Databases

Course Material • Hamburg Coding School • 16. + 17.01.2021

Outline

- Introduction
    - Databases
    - Classification
    - Differences
    - CRUD
- SQL Databases
    - Create a Database
    - Show Databases
    - Select a Database
    - Show Selected Database
    - Tables
    - Create a Table
    - Data Types
    - Data Type Specifications
    - Primary Key
    - Show All Tables
    - Show Columns of a Table
    - Create a Data Entry
    - Create Multiple Data Entries
    - Show All Rows of a Table
    - Show a Certain Column
    - Queries: Show Specific Rows
    - Sorting the Results
    - Update Certain Rows
    - Alter Columns of a Table
    - Delete Rows
    - Delete a Table
    - Delete Database
    - Relations
    - One-to-Many Relationships
    - Many-to-Many Relationships
    - Database Modeling
    - Joins

    - Inner Joins, Outer Joins, and even more Joins (optional knowledge)
    - Client Databases vs. Server Databases
- Document-based Databases
    - MongoDB
    - Installation
    - Command Line Tool
    - Show the Database
    - Switch to Other Database
    - Collections and Documents
    - Create a New Document
    - Insert Multiple Documents
    - Query All Documents
    - Query a Specific Document
    - Query Operators
    - Update
    - Delete All Documents Matching
    - Delete the First Document that Matches
    - Delete All
    - MongoDB With Node
    - Comparison MongoDB to Relational Databases
- Glossary
- Useful Links

Introduction

Databases

What is a database?

- A database is a system for storing data in an ordered way.
- It consists of the data that is stored and a software that manages it.
- Databases have a query language to retrieve and manipulate the data.

As an example: an SQL statement to retrieve data from a movie database.

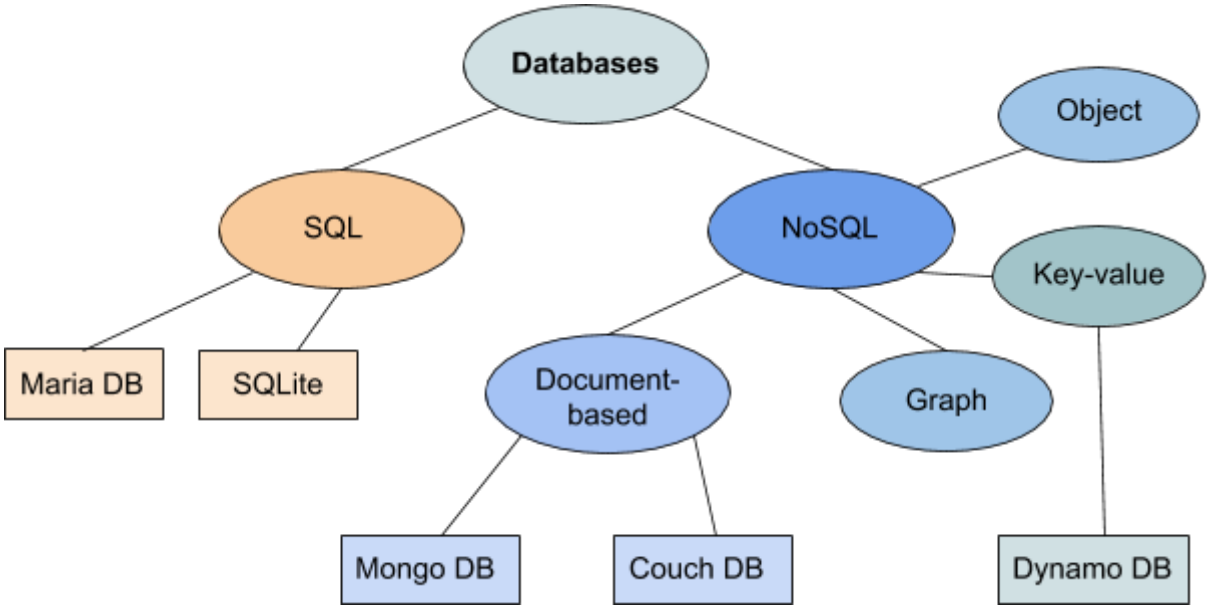
```
dvdrental=# select title, release_year, length, replacement_cost from film
dvdrental-#   where length > 120 and replacement_cost > 29.50
dvdrental-#   order by title desc;
          title          | release_year | length | replacement_cost
-------------------------+--------------+--------+------------------
 West Lion               |         2006 |    159 |            29.99
 Virgin Daisy            |         2006 |    179 |            29.99
 Uncut Suicides          |         2006 |    172 |            29.99
 Tracy Cider             |         2006 |    142 |            29.99
 Song Hedwig             |         2006 |    165 |            29.99
 Slacker Liaisons        |         2006 |    179 |            29.99
 Sassy Packer            |         2006 |    154 |            29.99
 River Outlaw            |         2006 |    149 |            29.99
 Right Cranes            |         2006 |    153 |            29.99
 Quest Mussolini         |         2006 |    177 |            29.99
 Poseidon Forever        |         2006 |    159 |            29.99
 Loathing Legally        |         2006 |    140 |            29.99
 Lawless Vision          |         2006 |    181 |            29.99
 Jingle Sagebrush        |         2006 |    124 |            29.99
 Jericho Mulan           |         2006 |    171 |            29.99
 Japanese Run            |         2006 |    135 |            29.99
 Gilmore Boiled          |         2006 |    163 |            29.99
 Floats Garden           |         2006 |    145 |            29.99
 Fantasia Park           |         2006 |    131 |            29.99
 Extraordinary Conquerer |         2006 |    122 |            29.99
 Everyone Craft          |         2006 |    163 |            29.99
 Dirty Ace               |         2006 |    147 |            29.99
 Clyde Theory            |         2006 |    139 |            29.99
 Clockwork Paradise      |         2006 |    143 |            29.99
 Ballroom Mockingbird    |         2006 |    173 |            29.99
(25 rows)
```

👉 **Public example databases to try out online:**

https://www.sachsen.schule/~terra2014/sql_abfragen.php
https://sqlzoo.net/wiki/SELECT_basics

Classification

There are multiple ways of classifying databases. A common way is this:

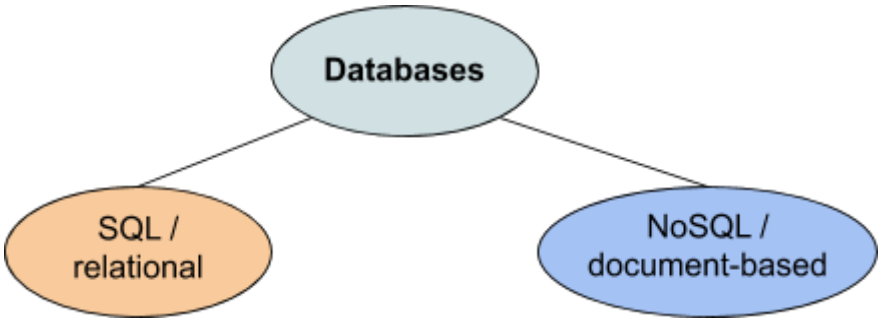


It is common practice to use these terms synonymously:

Relational databases = **SQL databases**

Document-based databases = **NoSQL databases**

Therefore, a simplified classification would be:



Differences

These two types of databases are used for different use-cases and have different characteristics.

Relational Databases	Document-based Databases
The classic type of databases, made popular by MySQL.	The modern approach to databases, addresses problems in web development.
Useful for more complex data that has a lot of relationships between entities.	Useful for large amounts of data that is rather uniform.
Structured	Mostly unstructured
Complex to set up	Simple to query
Data integrity built in	Data integrity relies heavily on programming
Uses SQL	Uses a custom query language, often resembling JavaScript
Examples: MariaDB, MySQL, SQLite (mobile), SQL Server	***Examples:*** MongoDB, CouchDB, Firebase Cloud Firestore

CRUD

Regardless of the type of database we use, we will always have four types of operations:

C - Create

R - Read

U - Update

D - Delete

💡 **CRUD** is not only used in databases, but also for REST APIs.

SQL Databases

SQL - Structured Query Language

SQL is the language that relational databases use for accessing their data.

Every **DBMS** (DataBase Management System, a fancy name for database software) has slight differences in how they implement SQL. But there is a large set of standard commands that are the same in all SQL databases. For now, don't worry about the differences.

In this course, we will work with **MariaDB**.

> 👉 MariaDB Knowledge Base: https://mariadb.com/kb/en/

Create a Database

To create a new database, use this command:

```
SQL> CREATE DATABASE testDB;
```

In general, the syntax is like this:

```
CREATE DATABASE DatabaseName;
```

> 🤯 **Note:** In MariaDB, you can write all SQL statements in lowercase as well. For example, **`create database testDB;`** will create the database just as well. Here, we will use uppercase, so that you see better which words are SQL keywords.

Show Databases

To show all databases, use:

```
SQL> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

Select a Database

To select a database to work with, use:

```
SQL> USE test;
Database changed
```

Show Selected Database

To show which database is selected, use:

```
SQL> SELECT database();
+------------+
| database() |
+------------+
| test       |
+------------+
1 row in set (0.000 sec)
```

Tables

In relational databases, data is organized in tables.

This example shows a table called “Customers”:

```
+----+----------+-----+-----------+----------+
| id | name     | age | address   | salary   |
+----+----------+-----+-----------+----------+
|  1 | Alice    |  32 | Hamburg   |  2000.00 |
|  2 | Bob      |  25 | Berlin    |  1500.00 |
|  3 | Max      |  23 | Köln      |  2000.00 |
|  4 | Maria    |  25 | München   |  6500.00 |
|  5 | Philipp  |  27 | Stuttgart |  8500.00 |
|  6 | Stefan   |  22 | Hamburg   |  4500.00 |
|  7 | Claudia  |  24 | Berlin    | 10000.00 |
+----+----------+-----+-----------+----------+
```

In such a table, a column is an **attribute**, and a row is a **data entry**.

Create a Table

On the command line, a table is created like this:

```
SQL> CREATE TABLE Customers(
   id       INT              NOT NULL,
   name     VARCHAR(20)      NOT NULL,
   age      INT              NOT NULL,
   address  TEXT,
   salary   DECIMAL(18, 2),
   PRIMARY KEY (id)
);
```

In general, the syntax is like this:

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

Data Types

For each column, you need to define the data type.

Common data types are:

```
INT
FLOAT
DECIMAL        DECIMAL(#digits, #digits after floating point)
BOOLEAN
VARCHAR        VARCHAR(length)
TEXT
ENUM
```

All other data types you can look up: https://mariadb.com/kb/en/library/data-types/

Data Type Specifications

Some data types you can specify even further:

```
UNSIGNED        only positive numbers
SIGNED          can have a sign (can be negative numbers as well)
NOT NULL        cannot be null, is required
AUTO_INCREMENT  if not set, increment automatically (used for IDs)
```

Primary Key

Every table needs one field that is unique. This is necessary for the database to uniquely identify each row in the table. This field (or column) is called the ***primary key***.

You usually use integers for it. They need to be **`NOT NULL`**.

In this example, we create a table with a column called "ID" and then specify in the last row that this column is the primary key:

```
SQL> CREATE TABLE Customers(
   id       INT              NOT NULL,
   name     VARCHAR(20)      NOT NULL,
   age      INT              NOT NULL,
   address  TEXT,
   salary   DECIMAL(18, 2),
   PRIMARY KEY (id)
);
```

It is recommended to set this column to **`AUTO_INCREMENT`**. This means that if you don't specify it for a new row, the database automatically sets it to a value that is the last value +1.

```
SQL> CREATE TABLE Customers(
   id INT NOT NULL AUTO_INCREMENT,
   ...
   PRIMARY KEY (ID)
);
```

Show All Tables

To show all tables in a database, use:

```
SQL> SHOW TABLES;
+----------------+
| Tables_in_test |
+----------------+
| Customers      |
| Students       |
+----------------+
2 rows in set (0.000 sec)
```

Show Columns of a Table

To show all columns and their data types, use:

```
SQL> DESCRIBE Students;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(20) | NO   |     | NULL    |                |
| email | varchar(20) | NO   |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
3 rows in set (0.010 sec)
```

Create a Data Entry

To insert a row into a table, use:

```
SQL> INSERT INTO Students(name, email) VALUES ('John Doe', 'john@doe.com');
Query OK, 1 row affected (0.012 sec)
```

Create Multiple Data Entries

Creating multiple rows at once works like this:

```
SQL> INSERT INTO Customers(name, age, address, salary) VALUES
    -> ('Teresa', 33, 'Borselstr. 7, 22765 Hamburg', 30000),
    -> ('John', 25, 'Hauptstr. 1, 22087 Hamburg', 45000),
    -> ('Max', 35, 'Bernstorffstr. 118, 22796 Hamburg', 50000);
Query OK, 3 rows affected (0.006 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Show All Rows of a Table

To show all rows of a table, use:

```
SQL> SELECT * FROM Customers;
+----+--------+-----+-----------------------------------+----------+
| id | name   | age | address                           | salary   |
+----+--------+-----+-----------------------------------+----------+
|  1 | Teresa |  33 | Borselstr. 7, 22765 Hamburg       | 30000.00 |
|  2 | John   |  25 | Hauptstr. 1, 22087 Hamburg        | 45000.00 |
|  3 | Max    |  35 | Bernstorffstr. 118, 22796 Hamburg | 50000.00 |
+----+--------+-----+-----------------------------------+----------+
3 rows in set (0.000 sec)
```

Show a Certain Column

To show only a certain column, use:

```
SQL> SELECT name FROM Customers;
+--------+
| name   |
+--------+
| Teresa |
| John   |
| Max    |
+--------+
3 rows in set (0.003 sec)
```

Queries: Show Specific Rows

Queries are the most important feature of the SQL language. You use it to show only specific rows that match a certain search criteria – the query.

```
SQL> SELECT * FROM Customers WHERE salary > 40000;
+----+------+-----+-----------------------------------+----------+
| id | name | age | address                           | salary   |
+----+------+-----+-----------------------------------+----------+
|  2 | John |  25 | Hauptstr. 1, 22087 Hamburg        | 45000.00 |
|  3 | Max  |  35 | Bernstorffstr. 118, 22796 Hamburg | 50000.00 |
+----+------+-----+-----------------------------------+----------+
2 rows in set (0.012 sec)
```

This follows a certain pattern:

```
SELECT <columns>   # what to show in the result
FROM <table>       # the table to search
WHERE <query>;     # what we search for
```

Have a look at some more examples. You can test them out on the website: https://www.sachsen.schule/~terra2014/ergebnis.php

```
SELECT * FROM BERG
WHERE B_NAME = "Chimborazo"


SELECT * FROM BERG
WHERE HOEHE >= 7001


SELECT * FROM BERG
WHERE HOEHE >= 7000 AND HOEHE <= 8000


SELECT * FROM BERG
WHERE HOEHE BETWEEN 7000 AND 8000
```

Sorting the Results

You can sort the results like this:

```
SQL> SELECT * FROM Customers WHERE salary > 40000 ORDER BY salary;
+----+------+-----+-----------------------------------+----------+
| id | name | age | address                           | salary   |
+----+------+-----+-----------------------------------+----------+
|  2 | John |  25 | Hauptstr. 1, 22087 Hamburg        | 45000.00 |
|  3 | Max  |  35 | Bernstorffstr. 118, 22796 Hamburg | 50000.00 |
+----+------+-----+-----------------------------------+----------+
2 rows in set (0.005 sec)
```

In ascending order:

```
SQL> SELECT * FROM Customers WHERE salary > 40000 ORDER BY salary ASC;
+----+------+-----+-----------------------------------+----------+
| id | name | age | address                           | salary   |
+----+------+-----+-----------------------------------+----------+
|  2 | John |  25 | Hauptstr. 1, 22087 Hamburg        | 45000.00 |
|  3 | Max  |  35 | Bernstorffstr. 118, 22796 Hamburg | 50000.00 |
+----+------+-----+-----------------------------------+----------+
2 rows in set (0.000 sec)
```

In descending order:

```
SQL> SELECT * FROM Customers WHERE salary > 40000 ORDER BY salary DESC;
+----+------+-----+-----------------------------------+----------+
| id | name | age | address                           | salary   |
+----+------+-----+-----------------------------------+----------+
|  3 | Max  |  35 | Bernstorffstr. 118, 22796 Hamburg | 50000.00 |
|  2 | John |  25 | Hauptstr. 1, 22087 Hamburg        | 45000.00 |
+----+------+-----+-----------------------------------+----------+
2 rows in set (0.000 sec)
```

Update Certain Rows

To update a certain column, we use a query statement as well. It looks like this:

```
SQL> UPDATE Customers SET salary = 45000 WHERE salary > 45000;
Query OK, 1 row affected (0.010 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

This would update the data to this:

```
SQL> SELECT * FROM Customers;
+----+--------+-----+-----------------------------------+----------+
| id | name   | age | address                           | salary   |
+----+--------+-----+-----------------------------------+----------+
|  1 | Teresa |  33 | Borselstr. 7, 22765 Hamburg       | 30000.00 |
|  2 | John   |  25 | Hauptstr. 1, 22087 Hamburg        | 45000.00 |
|  3 | Max    |  35 | Bernstorffstr. 118, 22796 Hamburg | 45000.00 |
+----+--------+-----+-----------------------------------+----------+
3 rows in set (0.000 sec)
```

Alter Columns of a Table

In addition to updating rows in a table, you can also change the table's columns. This is a complex topic, because you can change a lot of different things (e.g. the data type or default value of a column). We focus on adding and removing columns here.

To add a new column to a table, use:

```
SQL> ALTER TABLE Students ADD COLUMN address VARCHAR(100);
Query OK, 0 rows affected (0.046 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To remove a column, use:

```
SQL> ALTER TABLE Students DROP COLUMN address;
Query OK, 0 rows affected (0.018 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

You can look up more **ALTER TABLE** commands here:
https://mariadb.com/kb/en/alter-table/

Delete Rows

You can delete certain rows like this:

```
SQL> DELETE FROM Customers WHERE name = 'Teresa';
Query OK, 1 row affected (0.008 sec)
```

Delete a Table

☠ **DANGER ZONE! This cannot be reversed!**

Deleting the whole table works like this:

```
SQL> DROP TABLE Customers;
```

Delete Database

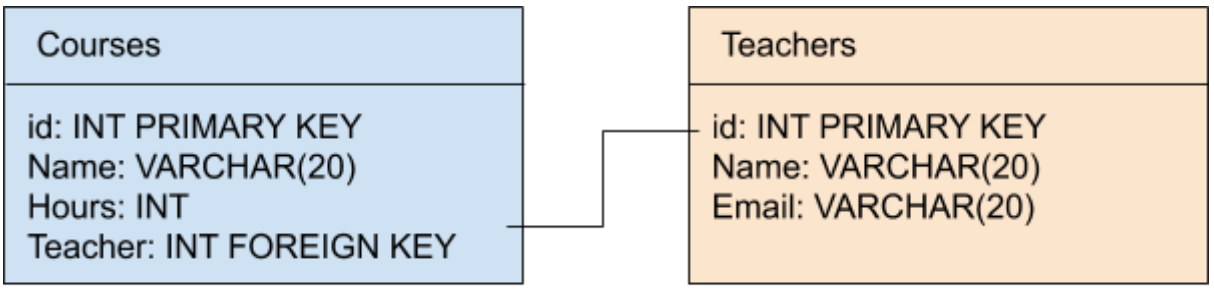
☠ **DANGER ZONE! This cannot be reversed!**

Deleting the whole database works like this:

```
SQL> DROP DATABASE bufg;
Query OK, 0 rows affected (0.39 sec)
```

Relations

You often have a situation where you want to refer from one table to the other. Say, you have a table “Teachers” and a table “Courses”. In the table “Courses” you have a column called “Teacher” where you want to refer to an entry in the table “Teachers”.



This is done with the help of ***foreign keys***.
A foreign key refers to a column in a different table.

In our example, we have a table “Teachers” and a table “Courses”. In the table courses, we have a column called “Teacher”, which refers to the “Teacher” table.

```
SQL> DESCRIBE Teachers;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(20) | YES  |     | NULL    |                |
| email | varchar(20) | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
3 rows in set (0.008 sec)
```

In the table “Courses” we reference the column “id” from the “Teachers” table.

```
SQL> DESCRIBE Courses;
+---------+-------------+------+-----+---------+----------------+
| Field   | Type        | Null | Key | Default | Extra          |
+---------+-------------+------+-----+---------+----------------+
| id      | int(11)     | NO   | PRI | NULL    | auto_increment |
| name    | varchar(20) | YES  |     | NULL    |                |
| hours   | int(11)     | YES  |     | NULL    |                |
| teacher | int(11)     | YES  | MUL | NULL    |                |
+---------+-------------+------+-----+---------+----------------+
4 rows in set (0.012 sec)
```

How was this created? 💡 We can use the **`SHOW CREATE TABLE`** command to have a look at that:

```
SQL> SHOW CREATE TABLE Courses;
+---------+---------------------------------------
| Table   | Create Table
+---------+---------------------------------------
| Courses | CREATE TABLE `Courses` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) DEFAULT NULL,
  `hours` int(11) DEFAULT NULL,
  `teacher` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `teacher` (`teacher`),
  CONSTRAINT `teacher` FOREIGN KEY (`teacher`) REFERENCES `Teachers` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+---------+---------------------------------------
1 row in set (0.004 sec)
```

Here we see in the last two lines how the foreign key was created.

It follows the form:

```
KEY <key_name> (<column_name>),  # key_name: you can choose a name
CONSTRAINT <constraint_name>     # constraint_name: you can choose a name
FOREIGN KEY (<column_name>)      # specify the column in the table
REFERENCES <foreign_table>(<foreign_column>) # specify table(column) from the
                                             # other table that we reference
```

👩🏫 **Note:** Foreign key relations can get complex and may be difficult to understand. But they are a very important topic in relational databases. We suggest you study this topic thoroughly.

📚 **Resource tip:** Tutorial on foreign keys
https://www.mariadbtutorial.com/mariadb-basics/mariadb-foreign-key/

One-to-Many Relationships

In the example above, we created a ***one-to-many*** relationship.

A teacher can teach multiple courses, but a course can only have one teacher.

In relational databases, if you want to model a one-to-many relationship, you do this by **creating a foreign key in the table of the entity of the "one" side**. This means, you put the reference in the table that can only have one reference to the other. In our example, that's the Courses table, because a course can only have one teacher.

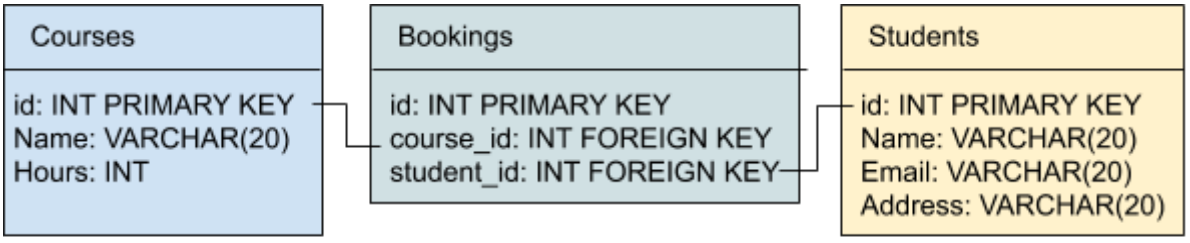
⚠ **Important:** Always put the foreign key into the table of the "one" side of the one-to-many relationship. It does not work on the "many" side.

Many-to-Many Relationships

Sometimes, we have ***many-to-many*** relationships that we want to model with our database.

For example, if you have a table "Courses" and a table "Students", you want to connect these two tables in some way that a student can take many courses, and a course can have many students.

This is done by creating a third table that represents the relationship.



Database Modeling

Database modeling is the art of creating tables and their relations in a relational database. It is a big topic with a lot of complexity. Some universities offer database courses that cover a whole semester. This course cannot cover this, but we want to give you the basics that help you create databases for web applications.

☝ Take your time to think about the tables that you will create. Pay close attention to the relationships between your tables.

✍ Use pen and paper or a whiteboard and draw some sketches first.

▸ Learn about ***Entity Relationship Diagrams*** (ERD).
https://www.youtube.com/watch?v=QpdhBUYk7Kk

> ☝ **The bigger your database gets, the more important is its structure.**

Joins

With ***joins***, you can pull data from different tables. You join tables together temporarily.

This works with the **`JOIN`** keyword.

In our example above, we have a table “Courses” and a table “Teachers”. If we want to have each teacher’s name and the courses they teach, this works like this:

```
SQL> SELECT * FROM Teachers;
+----+----------------+--------------------------------+
| id | name           | email                          |
+----+----------------+--------------------------------+
|  1 | Teresa Holfeld | teresa@hamburgcodingschool.com |
|  2 | Helder Pereira | helder@hamburgcodingschool.com |
|  3 | Thomas Hedeler | thomas@hamburgcodingschool.com |
+----+----------------+--------------------------------+
3 rows in set (0.001 sec)


SQL> SELECT * FROM Courses;
+----+----------------+-------+---------+
| id | name           | hours | teacher |
+----+----------------+-------+---------+
|  1 | Learn to Code  |    24 |       2 |
|  2 | Git and GitHub |     6 |       1 |
|  3 | Databases 1    |     6 |       2 |
|  4 | Databases 2    |     6 |       1 |
```

```
|  5 | Vue.js         |    24 |       3 |
+----+----------------+-------+---------+

5 rows in set (0.000 sec)


SQL> SELECT Courses.name, Teachers.name FROM Courses
  -> JOIN Teachers ON Courses.teacher = Teachers.id;

+----------------+----------------+
| name           | name           |
+----------------+----------------+
| Learn to Code  | Helder Pereira |
| Git and GitHub | Teresa Holfeld |
| Databases 1    | Helder Pereira |
| Databases 2    | Teresa Holfeld |
| Vue.js         | Thomas Hedeler |
+----------------+----------------+

5 rows in set (0.001 sec)
```

Joins follow this pattern:

```
SELECT <column_names>            # Can be * or the pattern Table.Column
FROM <left_table>                # Name of the left table
JOIN <right_table>               # Name of the right table
ON <left_table.column> = <right_table.column>; # Columns to join on
```

📚 **Resource Tips:**
Joining Tables: https://mariadb.com/kb/en/joining-tables-with-join-clauses/
Mode Advanced Joins: https://mariadb.com/kb/en/more-advanced-joins/

Inner Joins, Outer Joins, and even more Joins (optional knowledge)

Inner Joins

If you want to join two tables, but you only want to see the lines that have a match on both tables, you use an **`INNER JOIN`**.

Outer Joins

If you want to see all values, also the ones that don't have a match, use an **`OUTER JOIN`**.

Left Joins

If you want to see all values from the left table, but you don't want to see the values from the right table that don't have a match on the left table, use **`LEFT JOIN`**.

Right Joins

If you want to see all values from the right table, but not the values from the left table that don't have a match on the right table, use **`RIGHT JOIN`**.

> 😅 **Don't worry:** Joins are complicated. Even experienced developers need to look them up all the time. You don't need to know them by heart.
>
> 🔎 **Tip:** Google image search is a useful tool to understand joins quickly.

Client Databases vs. Server Databases

Databases can be used in two different ways:

1. As a **client database** (local).
2. As a **server database** (server-side).

Both are very common. You might be using them already without knowing about them.

Local	Server-side
On the client side: • Web browser • Mobile App • Desktop application	On the server side: • Websites • APIs • Distributed systems (e.g. large customer databases)
Are private	Are public (with restrictions)
Are not shared	Are shared (with specific user rights)
Stays on your machine (phone, laptop, browser)	Accessed via the internet, usually hidden behind a web application
Used mainly for cache	Used for saving data centrally
Usually small amounts of information	Often huge amounts of data

Client-side Databases

Client-side databases are often used without the user noticing it. Their main purpose is to cache data locally. This is, for example, commonly used by mobile apps. Android apps usually use SQLight databases in the background. They are a fast and lightweight way for the app to store information, e.g. about the user profile, or content of views that the user accesses frequently.

Server-side Databases

Many web applications use a database in the background. If you log in to your favorite social network, for example, you access a database remotely.

The server that delivers your HTML page that your browser requested, accesses the database and uses the data to populate the content of the HTML file before sending it back to you.

This is commonly done with a server software, e.g. in PHP, or with a Node.js server application. The web application typically hides the database, so that it is not accessible directly over the internet.

The server application accesses the database via the database's API. This API is called a ***DataBase Management System (DBMS)***.

Document-based Databases

MongoDB

There is a variety of document-based databases. In this course, we will use **MongoDB**.

https://www.mongodb.com/

MongoDB is a commonly used database for web applications. It is very scalable for its purpose. It can be distributed through multiple instances, and servers can run hundreds of nodes with millions of documents in the database at the same time.

Installation

Install MongoDB Community Edition. It is the non-commercial edition, but has all the features we need.

Follow the installation instructions of your operating system:
https://docs.mongodb.com/manual/administration/install-community/

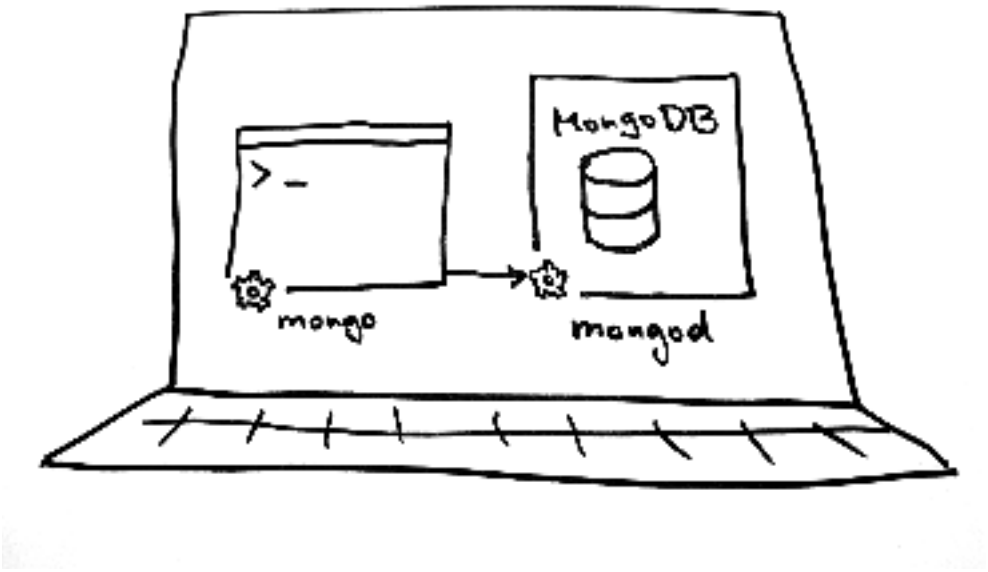
💡 **Tip for Mac Users:** It is super easy to install MongoDB via Homebrew. Once you have homebrew installed, you can type in your terminal:

```
brew tap mongodb/brew
brew install mongodb-community@4.2
```

Command Line Tool

MongoDB has a command line tool. We will mainly use this in this course.

MongoDB is a database software that is running locally on your laptop. With the command line tool, which is also a software that is running in a new shell, we connect to the MongoDB instance.



To run the mongo database software, open your terminal or command prompt and type:

Mac:

```
mongod --config /usr/local/etc/mongod.conf
```

Windows:

```
"C:\Program Files\MongoDB\Server\4.2\bin\mongod.exe" --dbpath="c:\data\db"
```

Then we need to start the mongo command line tool in its shell. Open a new terminal or tab, and run:

Mac:

```
mongo
```

Windows:

```
"C:\Program Files\MongoDB\Server\4.2\bin\mongo.exe"
```

You'll see whether you are connected or not.
If you are inside the mongo shell, you will see:

```
> _
```

You can exit the mongo shell by typing:

```
> exit
```

Show the Database

To see which database you are on, type:

```
> db
```

Switch to Other Database

To switch to a different database, type:

```
> use example
```

The second word, “example”, is the name of the database you are switching to.

> 💡 If the database does not exist, mongo will create it for you.

Collections and Documents

In MongoDB, your data is organized in ***collections***.

A collection is equivalent to a table in relational databases. It is not a table, however.

A ***document*** is a data item stored in a collection. It is the equivalent of a row in a table in relational databases.

In MongoDB, data is stored in ***JSON*** (JavaScript Object Notation) format. This makes it easy to be used with JavaScript later on.

Create a New Document

To create a new document, type:

```
> db.students.insertOne(
...   {
...     "FirstName": "Stefan",
...     "Email": "stefan@gmail.com",
...     "Address": "Hauptstr. 1, 22679 Hamburg"
...   }
... )
```

The first object, **db**, is the database object.
The second, **students**, is the collection that you are using.
You are then calling a method on the collection: **insertOne(...)**. There, you enter your document as JSON.

> 💡 If the collection does not exist yet, mongo will create it for you.

You will see in the output a reply that looks like:

```
{
      "acknowledged" : true,
      "insertedId" : ObjectId("5e11ebb8afe9bc071cc2389e")
}
```

In the second line, mongo tells you the ID for the document that it created automatically. Similar to relational databases, MongoDB requires a unique ID to be able to address each document. You don't need to create an ID field in your document, however, but MongoDB is taking care of that for you.

Insert Multiple Documents

You can insert multiple documents at once with the **`insertMany()`** function:

```
db.students.insertMany([
  {
    "FirstName": "Anna",
    "Email": "anna@hotmail.de",
    "Address": "Rödingsmarkt 4, 22089 Hamburg"
  },
  {
    "FirstName": "Michael",
    "Email": "info@micha.me"
  }
])
```

> 👆 Notice that we wrapped our two documents in an array.

You will see in the acknowledgement that two IDs have been created:

```
{
      "acknowledged" : true,
      "insertedIds" : [
            ObjectId("5e11f5bfafe9bc071cc2389f"),
            ObjectId("5e11f5bfafe9bc071cc238a0")
      ]
}
```

Query All Documents

To show all documents in a collection, type:

```
db.students.find({})
```

The empty JSON **{}** works as a wildmark here, similarly to **SELECT** ***** in SQL.

Query a Specific Document

To query for a specific document, type:

```
db.students.find({ "FirstName": "Anna" })
```

The JSON that we pass to the **`find()`** function is our ***query***.

Query Operators

We can use more complex queries, such as:

```
db.students.find( { "FirstName": { $in: [ "Anna", "Michael" ] } } )
```

Here, we use the query operator **`$in`**. This enables us to query for a list of values.

There are many more query operators. You can look them up here:
https://docs.mongodb.com/manual/tutorial/query-documents/

Update

To update a document, use the **`updateOne()`** function.

```
db.students.updateOne(
   { "FirstName": "Michael" },
   {
     $set: { "Address": "Borselstraße 7, 22765 Hamburg" }
   }
)
```

Similarly, you can use **`updateMany()`** with a query operator.
You can look it up here:
https://docs.mongodb.com/manual/tutorial/update-documents/

Delete All Documents Matching

You can delete all documents that match a query like this:

```
db.students.deleteMany({ "FirstName": "Anna" })
```

Delete the First Document that Matches

If you want to delete only the first match, use:

```
db.students.deleteOne({ "FirstName": "Anna" })
```

Delete All

To delete all documents in a collection, you can use the wildmark query operator **{}**.

```
db.students.deleteMany({})
```

Methodology for Finding the Right Query

Finding the right query can be difficult, especially on the command line. To find the right query, you can follow the following steps:

1) **Browse the dataset and familiarize yourself with it.**
2) **Identify the field you are interested in.**
3) **Identify the query operator or the syntax you need.**
    - Check the documentation
    - Search the web for examples
4) **Try it out.**
   (Trial and error)

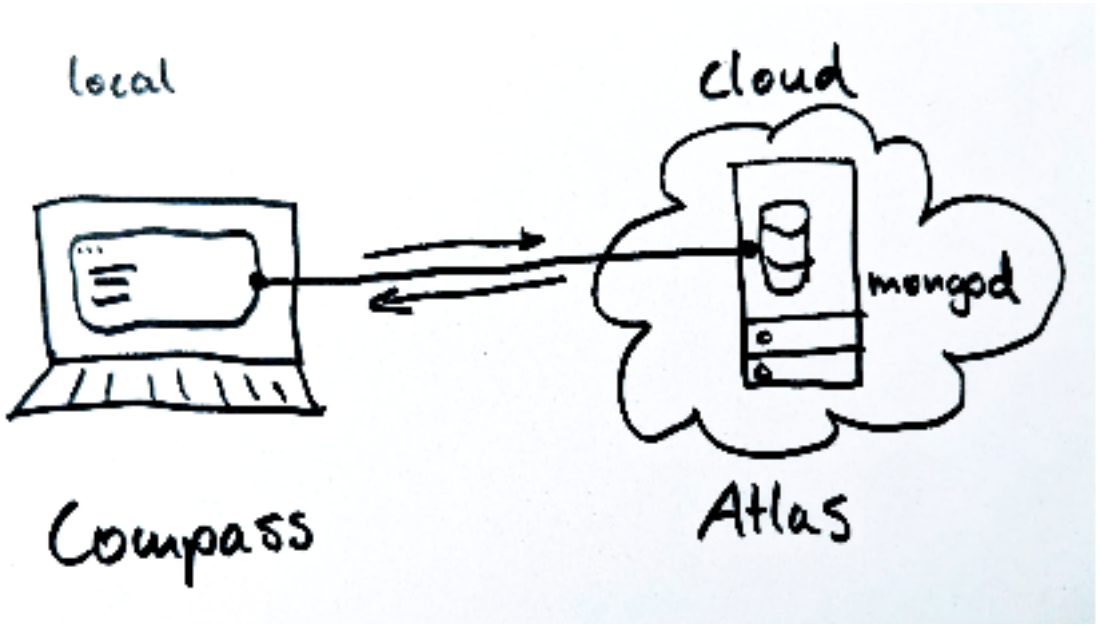
😇 Even experienced developers use Google and Stack Overflow in step 3 all the time.

MongoDB Atlas and Compass

MongoDB provides a cloud database called Atlas.

In addition, there is an application that you can install on your laptop that is called Compass. Compass is giving you a GUI (Graphical User Interface) that makes it a bit easier to work with the database.

If you create your own Atlas in the MongoDB cloud, you can connect your Compass application to it.



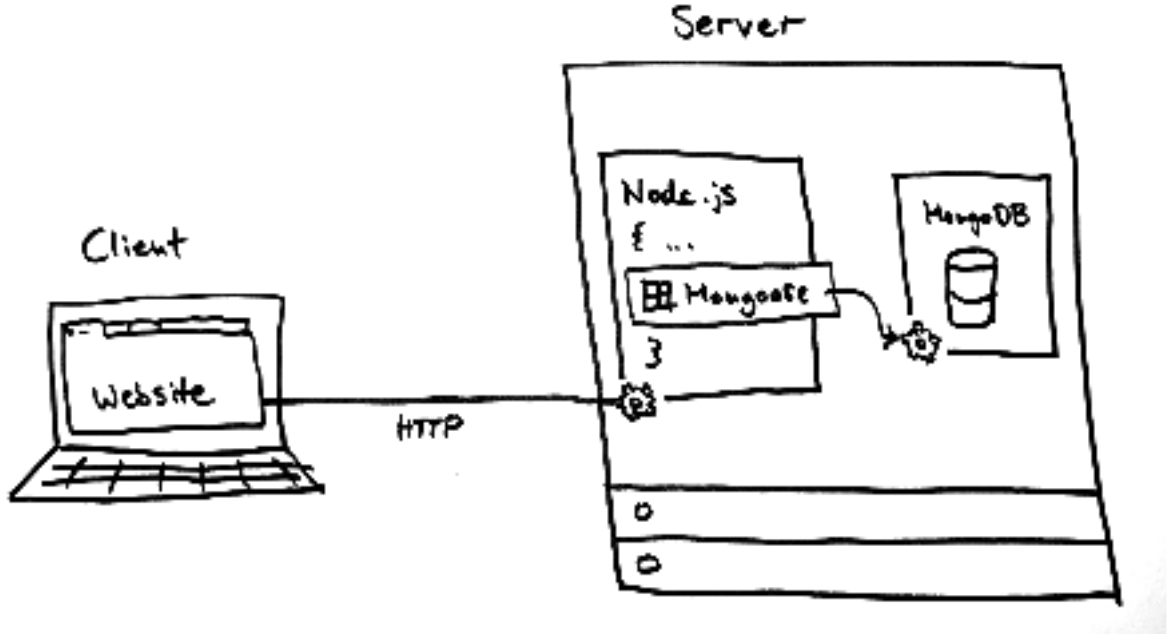
MongoDB With Node

MongoDB is made to be used with JavaScript:

- It stores objects in JSON format.
- It provides interfaces for JavaScript.

This is the reason why MongoDB is widely used for web applications.

If it is used in a web application, **mongod** will run the database on a server.



A server application, e.g. a Node.js application, is running on the server as well. This application is connected to the internet and can receive and respond to HTTP requests.

The Node application in this scenario will then connect to the running instance of MongoDB. For that, there's the package mongoose.

On the other side, the web browser connects to the Node application via HTTP.

> 💡 Both the Node.js application and the mongoose / mongodb interface implement **CRUD - Create, Read, Update, Delete**.

Comparison MongoDB to Relational Databases

What makes it so different from relational databases?

- Objects can all be different from another
- There are no empty fields like in a table
- Relational databases are better in securing data integrity.
- In MongoDB you have no hard-wired references, like foreign keys in SQL.

What are the advantages of MongoDB over relational databases?

- They are much faster for most operations.
- They are more scalable.
- We can process JSON directly in JavaScript. 😉

When does it make sense to use which type of database?

- Document-based databases are good at dealing with large amounts of the same data, e.g. if you have large collections with a large amount of documents.
- Relational databases are the better choice if you need to model a lot of relations, because it brings features like foreign keys and joins.

Glossary

`Database`	A software that stores and manages data in an ordered way.
`Relational database`	A type of database where data is stored in tables. Usually uses an SQL language.
`Document-based database`	A type of database where data is stored in documents. One document is one data entry.
`SQL - Structured Query Language`	A language used to query data in relational databases.
`NoSQL`	A class of databases that is different from relational databases. Includes types of databases like document-based, graph or key-value databases.
`CRUD: Create, Read, Update, Delete`	The set of operations typically supported by databases and APIs.
`DBMS - DataBase Management System`	The software that runs a database.
`MariaDB`	A DBMS for relational databases. The successor of the well-known MySQL.
`table`	In relational databases, data is stored in tables, where each row is a data entry.
`attribute`	A column in a table.
`row`	A data entry in a table.
`data type`	The type of data in a column, e.g. integer, boolean, varchar.

`signed`	A data type specification. Specifies that a number that can have a minus sign, so the number can be positive or negative.
`unsigned`	A data type specification. Specifies that a number cannot have a minus sign, so only positive numbers are allowed.
`not null`	A data type specification. Specifies that this attribute cannot be empty.
`auto_increment`	A data type specification. Specifies that the attribute will be inserted automatically with the next increment. Can only be used with numbers.
`Primary key`	The column that is used for uniquely identifying each data entry.
`Foreign key`	Specifies that a column contains the primary key of a data entry in a different table.
`constraint`	A specification for a column for data integrity, e.g. a foreign key referencing to a different table.
`Entity`	The thing that is represented by a table.
`ER - Entity Relationship`	The relationship between tables.
`ERD - Entity Relationship Diagram`	A diagram that visualises the relationships between entities.
`UML - Unified Modeling Language`	A standard for diagrams. Contains a standard of how to draw ERDs.
`One-to-many relationship`	A relationship, where e.g. an Owner can have multiple Cars, but a Car can have only one Owner.

`Many-to-many relationship`	A relationship, where e.g. a Student can take multiple Courses, and a Course can have multiple Students.
`Join`	An SQL command that joins data from one table with data from another table and displays the result as a temporary table.
`Client database`	A database that is running in a client application, e.g. a mobile app, a website or a desktop application. Usually used for caching data.
`Server database`	A database on a server, typically used and hidden behind a web application.
`MongoDB`	A document-based database software.
`mongod`	The database software of MongoDB.
`mongo`	The command line interface of MongoDB that connects to `mongod`.
`collection`	In document-based databases, data is sorted in collections. A collection contains data of the same form. Equivalent to tables in relational databases.
`document`	A document is a data entry in relational databases. Equivalent to rows in relational databases.
`query operator`	An operator for specifying or composing queries. Examples in mongo: `$in, $lt, $gt`
`mongoose`	A node package that provides an interface for connecting to MongoDB.

Useful Links

MariaDB Knowledge Base: https://mariadb.com/kb/en/

W3school SQL tutorial: https://www.w3schools.com/sql/

Public examples to try out SQL:
https://www.sachsen.schule/~terra2014/sql_abfragen.php
https://sqlzoo.net/wiki/SELECT_basics

MariaDB Foreign Key tutorial:
https://www.mariadbtutorial.com/mariadb-basics/mariadb-foreign-key/

ER Diagrams Youtube tutorial: https://www.youtube.com/watch?v=QpdhBUYk7Kk

Joining Tables: https://mariadb.com/kb/en/joining-tables-with-join-clauses/

Mode Advanced Joins: https://mariadb.com/kb/en/more-advanced-joins/

MongoDB Documentation: https://docs.mongodb.com/manual/

MongoDB Courses: https://university.mongodb.com/

MongoDB tutorial: https://www.guru99.com/what-is-mongodb.html

MongoDB Cheat Sheet:
https://gist.github.com/bradtraversy/f407d642bdc3b31681bc7e56d95485b6

Mongoose: https://mongoosejs.com/

Node with MongoDB Tutorial: https://www.youtube.com/watch?v=4yqu8YF29cU