* sqlite:///SQL-DML-db.db  
(sqlite3.IntegrityError) FOREIGN KEY constraint failed  
[SQL: DELETE FROM Professor WHERE PersNr = 1001;]  
(Background on this error at: https://sqlalche.me/e/20/gkpj)
```

Was müssen wir tun um das zu beseitigen?

```
In [12]: %%sql  
DELETE FROM Vorlesung WHERE VorlNr = 2413 OR VorlNr = 2414;  
  
* sqlite:///SQL-DML-db.db  
2 rows affected.
```

Out[12]: []

Dann versuchen wir es nochmal:

```
In [13]: %%sql  
DELETE FROM Professor WHERE PersNr = 1001;  
  
* sqlite:///SQL-DML-db.db  
1 rows affected.
```

Out[13]: []

Wenn Sie in den Übungen Fehlersituationen haben, achten Sie bitte darauf dass Sie diese auch beheben!

Das sind die grundlegenden SQL-DML-Befehle ohne SELECT, mit denen Sie Daten in einer relationalen Datenbank verwalten können.

In []:

```
In [1]: # Set-up
%reload_ext sql
%sql sqlite:///SQL-Beispieldb.db
```

Einführung in die SQL Data Query Language (DQL)



Eine SQL SELECT-Abfrage ist eine Anweisung, die zum Abrufen von Daten aus einer oder mehreren Tabellen in einer relationalen Datenbank verwendet wird. Mit der SELECT-Abfrage können Sie bestimmte Spalten auswählen, Bedingungen festlegen, um Zeilen zu filtern, Daten aus mehreren Tabellen verknüpfen, Daten sortieren und gruppieren sowie Aggregatfunktionen anwenden. Hier ist die grundlegende Syntax einer SELECT-Abfrage:

```
SELECT column1, column2, ...
FROM table_name
WHERE conditions
GROUP BY column(s)
HAVING conditions
ORDER BY column(s) ASC|DESC
```

Wir besprechen zunächst die ersten drei Komponenten. Diese entsprechen der relationalen Algebra wie folgt:

SQL Statement	Argumente	Relationale Algebra
SELECT	<Liste von Attributsnamen>	Projektion: π

SQL Statement	Argumente	Relationale Algebra
FROM	<Liste von Tabellen>	Kreuzprodukt: $\times$
WHERE (optional)	<Bedingung>	Selektion: σ

Wenn ein Attribut (z.B. A) in zwei Relationen vorkommt (z.B. in R und S), muss man in SQL den Relationsnamen als „Tupelvariable“ angeben:

```
SELECT R.A, S.A, C, ...
FROM R, S, T, ...
WHERE Bedingung
```

Das Schlüsselwort DISTINCT ist in SQL-Anfragen wichtig, um Duplikate aus den Ergebnissen zu entfernen. Es wird in Kombination mit SELECT verwendet und stellt sicher, dass jede Zeile im Ergebnisse einzigartig ist.

SELECT

Beispiele: Alle Studierenden aus der Datenbank mit allen Attributen:

In [2]:

```
%%sql

SELECT
    MatrNr, Name, Semester
from
    Student
```

* sqlite:///SQL-Beispieldb.db
Done.

Out[2]:

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Alle Attribute einer Tabelle kann man auch vereinfacht mit einem '*' erhalten.

In [3]:

```
%%sql

SELECT
    *
```

```
from
  Student
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[3]:
```

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

```
In [4]: %%sql
```

```
SELECT
  *
from
  Professor
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[4]:
```

PersNr	Name	Rang	Raum
2125	Sokrates	W3	226
2126	Russel	W3	232
2127	Kopernikus	W2	310
2133	Popper	W2	52
2134	Augustinus	W2	309
2136	Curie	W3	36
2137	Kant	W3	7

Die Anfrage mit mehreren Tabellen resultiert in der Ausgabe aller Attribute des Kreuzproduktes

```
In [5]: %%sql
```

```
SELECT
  *
from
  Student,Professor
```

```
* sqlite:///SQL-Beispieldb.db  
Done.
```

Out[5]:	MatrNr	Name	Semester	PersNr	Name_1	Rang	Raum
	24002	Xenokrates	18	2125	Sokrates	W3	226
	24002	Xenokrates	18	2126	Russel	W3	232
	24002	Xenokrates	18	2127	Kopernikus	W2	310
	24002	Xenokrates	18	2133	Popper	W2	52
	24002	Xenokrates	18	2134	Augustinus	W2	309
	24002	Xenokrates	18	2136	Curie	W3	36
	24002	Xenokrates	18	2137	Kant	W3	7
	25403	Jonas	12	2125	Sokrates	W3	226
	25403	Jonas	12	2126	Russel	W3	232
	25403	Jonas	12	2127	Kopernikus	W2	310
	25403	Jonas	12	2133	Popper	W2	52
	25403	Jonas	12	2134	Augustinus	W2	309
	25403	Jonas	12	2136	Curie	W3	36
	25403	Jonas	12	2137	Kant	W3	7
	26120	Fichte	10	2125	Sokrates	W3	226
	26120	Fichte	10	2126	Russel	W3	232
	26120	Fichte	10	2127	Kopernikus	W2	310
	26120	Fichte	10	2133	Popper	W2	52
	26120	Fichte	10	2134	Augustinus	W2	309
	26120	Fichte	10	2136	Curie	W3	36
	26120	Fichte	10	2137	Kant	W3	7
	26830	Aristoxenos	8	2125	Sokrates	W3	226
	26830	Aristoxenos	8	2126	Russel	W3	232
	26830	Aristoxenos	8	2127	Kopernikus	W2	310
	26830	Aristoxenos	8	2133	Popper	W2	52
	26830	Aristoxenos	8	2134	Augustinus	W2	309
	26830	Aristoxenos	8	2136	Curie	W3	36
	26830	Aristoxenos	8	2137	Kant	W3	7
	27550	Schopenhauer	6	2125	Sokrates	W3	226
	27550	Schopenhauer	6	2126	Russel	W3	232

MatrNr	Name	Semester	PersNr	Name_1	Rang	Raum
27550	Schopenhauer	6	2127	Kopernikus	W2	310
27550	Schopenhauer	6	2133	Popper	W2	52
27550	Schopenhauer	6	2134	Augustinus	W2	309
27550	Schopenhauer	6	2136	Curie	W3	36
27550	Schopenhauer	6	2137	Kant	W3	7
28106	Carnap	3	2125	Sokrates	W3	226
28106	Carnap	3	2126	Russel	W3	232
28106	Carnap	3	2127	Kopernikus	W2	310
28106	Carnap	3	2133	Popper	W2	52
28106	Carnap	3	2134	Augustinus	W2	309
28106	Carnap	3	2136	Curie	W3	36
28106	Carnap	3	2137	Kant	W3	7
29120	Theophrastos	2	2125	Sokrates	W3	226
29120	Theophrastos	2	2126	Russel	W3	232
29120	Theophrastos	2	2127	Kopernikus	W2	310
29120	Theophrastos	2	2133	Popper	W2	52
29120	Theophrastos	2	2134	Augustinus	W2	309
29120	Theophrastos	2	2136	Curie	W3	36
29120	Theophrastos	2	2137	Kant	W3	7
29555	Feuerbach	2	2125	Sokrates	W3	226
29555	Feuerbach	2	2126	Russel	W3	232
29555	Feuerbach	2	2127	Kopernikus	W2	310
29555	Feuerbach	2	2133	Popper	W2	52
29555	Feuerbach	2	2134	Augustinus	W2	309
29555	Feuerbach	2	2136	Curie	W3	36
29555	Feuerbach	2	2137	Kant	W3	7

Man kann nach Konstanten fragen:

In [6]: %%sql

```
SELECT
  42
```

```
from
  Student
```

```
* sqlite:///SQL-Beispieldb.db
```

```
Done.
```

```
Out[6]: 42
```

```
42
```

```
42
```

```
42
```

```
42
```

```
42
```

```
42
```

```
42
```

```
42
```

Einige Datenbanken haben spezielle "Dummy"-Tabellen mit genau einer Zeile zu dem Zweck um Berechnungen durchführen zu können (z.B., die "dual"-Table in Oracle).

```
In [7]: %%sql
```

```
SELECT
    42 AS The_Final_Answer
from
  Student
```

```
* sqlite:///SQL-Beispieldb.db
```

```
Done.
```

```
Out[7]: The_Final_Answer
```

```
42
```

```
42
```

```
42
```

```
42
```

```
42
```

```
42
```

```
42
```

```
42
```

Oder auch mit arithmetische Ausdrücke in der Select-Klausel rechnen. Da geht auch mit Attributwerten:

```
In [8]: %%sql

SELECT
    2*3*7 AS The_Final_Answer
from
    Student
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[8]: The_Final_Answer
```

The_Final_Answer
42
42
42
42
42
42
42
42

Mit `DISTINCT` bekommt man die Antworten ohne Duplikate

```
In [9]: %%sql

SELECT DISTINCT
    2*3*7 AS The_Final_Answer
from
    Student
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[9]: The_Final_Answer
```

The_Final_Answer
42

Beispiele für die Verwendung von WHERE

Suchen nach Werten von Attributen

```
In [10]: %%sql

SELECT
    *
from
    Student
WHERE
    Semester = 2
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[10]:
```

MatrNr	Name	Semester
29120	Theophrastos	2
29555	Feuerbach	2

Mit Attributwerten kann auch gerechnet werden:

```
In [11]: %%sql
SELECT
    Semester+1
from
    Student
WHERE
    Semester = 2
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[11]:
```

Semester+1
3
3

Relationen können benannt werden

```
In [12]: %%sql
SELECT
    *
from
    Student s
WHERE
    s.Semester = 2 AND s.Name='Feuerbach'
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[12]:
```

MatrNr	Name	Semester
29555	Feuerbach	2

Die Anatomie der WHERE-Klausel

Die Bedingung nach dem `WHERE` kann verschiedene Operatoren und Ausdrücke enthalten:

- Vergleichsoperatoren: $A \theta B$ für zwei Werte A und B mit $\theta \in \{<, >, =, <=, >=, <>\}$
- BETWEEN, wie z.B., "Semester BETWEEN 5 AND 10"
- LIKE, wie z.B., "A LIKE "RWTH%"

- IS NULL, zu überprüfen, ob ein Attribute einen Wert hat oder nicht
- Logische Verknüpfungen: AND, OR, NOT
- Arithmetische Operatoren: +, -, *, /
- Mengenoperatoren: IN, NOT IN, θ ANY, θ ALL, EXISTS, dabei ist θ einer der obigen Vergleichsoperatoren

Vergleichsoperatoren

Die üblichen Vergleichsoperatoren werden wie erwartet verwendet.

```
In [13]: %%sql
SELECT
    Semester
from
    Student
WHERE
    Semester > 5

* sqlite:///SQL-Beispieldb.db
Done.
```

Out[13]: **Semester**

18
12
10
8
6

```
In [14]: %%sql
SELECT
    Semester
from
    Student
WHERE
    Semester BETWEEN 5 AND 10

* sqlite:///SQL-Beispieldb.db
Done.
```

Out[14]: **Semester**

10
8
6

LIKE Bedingung

Suche nach Teilstrings: Das SQL-Schlüsselzeichen '%' repräsentiert einen beliebigen String. Ein Unterstrich '_' steht für ein beliebiges Zeichen. Mit dem Schlüsselwort 'ESCAPE' kann ein Escape-Zeichen definiert werden, um Prozentzeichen und Unterstrich werden zu können. Beispiel: Gibt es Studenten deren Namen mit 'ca' beginnt, dann ein beliebiges Zeichen gefolgt von einem 'n' enthält, und dann beliebig weitergeht?

```
In [15]: %%sql

SELECT
    *
from
    Student
where
    Name LIKE 'ca_n%'

* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[15]: MatrNr   Name   Semester
         -----
         28106  Carnap         3
```

IS NULL Bedingung

Test auf Null-Wert: IS [NOT] NULL Beispiel: Gib alle Studenten aus, deren Semester nicht vorhanden ist. Die Bedingung kann nicht als "Semester = NULL" geschrieben werden.

```
In [16]: %%sql

SELECT
    Name
from
    Student
WHERE
    Semester IS NOT NULL

* sqlite:///SQL-Beispieldb.db
Done.
```

Out[16]:

Name
Xenokrates
Jonas
Fichte
Aristoxenos
Schopenhauer
Carnap
Theophrastos
Feuerbach

Verknüpfte Bedingungen

Bedingungen in einer WHERE-Klausel können logisch verknüpft werden. Welche Chefs haben die Assistenten, die die Fachgebiete "Sylogistik" oder "Ideenlehre" vertreten?

```
In [17]: %%sql
SELECT DISTINCT Boss
FROM Assistent
WHERE Fachgebiet = 'Sylogistik' OR Fachgebiet = 'Ideenlehre';

* sqlite:///SQL-Beispieldb.db
Done.
```

Out[17]:

Boss
2125

Mengenoperationen und Verschachtelungen

In SQL können Mengenoperationen in einer WHERE -Bedingung verwendet werden, um auf Basis von Mengenbeziehungen zwischen Datensätzen zu filtern. Hier sind einige gängige Mengenoperationen, die in einer WHERE -Bedingung verwendet werden können:

1. **IN**: Die IN -Operation wird verwendet, um festzustellen, ob ein bestimmter Wert in einer Liste oder einem Ergebnis einer Subquery enthalten ist. Sie gibt TRUE zurück, wenn der Wert in der Liste oder dem Ergebnis der Subquery enthalten ist, und FALSE sonst.

Beispiel:

```
In [18]: %%sql
SELECT * FROM Student
WHERE Semester IN (1, 2, 3);
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[18]:
```

MatrNr	Name	Semester
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

2. **NOT IN**: Die **NOT IN** -Operation ist das Gegenteil von **IN** . Sie gibt **TRUE** zurück, wenn der Wert *nicht* in der Liste oder dem Ergebnis der Subquery enthalten ist, und **FALSE** sonst.

Beispiel:

```
In [19]: %%sql

SELECT * FROM Student
WHERE Semester NOT IN (1, 2, 3);
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

```
Out[19]:
```

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6

Man kann Anfragen auch verschachteln:

```
In [20]: %%sql

SELECT DISTINCT
    *
FROM
    Student
WHERE
    Semester NOT IN (
        Select
            Semester
        from
            Student
        where
            Semester = 2)
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

Out[20]:

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3

5. **EXISTS**: Die `EXISTS` -Operation wird verwendet, um festzustellen, ob eine Subquery mindestens einen Datensatz zurückgibt. Sie gibt `TRUE` zurück, wenn die Subquery mindestens einen Datensatz zurückgibt, und `FALSE` sonst. (Bei `NOT EXISTS` umgekehrt). Beispiel: Finde alle Studierenden, die weniger Semester studiert haben als "Fichte")

In [21]:

```
%%sql
SELECT * FROM Student s1
WHERE EXISTS
(SELECT
  *
  FROM
    Student s2
  WHERE
    s1.Semester < s2.Semester
  AND
    s2.Name = 'Fichte'
);
```

* sqlite:///SQL-Beispieldb.db
Done.

Out[21]:

MatrNr	Name	Semester
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

3. θ **ANY**: Die θ ANY-Operation (wobei θ einer der obigen Vergleichsoperatoren ist, z.B. `<`, `>`, `=`, usw.) wird verwendet, um einen Wert mit *jedem* Wert in einer Liste oder einem Ergebnis einer Subquery zu vergleichen. Sie gibt `TRUE` zurück, wenn der Vergleich für mindestens einen Wert in der Liste oder dem Ergebnis der Subquery zutrifft, und `FALSE` sonst. θ **SOME** ist äquivalent zu θ **ANY**.

θ ANY . Die θ ANY-Operation wird nicht von SQLite unterstützt, man kann aber eine solche Query auch mit EXISTS ausdrücken (wie?). Beispiel:

```
SELECT * FROM Student
WHERE Semester <= ANY (SELECT Semester FROM Student)
```

4. θ ALL: Die θ ALL-Operation (wobei θ ein Vergleichsoperator ist) wird verwendet, um einen Wert mit *allen* Werten in einer Liste oder einem Ergebnis einer Subquery zu vergleichen. Sie gibt TRUE zurück, wenn der Vergleich für alle Werte in der Liste oder dem Ergebnis der Subquery zutrifft, und FALSE sonst. Die θ ALL-Operation wird nicht von SQLite unterstützt, aber auch diese Query kann mit Hilfe von EXISTS ausgedrückt werden (wie?). Beispiel:

```
SELECT * FROM Student
WHERE Semester < ALL (SELECT Semester FROM Student WHERE Name = 'Fichte');
```

Bemerkung: die letzte Anfrage ist ebenfalls ein Beispiel einer verschachtelten Query. Verschachtelungen gehen übrigens auch für den FROM Teil.

In [22]:

```
%%sql

SELECT * FROM (SELECT PersNr, Name FROM Professor WHERE Name LIKE '%so%')
WHERE PersNr > 10
```

* sqlite:///SQL-Beispieldb.db

Done.

Out[22]:

PersNr	Name
2125	Sokrates

SQL, die Relationale Algebra, und das Tupelkalkül

Relationale Algebra und SQL

Für eine allgemeine SQL-Anfrage:

```
SELECT DISTINCT A, B, C, ...
FROM R, S, T, ...
WHERE Bedingung
```

lautet die entsprechende Darstellung in relationaler Algebra:

$$\pi_{\{A,B,C,\dots\}}(\sigma_{\{\text{Bedingung}\}}(R \times S \times T \times \dots))$$

Und die Darstellung im Tupelkalkül: $\{[t_i.A, t_j.B, t_k.C, \dots] \mid t_1 \in R \wedge t_2 \in S \wedge t_3 \in T \wedge \dots \wedge \text{Bedingung}\}$

Operation	Relationale Algebra	SQL-Abfrage
Vereinigung	$R \cup T$	SELECT * FROM R UNION SELECT * FROM T
Differenz	$R - T$	SELECT * FROM R EXCEPT SELECT * FROM T
Kreuzprodukt	$R \times T$	SELECT * FROM R CROSS JOIN T
Selektion	$\sigma_{B=b}(R)$	SELECT * FROM R WHERE B = 'b'
Projektion	$\pi_{A,C}(R)$	SELECT DISTINCT A, C FROM R
Umbenennung (Relation)	$\rho_V(R)$	SELECT V.A FROM R AS V
Umbenennung (Attribut)	$\rho_{K \leftarrow C}(R)$	SELECT A, B, C AS K, D FROM R

Die von den SELECT-Anweisungen zurückgegebenen Spalten müssen den gleichen oder einen konvertierbaren Datentyp und die gleiche Größe haben und in der gleichen Reihenfolge angeordnet sein. Anders als bei der relationalen Algebra müssen die Attributnamen aber nicht übereinstimmen. Ein Beispiel zu UNION:

In [23]:

%%sql

```
SELECT MatrNr, Name from Student
WHERE Semester > 10
UNION SELECT PersNr, Name from Professor
WHERE Raum = 309
```

* sqlite:///SQL-Beispieldb.db

Done.

Out[23]:

MatrNr	Name
2134	Augustinus
24002	Xenokrates
25403	Jonas

JOINS



SQL für weitere Operationen der relationalen Algebra:

Operation	Relationale Algebra	SQL-Abfrage
Relation	R	<code>SELECT * FROM R</code>
Projektion (ohne Duplikate)	$\pi_{\{A,C\}}(R)$	<code>SELECT A, C FROM R</code>
Natürlicher Verbund (Natural Join)	$R \bowtie U$	<code>SELECT * FROM R NATURAL JOIN U</code>
Theta-Join	$R \bowtie_{\{B \theta F\}} S$	<code>SELECT * FROM R, S WHERE B θ F</code>
Theta-Join	$R \bowtie_{\{B \theta F\}} S$	<code>SELECT * FROM R JOIN S ON B θ F</code>

Analog auch andere Join-Arten

- `LEFT (OUTER) JOIN`
- `RIGHT (OUTER) JOIN`
- `FULL (OUTER) JOIN`

(Siehe Vorlesung über relationale Algebra über die verschiedenen JOINS!)

Beispiel für einen Selfjoin: Angenommen, Sie möchten alle Paare von Studenten finden, die dieselbe Vorlesung hören:

```
In [24]: %%sql
SELECT
    S1.Name AS Student1,
    S2.Name AS Student2,
    V.Titel AS Vorlesung
FROM
    Student S1
JOIN
```

```

Hört H1 ON S1.MatrNr = H1.MatrNr
JOIN
Vorlesung V ON H1.VorlNr = V.VorlNr
JOIN
Hört H2 ON V.VorlNr = H2.VorlNr
JOIN
Student S2 ON H2.MatrNr = S2.MatrNr
WHERE
S1.MatrNr < S2.MatrNr;

```

* sqlite:///SQL-Beispieldb.db
Done.

Out[24]:

Student1	Student2	Vorlesung
Fichte	Schopenhauer	Grundzüge
Fichte	Theophrastos	Grundzüge
Schopenhauer	Theophrastos	Grundzüge
Carnap	Theophrastos	Ethik
Jonas	Feuerbach	Glaube und Wissen

In diesem Beispiel verwenden wir die `Student` - und `Hört` -Tabellen zweimal, um zwei verschiedene Referenzen auf dieselbe Tabelle zu erstellen (`S1` und `H1` für die ersten Referenzen, `S2` und `H2` für die zweiten Referenzen). Der Self-Join erfolgt über die `Vorlesung` -Tabelle, die die gemeinsame Vorlesung repräsentiert, die beide Studenten hören. Die `WHERE` -Klausel stellt sicher, dass jedes Studentenpaar nur einmal in den Ergebnissen erscheint (durch den Vergleich von `S1.MatrNr` und `S2.MatrNr`).

Von SQL zum Tupelkalkül

Beispiel `SELECT DISTINCT` einmal als relationale Algebra und als Tupelkalkül:

```

SELECT DISTINCT Boss
FROM Assistent
WHERE Fachgebiet = 'Syllogistik' OR Fachgebiet = 'Ideenlehre';

```

Relationale Algebra: $\pi_{\{\text{Boss}\}}(\sigma_{\{\text{Fachgebiet} = \text{'Syllogistik'} \vee \text{Fachgebiet} = \text{'Ideenlehre'}\}}(\text{Assistent}))$

Tupelkalkül: $\pi_{\{t.\text{Boss}\}} \mid t \in \text{Assistent} \wedge (t.\text{Fachgebiet} = \text{'Syllogistik'} \vee t.\text{Fachgebiet} = \text{'Ideenlehre'})$

Beispiel mit natürlichem Join als relationale Algebra und als Tupelkalkül: SQL:

```

SELECT DISTINCT
    Name, Titel
FROM
    Student NATURAL JOIN Hört NATURAL JOIN Vorlesung

```

WHERE Semester > 4

Relationale Algebra: $\pi_{\{\text{Name}, \text{Titel}\}}(\sigma_{\{\text{Semester} > 4\}}(\text{Student} \bowtie \text{Hört} \bowtie \text{Vorlesung}))$

Tupelkalkül:

$\{ [s.\text{Name}, v.\text{Titel}] \mid s \in \text{Student} \wedge \exists h \in \text{Hört} \wedge v \in \text{Vorlesung} \wedge s.\text{MatrNr} = h.\text{MatrNr} \wedge \exists h.\text{VorlNr} = v.\text{VorlNr} \wedge s.\text{Semester} > 4 \}$

In [25]:

```
%%sql

SELECT DISTINCT
    Name, Titel
FROM
    Student
NATURAL JOIN
    Hört
NATURAL JOIN
    Vorlesung
WHERE
    Semester > 4
```

* sqlite:///SQL-Beispieldb.db
Done.

Out[25]:

Name	Titel
Fichte	Grundzüge
Schopenhauer	Grundzüge
Schopenhauer	Logik
Jonas	Glaube und Wissen

Vom Tupelkalkül zu SQL oder Wozu zur Hölle ist das Tupelkalkül zu gebrauchen?

Die Frage: "Welche Studierenden hören alle Vorlesungen von Popper?" kann man nicht so einfach in SQL ausdrücken. Mit Prädikatenlogik geht dies viel einfacher und man kann die Anfrage mit einer ungebundenen Variable direkt hinschreiben: $\pi_{\{\text{Student}(s) \wedge \forall v (\text{Vorlesung}(v, \text{Popper}) \rightarrow \text{Hört}(s, v))\}}$

Der Tupelkalkül basiert auf Logik, daher kann man diese Formulierung mit den vorhandenen Tabellen der Datenbank umschreiben:

$\{ [s.\text{Name}] \mid \exists s \in \text{Student} \wedge \forall v \in \text{Vorlesung} (\exists p \in \text{Professor} (p.\text{Name} = \text{'Popper'} \wedge v.\text{gelesenVon} = p.\text{PersNr}) \rightarrow s.\text{Semester} > 4) \}$

$$\exists h \in \text{Hört}(h.\text{MatrNr} = s.\text{MatrNr} \wedge h.\text{VorlNr} = v. \\ \text{VorlNr})) \setminus \setminus$$

Wir wollen die Namen aller Studierenden ($s.\text{Name}$), die für alle Vorlesungen v , die von Professor Popper ($p.\text{Name} = \text{'Popper'}$) gelesen werden, diese auch hören ($h.\text{MatrNr} = s.\text{MatrNr}$). Aber: den Allquantor $\forall$ kann man in SQL so nicht verwenden - daher ist diese Anfrage unmittelbar nicht in SQL ausdrückbar. Der Allquantor kann aber mit folgender Äquivalenz umgeformt werden: $\forall a \psi \Leftrightarrow \neg \exists a \neg \psi$

d.h. der Allquantor $\forall v$ wird ersetzt durch $\neg \exists v \neg$.

Das Ergebnis ist:

$$\{s.\text{Name} \mid s \in \text{Student} \wedge \neg \exists v \in \text{Vorlesung} (\\ \neg (\exists p \in \text{Professor} (p.\text{Name} = \text{'Popper'} \wedge v. \\ \text{gelesenVon} = p.\text{PersNr})) \rightarrow \\ \exists h \in \text{Hört}(h.\text{MatrNr} = s.\text{MatrNr} \wedge h.\text{VorlNr} = v. \\ \text{VorlNr})) \setminus \setminus$$

Nach der Ersetzung der Implikation $\neg (x \rightarrow y) \Leftrightarrow x \wedge \neg y$ erhalten wir:

$$\{s.\text{Name} \mid s \in \text{Student} \wedge \neg \exists v \in \text{Vorlesung} (\\ (\exists p \in \text{Professor} (p.\text{Name} = \text{'Popper'} \wedge v.\text{gelesenVon} \\ = p.\text{PersNr})) \wedge \\ \neg \exists h \in \text{Hört}(h.\text{MatrNr} = s.\text{MatrNr} \wedge h.\text{VorlNr} = v. \\ \text{VorlNr})) \setminus \setminus$$

Die letzte Formel beschreibt eine Menge von Studentennamen, die alle Vorlesungen des Professors mit dem Namen "Popper" hören. Hier ist eine Schritt-für-Schritt-Erklärung der Formel:

1. $s \in \text{Student}$: Wähle einen Studenten s aus der Menge der Studenten, für die das folgende gelten muss:
2. $\neg \exists v \in \text{Vorlesung}(\dots)$: Es gibt keine Vorlesung v in der Menge der Vorlesungen, die die in Klammern stehende Bedingung erfüllt.
3. $(\exists p \in \text{Professor} (p.\text{Name} = \text{'Popper'} \wedge v.\text{gelesenVon} = p.\text{PersNr}))$: Es gibt einen Professor p in der Menge der Professoren, dessen Name "Popper" ist und der die Vorlesung v hält (gelesenVon entspricht der PersNr des Professors).
4. $\neg \exists h \in \text{Hört}(h.\text{MatrNr} = s.\text{MatrNr} \wedge h.\text{VorlNr} = v. \text{VorlNr})$: Es gibt keine Relation h in der Menge "Hört", die besagt, dass der Student s (mit der MatrNr von s) die Vorlesung v (mit der VorlNr von v) hört.

Die Formel besagt, dass wir die Namen aller Studenten (s.Name) erfragen, für die gilt, dass es keine Vorlesung gibt, die von Professor Popper gehalten wird und die der Student nicht hört. In anderen Worten, wir erhalten die Namen der Studenten, die alle Vorlesungen von Professor Popper hören.

Diese Formel können wir direkt in die folgende SQL Anfrage übersetzen.

```
In [26]: %%sql

SELECT s.Name
FROM Student s
WHERE NOT EXISTS (
    SELECT *
    FROM Vorlesung v
    WHERE EXISTS (
        SELECT *
        FROM Professor p
        WHERE p.Name = 'Popper' AND v.gelesenVon = p.PersNr
    ) AND NOT EXISTS (
        SELECT *
        FROM Hört h
        WHERE h.MatrNr = s.MatrNr AND h.VorlNr = v.VorlNr
    )
);
```

* sqlite:///SQL-Beispieldb.db

Done.

Out[26]: **Name**

Carnap

Wer hätte diese Anfrage direkt hinschreiben können? Doppelte Verneinungen sind schwierig handzuhaben - und das Tupelkalkül liefert einen systematischen Weg auch komplizierte Anfragen zu entwickeln.

Übrigends: Warum funktioniert diese Transformation für jede Anfrage?

Die folgende Anfrage ist äquivalent. (Warum eigentlich?)

```
In [27]: %%sql

SELECT s.Name
FROM Student s
WHERE NOT EXISTS (
    SELECT 42
    FROM Vorlesung v
    WHERE EXISTS (
        SELECT 42
        FROM Professor p
        WHERE p.Name = 'Popper' AND v.gelesenVon = p.PersNr
    ) AND NOT EXISTS (
        SELECT 42
```

```

        FROM Hört h
        WHERE h.MatrNr = s.MatrNr AND h.VorlNr = v.VorlNr
    )
);

```

```

* sqlite:///SQL-Beispieldb.db
Done.

```

Out[27]: **Name**

Carnap

Aggregatfunktionen, Gruppieren, Sortieren, und Paginieren

Aggregatfunktionen in SQL

In SQL werden Aggregatfunktionen verwendet, um Berechnungen auf einer Gruppe von Zeilen durchzuführen und einen einzigen Wert als Ergebnis zurückzugeben. Hier sind einige gängige Aggregatfunktionen:

1. **COUNT**: Zählt die Anzahl der Zeilen, die ein bestimmtes Kriterium erfüllen.

```
SELECT COUNT(*) FROM Tabelle WHERE Bedingung;
```

2. **SUM**: Summiert die Werte in einer bestimmten Spalte für Zeilen, die ein bestimmtes Kriterium erfüllen.

```
SELECT SUM(Spalte) FROM Tabelle WHERE Bedingung;
```

3. **AVG**: Berechnet den Durchschnittswert einer bestimmten Spalte für Zeilen, die ein bestimmtes Kriterium erfüllen.

```
SELECT AVG(Spalte) FROM Tabelle WHERE Bedingung;
```

4. **MIN**: Gibt den kleinsten Wert in einer bestimmten Spalte für Zeilen zurück, die ein bestimmtes Kriterium erfüllen.

```
SELECT MIN(Spalte) FROM Tabelle WHERE Bedingung;
```

5. **MAX**: Gibt den größten Wert in einer bestimmten Spalte für Zeilen zurück, die ein bestimmtes Kriterium erfüllen.

```
SELECT MAX(Spalte) FROM Tabelle WHERE Bedingung;
```

- Zusätzlich werden STDDEV (Standardabweichung) und VARIANCE angeboten.
- Um die Anzahl unterschiedlicher Attributwerte in einer Spalte (in Algebra: Größe der Projektion) zu bestimmen, muss DISTINCT verwendet werden (normalerweise ist das bei Statistik-Anwendungen nicht erwünscht).
- NULL-Werte werden bei allen Funktionen vorher aus der Argumentmenge eliminiert.
- Falls die Argumentmenge leer ist, liefert COUNT den Wert 0, die anderen Funktionen NULL.

Beispiele: Wieviele Studierende sind in der Datenbank, wie lange haben sie im Durchschnitt studiert, was ist die Gesamtsumme aller Studiensemester, and was ist die kürzeste und längste Studiendauer?

```
In [28]: %%sql

SELECT
    COUNT(*), AVG(Semester), SUM(Semester), MIN(Semester), Max(Semester)
FROM
    Student
```

* sqlite:///SQL-Beispieldb.db

Done.

```
Out[28]: COUNT(*)  AVG(Semester)  SUM(Semester)  MIN(Semester)  Max(Semester)
-----
           8           7.625           61              2             18
```

Group-By in SQL



In SQL wird die `GROUP BY`-Klausel verwendet, um Zeilen in einer Tabelle basierend auf den Werten einer oder mehrerer Spalten zu gruppieren. Zusammen mit Aggregatfunktionen wie `COUNT`, `SUM`, `AVG`, `MIN` und `MAX` können Sie Berechnungen auf jeder Gruppe von Zeilen durchführen.

Beispiel:

```
SELECT Spalte1, AVG(Spalte2)
FROM Tabelle
GROUP BY Spalte1;
```

Beispiel: Angenommen, Sie möchten die Anzahl der Studenten, die jede Vorlesung besuchen, zusammen mit dem Titel der Vorlesung und dem Namen des Professors, der die Vorlesung hält, abrufen:

```
In [29]: %%sql

SELECT
    V.Titel,
    P.Name AS Professor,
    COUNT(*) AS Anzahl_Studenten
FROM
    Student S
NATURAL JOIN
    Hört
NATURAL JOIN
    Vorlesung V
JOIN
    Professor P ON V.gelesenVon = P.PersNr
GROUP BY
    V.Titel,
    P.Name
```

```
* sqlite:///SQL-Beispieldb.db
Done.
```

Out[29]:

Titel	Professor	Anzahl_Studenten
Bioethik	Russel	1
Der Wiener Kreis	Popper	1
Ethik	Sokrates	2
Glaube und Wissen	Augustinus	2
Grundzüge	Kant	3
Logik	Sokrates	1
Mäeutik	Sokrates	1
Wissenschaftstheorie	Russel	1

In diesem Beispiel werden die Tabellen Student, Hört, Vorlesung und Professor über JOIN-Operationen verbunden. Die GROUP BY-Klausel gruppiert die Ergebnisse basierend auf dem Titel der Vorlesung und dem Namen des Professors. Die COUNT(*)-Aggregatfunktion zählt die Anzahl der Studenten in jeder Gruppe.

HAVING Fun already?

In SQL wird die HAVING -Klausel verwendet, um die Ergebnisse einer Abfrage zu filtern, die mit der GROUP BY -Klausel und Aggregatfunktionen erstellt wurde. Die HAVING -Klausel wird nach der GROUP BY -Klausel angegeben und ermöglicht es Ihnen, Bedingungen für die Ergebnisse der Aggregatfunktionen anzugeben.

Beispiel:

```
SELECT Spalte1, COUNT(*)
FROM Tabelle
GROUP BY Spalte1
HAVING COUNT(*) > 5;
```

In diesem Beispiel werden die Zeilen in der Tabelle basierend auf den Werten in Spalte1 gruppiert und die Anzahl der Zeilen in jeder Gruppe gezählt. Die HAVING-Klausel filtert die Ergebnisse, sodass nur Gruppen mit einer Anzahl von mehr als 5 angezeigt werden.

Es ist wichtig zu beachten, dass die HAVING-Klausel nur in Kombination mit der GROUP BY-Klausel und Aggregatfunktionen verwendet werden sollte. Wenn Sie die Ergebnisse einer Abfrage ohne Gruppierung filtern möchten, verwenden Sie stattdessen die WHERE-Klausel.

Beispiel: Angenommen, Sie möchten nur die Vorlesungen abrufen, bei denen mindestens 2 Studenten eingeschrieben sind. Sie können die HAVING-Klausel verwenden, um die Ergebnisse zu filtern:

In [30]: %%sql

```

SELECT
    V.Titel,
    P.Name AS Professor,
    COUNT(*) AS Anzahl_Studenten
FROM
    Student
NATURAL JOIN
    Hört
NATURAL JOIN
    Vorlesung V
JOIN
    Professor P ON V.gelesenVon = P.PersNr
GROUP BY
    V.Titel,
    P.Name
HAVING
    COUNT(*) >= 2

```

* sqlite:///SQL-Beispieldb.db

Done.

Out[30]:

	Titel	Professor	Anzahl_Studenten
	Ethik	Sokrates	2
	Glaube und Wissen	Augustinus	2
	Grundzüge	Kant	3

Find den Namen aller Studierenden mit mehr als 2 Semestern, die mehr als eine Vorlesung hören

In [31]:

```

%%sql
select Name
from Student
NATURAL JOIN
    Hört
Where
    Semester > 2
GROUP BY
    Name
HAVING
    count() > 1

```

* sqlite:///SQL-Beispieldb.db

Done.

Out[31]:

Name
Carnap
Schopenhauer

In []: Die gleiche Anfrage ohne HAVING

In [32]:

```

%%sql
select Name

```

```

from Student s1
where
    Semester > 2 AND
    1 < (select count() from Hört
        WHERE s1.MatrNr= Hört.MatrNr)

```

* sqlite:///SQL-Beispieldb.db
Done.

Out[32]:

Name
Schopenhauer
Carnap

Welche Studierenden haben 50% länger studiert als der Durchschnitt aller Studierenden?

In [33]:

```

%%sql

select Name, Semester
from Student
group by Name
having Semester > (select 1.5 * avg(Semester) from Student)

```

* sqlite:///SQL-Beispieldb.db
Done.

Out[33]:

Name	Semester
Jonas	12
Xenokrates	18

Ordnung ins Chaos: ORDER-BY

Die `ORDER BY`-Klausel wird in SQL verwendet, um die Ergebnisse einer Abfrage basierend auf den Werten einer oder mehrerer Spalten zu sortieren. Sie können die Sortierreihenfolge mit den Schlüsselwörtern `ASC` (aufsteigend) oder `DESC` (absteigend) angeben. Wenn keine Sortierreihenfolge angegeben ist, wird standardmäßig `ASC` verwendet.

Beispiel:

```

SELECT Spalte1, Spalte2
FROM Tabelle
ORDER BY Spalte1 ASC, Spalte2 DESC;

```

In diesem Beispiel werden die Ergebnisse basierend auf den Werten in Spalte1 in aufsteigender Reihenfolge und dann basierend auf den Werten in Spalte2 in absteigender Reihenfolge sortiert.

Beispiel: Wir wollen die letzte Anfrage absteigend sortieren lassen

In [34]:

```

%%sql

```

```

SELECT
    V.Titel,
    P.Name AS Professor,
    COUNT(*) AS Anzahl_Studenten
FROM
    Student S
NATURAL JOIN
    Hört H
NATURAL JOIN
    Vorlesung V
JOIN
    Professor P ON V.gelesenVon = P.PersNr
GROUP BY
    V.Titel,
    P.Name
HAVING
    COUNT(*) >= 2
ORDER BY
    Anzahl_Studenten DESC;

```

* sqlite:///SQL-Beispieldb.db

Done.

Out[34]:

	Titel	Professor	Anzahl_Studenten
	Grundzüge	Kant	3
	Ethik	Sokrates	2
	Glaube und Wissen	Augustinus	2

SQL LIMIT, OFFSET, und OFFSET FETCH

`LIMIT` und `OFFSET` sind SQL-Schlüsselworte, die dazu verwendet werden, die Anzahl der in einer Abfrage zurückgegebenen Zeilen zu steuern. Sie sind nützlich für die Implementierung von Paginierung in Anwendungen und für das Abrufen einer Teilmenge von Daten aus einer größeren Ergebnismenge. Sie gehören aber nicht zum SQL Standard.

```

SELECT spalte1, spalte2, ...
FROM tabellenname
LIMIT <anzahl_der_reihen>
OFFSET <offset_wert>;

```

In dieser Abfrage wird `LIMIT` verwendet, um die maximale Anzahl der zurückgegebenen Zeilen anzugeben, und `OFFSET` wird verwendet, um die Anzahl der zu überspringenden Zeilen vor dem Beginn der Rückgabe von Zeilen anzugeben.

Mit SQL:2008 wurde die `OFFSET FETCH` -Klausel eingeführt, die eine ähnliche Funktion wie die `LIMIT` -Klausel hat. Mit der `OFFSET FETCH` -Klausel können Sie die ersten `n` Zeilen in einer Ergebnismenge überspringen, bevor Sie mit der Rückgabe von Zeilen beginnen. Hier ist ein Beispiel für eine Abfrage mit `OFFSET FETCH` in SQL:

```

SELECT column1, column2, ...
FROM table_name
ORDER BY column_name
OFFSET <offset_value> ROWS
FETCH NEXT <number_of_rows> ROWS ONLY;

```

SQLite implementiert OFFSET FETCH nicht. Wir werden in den Übungen und der Klausur ggf. LIMIT verwenden, um Ausschnitte aus großen Beispieltabellen zu zeigen.

Zusammenfassung: Anatomie und Formulierungsstrategie von SQL-Anfragen

Klausel	Reihenfolge	Semantik (was passiert?)
Select (DISTINCT)	5	Projektion: Übernehme nur die genannten Spalten, streiche die restlichen; wende Funktionen (sum, avg, ...) an. Streiche Duplikate aus Ergebnis-Tabelle
FROM	1	Bilde das kartesische Produkt (... , ...) oder den Join (... JOIN ... ON ...) über die angegebenen Tabellen
WHERE	2	Selektion: streiche alle Tupel des Joins/kart. Produkts, die die WHERE-Bedingung nicht erfüllen
GROUP BY	3	Gruppiere Tupel der selektierten Eingabedatenkombinationen
HAVING	4	Streiche alle Tupel der Gruppierung, die die HAVING-Bedingung nicht erfüllen
ORDER BY	6	Sortiere das Ergebnis

SQL-Injection : Datenbanken unter Beschuss



(Quelle: <https://xkcd.com/327/>)

Was ist SQL-Injection?

SQL-Injection ist eine häufige Sicherheitslücke in Webanwendungen, bei der Angreifer

schädliche SQL-Code in Eingabefelder oder URL-Parameter einfügen. Dadurch erhalten sie unerlaubten Zugriff auf die zugrunde liegende Datenbank, was zu Datenlecks, Datenmanipulation oder sogar kompletten Datenbankübernahmen führen kann.

Wie funktioniert SQL-Injection?

Ein SQL-Injection-Angriff erfolgt, wenn Benutzereingaben direkt in SQL-Abfragen eingefügt werden, ohne diese Eingaben ordnungsgemäß zu validieren oder zu bereinigen. Dies ermöglicht es Angreifern, SQL-Befehle in die Abfrage einzufügen, die nicht beabsichtigt waren, und dadurch auf sensible Daten zuzugreifen oder die Datenbank zu manipulieren.

Beispiele für SQL-Injection

Login-Formular

Ein einfaches Beispiel für eine anfällige Anwendung ist ein Login-Formular, das die folgende SQL-Abfrage verwendet, um den Benutzer zu authentifizieren:

```
SELECT * FROM users WHERE username = '[EINGABE]' AND password = '[EINGABE]';
```

Ein Angreifer kann versuchen, das Formular mit dem Benutzernamen ' OR 1=1 --' und einem beliebigen Passwort auszufüllen. Dies führt zu folgender SQL-Abfrage:

```
SELECT * FROM users WHERE username = '' OR 1=1 --' AND password = '[EINGABE]';
```

Da `1=1` immer wahr ist, wird der Angreifer als gültiger Benutzer authentifiziert, ohne ein korrektes Passwort zu kennen. Der doppelte Bindestrich `--` ist ein Kommentarzeichen in SQL. Alles, was nach dem doppelten Bindestrich kommt, wird als Kommentar behandelt und nicht als Teil der SQL-Abfrage ausgeführt. In diesem Fall wird die Bedingung `AND password = '[EINGABE]'` ignoriert.

Benutzereingaben

Was ist mit dem obigen Comic? Die Zeile `Robert'); DROP TABLE STUDENTS; --)` aus dem obigen Comic ist problematisch. Angenommen ein Benutzer gibt in einer Eingabemaske den String `Robert'); DROP TABLE STUDENTS; --)` als Vorname und `Nachname` als Nachnamen ein, dann würde dies zusammen mit der Programmzeile:

```
q = "INSERT INTO Students VALUES ('" + FirstName.Text + "', '" + LastName.Text + "')";
```

die folgenden SQL Befehle ergeben:

```
INSERT INTO Students VALUES ('Robert'); DROP TABLE Students; --', 'Nachname')
```

Diese machen zwei Dinge:

1. Fügt der Tabelle Students eine neue Zeile mit dem Namen "Robert" hinzu.
2. Löscht die Tabelle Students Alles nach der zweiten Anfrage wird als Kommentar behandelt.

Wie kann man SQL-Injection verhindern?

Um SQL-Injection-Angriffe zu verhindern, sollten Sie folgende Sicherheitsmaßnahmen ergreifen:

1. **Benutzereingaben validieren:** Validieren Sie alle Benutzereingaben, um sicherzustellen, dass sie den erwarteten Datentyp und Format aufweisen.
2. **Parameterisierte Abfragen verwenden:** Verwenden Sie Prepared Statements oder Parameterized Queries, um sicherzustellen, dass Benutzereingaben nicht als Teil des SQL-Codes interpretiert werden.
3. **Stored Procedures nutzen:** Verwenden Sie Stored Procedures, um den Zugriff auf die Datenbank zu kontrollieren und den Umfang der möglichen Angriffe zu reduzieren.
4. **Least Privilege Prinzip anwenden:** Gewähren Sie Ihrer Anwendung nur die minimalen Berechtigungen, die sie benötigt, um auf die Datenbank zuzugreifen.
5. **Escaping von Eingaben:** Wenn die Verwendung von Parameterized Queries oder Stored Procedures nicht möglich ist, stellen Sie sicher, dass Benutzereingaben korrekt "escaped" werden, um schädliche Zeichen zu neutralisieren.

Indem Sie diese Best Practices befolgen, können Sie Ihre Anwendung und Datenbank vor SQL-Injection-Angriffen schützen und die Sicherheit Ihrer Benutzer und Daten gewährleisten.



In [35]: %%sql

```
SELECT * FROM Student WHERE MatrNr = '123' OR 1=1 --' AND password = '[EINGABE]';
```

* sqlite:///SQL-Beispieldb.db

Done.

Out[35]:

MatrNr	Name	Semester
--------	------	----------

24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

In []:

SQL Transaction Control Language (TCL)

In diesem Abschnitt werden wir die grundlegenden SQL-TCL-Befehle kennenlernen, die zum Verwalten von Transaktionen in einer relationalen Datenbank verwendet werden. Die vier Haupt-TCL-Befehle sind:

1. BEGIN TRANSACTION
2. COMMIT
3. ROLLBACK
4. SAVEPOINT

Transaktionen werden später noch ausführlich besprochen werden.

BEGIN TRANSACTION

Mit dem BEGIN TRANSACTION-Befehl können wir den Beginn einer neuen Transaktion festlegen. Die Syntax lautet:

```
BEGIN TRANSACTION;
```

COMMIT

Mit dem COMMIT-Befehl können wir die Änderungen, die während einer Transaktion vorgenommen wurden, dauerhaft in der Datenbank speichern. Die Syntax lautet:

```
COMMIT;
```

ROLLBACK

Mit dem ROLLBACK-Befehl können wir die Änderungen, die während einer Transaktion vorgenommen wurden, rückgängig machen und die Datenbank auf den Zustand vor der Transaktion zurücksetzen. Die Syntax lautet:

```
ROLLBACK;
```

SAVEPOINT

Mit dem SAVEPOINT-Befehl können wir einen Zwischenzustand innerhalb einer Transaktion festlegen, zu dem wir später zurückkehren können. Die Syntax lautet:

```
SAVEPOINT savepoint_name;
```

Um zu einem zuvor definierten Savepoint zurückzukehren, verwenden wir den ROLLBACK-

Befehl zusammen mit dem Savepoint-Namen:

```
ROLLBACK TO savepoint_name;
```

iPython unterstützt keine Transaktionen, daher hier ein explizites Program!

Demonstration von Begin, Commit, Rollback

```
In [1]: import sqlite3 as lite
import sys
```

1. Datenbankverbindung aufbauen

```
In [2]: con = lite.connect('SQL-TCL-db.db')
```

2. Testtabelle erzeugen

```
In [3]: with con:
    cur = con.cursor()
    # triple quotes for multi-line strings
    sql = """
    DROP TABLE IF EXISTS Student;
    CREATE TABLE Student ( MatrNr INTEGER PRIMARY KEY, Name VARCHAR(255), Semester
    INSERT INTO Student (MatrNr, Name, Semester) VALUES (4711, 'Max Mustermann', 3)
    INSERT INTO Student (MatrNr, Name, Semester) VALUES (4712, 'Ina Musterfrau', 3)
    """

    cur.executescript(sql)
```

3. Testtabelle ändern

```
In [4]: with con:
    cur = con.cursor()
    sql = """
    UPDATE Student
    SET Semester = 4
    WHERE
    MatrNr = 4711
    """

    cur.executescript(sql)
```

4. Daten anschauen

```
In [5]: with con:
    cur = con.cursor()
    cur.execute("SELECT * FROM Student")

    rows = cur.fetchall()
```

```
for row in rows:
    print(row)
```

```
(4711, 'Max Mustermann', 4)
(4712, 'Ina Musterfrau', 3)
```

5. ROLLBACK einer DELETE Aktion. ISOLATION von Datenbanktransaktionen werden bei später bei den ACID Eigenschaften besprechen.

```
In [6]: con.isolation_level = None
cur = con.cursor()
cur.execute("BEGIN TRANSACTION")
sql = """
        DELETE FROM Student WHERE MatrNr IS 4711
        """
cur = con.execute(sql)
cur.execute("ROLLBACK")
print("Done")
```

Done

6. Daten anschauen

```
In [7]: cur = con.cursor()
cur.execute("SELECT * FROM Student")
rows = cur.fetchall()

for row in rows:
    print(row)
```

```
(4711, 'Max Mustermann', 4)
(4712, 'Ina Musterfrau', 3)
```

7. COMMIT einer DELETE Transaktion

```
In [8]: cur = con.cursor()
cur.execute("BEGIN")
sql = """
        DELETE FROM Student WHERE MatrNr IS 4711
        """
cur = con.execute(sql)
cur.execute("COMMIT")
print("Done")
```

Done

Wenn man jetzt nachschaut ist die Zeile nicht mehr vorhanden

```
In [9]: cur = con.cursor()
cur.execute("SELECT * FROM Student")

rows = cur.fetchall()

for row in rows:
```

```
print(row)
```

```
(4712, 'Ina Musterfrau', 3)
```

8. Schliessen der Datenbankverbindung

```
In [10]: con.close()
```

```
In [ ]:
```

SQL Data Control Language (DCL)

Die Data Control Language (DCL) in SQL umfasst Befehle zum Verwalten von Benutzerberechtigungen und zur Steuerung des Zugriffs auf Datenbankressourcen. Die DCL geht über die Vorlesung hinaus und wird von SQLITE nicht unterstützt, daher stellen wir sie hier nur rudimentär vor.

Was ist DCL?

DCL steht für Data Control Language und ist ein Teil von SQL, der Befehle zum Verwalten von Benutzerberechtigungen und zum Steuern des Zugriffs auf Datenbankressourcen enthält. Die Hauptbefehle in DCL sind `GRANT` und `REVOKE`.

GRANT

`GRANT` ist ein DCL-Befehl, der zum Erteilen von Berechtigungen an Benutzer oder Rollen verwendet wird. Durch die Verwendung von `GRANT` können Sie steuern, welche Benutzer auf bestimmte Datenbankobjekte zugreifen und welche Aktionen sie ausführen können.

Syntax

```
GRANT permission_type ON object_name TO user_or_role [WITH GRANT OPTION];
```

Erläuterung

Beispiel

Angenommen, Sie möchten einem Benutzer namens `student_user` die Berechtigung zum Lesen (Auswählen) von Daten aus der Tabelle `Student` erteilen:

```
GRANT SELECT ON Student TO student_user;
```

- `permission_type` : Der Berechtigungstyp, den Sie gewähren möchten. Beispiele für Berechtigungstypen sind `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `EXECUTE` und `ALL PRIVILEGES`.
- `object_name` : Der Name des Datenbankobjekts, für das Sie die Berechtigung gewähren möchten, z. B. eine Tabelle, Sicht oder Prozedur.
- `user_or_role` : Der Benutzer oder die Rolle, dem/der Sie die Berechtigung gewähren möchten.
- `WITH GRANT OPTION` (optional): Wenn Sie diese Option angeben, kann der Benutzer oder die Rolle, dem/der Sie die Berechtigung gewähren, diese Berechtigung auch an andere Benutzer oder Rollen weitergeben.

REVOKE

REVOKE ist ein DCL-Befehl, der zum Entziehen von Berechtigungen von Benutzern oder Rollen verwendet wird. Mit REVOKE können Sie den Zugriff auf Datenbankressourcen einschränken oder entfernen, wenn ein Benutzer oder eine Rolle diese Berechtigungen nicht mehr benötigt.

Syntax

```
REVOKE [GRANT OPTION FOR] permission_type ON object_name FROM user_or_role  
[CASCADE | RESTRICT];
```

Erläuterung:

- **GRANT OPTION FOR** (optional): Wenn Sie diese Option angeben, entziehen Sie nur das Recht, die angegebene Berechtigung an andere Benutzer oder Rollen weiterzugeben, ohne die eigentliche Berechtigung zu entziehen.
- **permission_type** : Der Berechtigungstyp, den Sie entziehen möchten. Beispiele für Berechtigungstypen sind **SELECT** , **INSERT** , **UPDATE** , **DELETE** , **EXECUTE** und **ALL PRIVILEGES** .
- **object_name** : Der Name des Datenbankobjekts, für das Sie die Berechtigung entziehen möchten, z. B. eine Tabelle, Sicht oder Prozedur.
- **user_or_role** : Der Benutzer oder die Rolle, von dem/der Sie die Berechtigung entziehen möchten.
- **CASCADE** (optional): Wenn Sie diese Option angeben, werden alle abhängigen Berechtigungen, die von der ursprünglich gewährten Berechtigung abgeleitet wurden, ebenfalls entzogen.
- **RESTRICT** (optional): Wenn Sie diese Option angeben, wird das Entziehen der Berechtigung verhindert, wenn es abhängige Berechtigungen gibt, die von der ursprünglich gewährten Berechtigung abgeleitet wurden. In diesem Fall müssen Sie zuerst die abhängigen Berechtigungen explizit entziehen, bevor Sie die ursprüngliche Berechtigung entziehen können. Abhängige Berechtigungen sind Berechtigungen, die von einer ursprünglich gewährten Berechtigung abgeleitet oder indirekt durch sie ermöglicht wurden. Sie entstehen, wenn ein Benutzer oder eine Rolle, der/die eine Berechtigung erhalten hat, diese Berechtigung an andere Benutzer oder Rollen weitergibt.

Beispiel

Angenommen, Sie möchten die Berechtigung zum Lesen (Auswählen) von Daten aus der Tabelle **Student** vom Benutzer **student_user** entziehen:

```
REVOKE SELECT ON Student FROM student_user;
```

