Database technologies

Programme	Advanced bachelor of bioinformatics
Course unit	BIT 05 Database technologies
Semester	1
Lecturer	Mr. Anckaert
Role	Informatics

howest
we develop people

Table of Contents

1 Introduction

1.1 Module organization

Code + name course unit	BIT05 Database technologies
Semester (+ period)	1B
Education language	English
Course unit responsible	Jasper Anckaert (JA)
Role	Informatician
Level	Beginner
Credits	5
# Contact hours (CH)	48
Factor study load	30

1.1.1 General objective

On a daily basis people are confronted with databases, including in biological research, where nowadays they could not be missed. Exchanging data and using different research results for further analysis requires structured (data)storage. Next to publicly accessible databases, it is an asset that a bioinformatician can assemble a decent database system on his own. The used system is highly dependable of the further applications.

In the course unit BIT05 Database technologies, first the importance and basic structure of a database system is being taught. A separation is made between relational and NoSQL system, this is done by working with MySQL and MongoDB. In both systems, the student learns how to assemble a database, fill it with relevant data and retrieve, adjust and delete this data. Next to this, correct normalisation of data to construct a decent database model is being learned. Moreover, the principles to retrieve data from public databases like the Ensembl REST API is also taught. Finally, the student learns how to subject all created files to version control by using the revision software git.

1.1.2 Learning outcomes, VKS 6 descriptors, behavioural indicators, structural decree descriptors

Learning Outcomes (LO)	☒ BIT0100 The advanced bachelor in bioinformatics autonomously clarifies and solves a biological problem by selecting a relevant programming language and by applying it efficiently to develop an own program ☒ BIT0300 The advanced bachelor in bioinformatics manages, processes and retrieves biological complex data in a user-friendly

	manner by using existing or still to be developed database or software structures ☒ BIT0500 The advanced bachelor in bioinformatics chooses, depending on the given biological problem, the relevant software and efficiently uses this software to clarify and solve the given biological problem ☒ BIT0600 The advanced bachelor in bioinformatics autonomously develops a multidisciplinary perspective on bioinformatics which joins biological and computational skills in practical applications within an authentic context ☒ BIT0700 The advanced bachelor in bioinformatics reports transparent and correct about research data and analysis results with the proper technical terminology. He/she cooperates constructively, respectfully and in a team-oriented way in an intraprofessional and interprofessional context
Descriptors VKS 6 per LO	☒ Critically evaluating (theoretical) knowledge and insights from a specific domain and combining them ☒ Applying complex and specialized skills, linked to research outcomes ☒ Collecting and interpreting data and applying selected methods and resources innovatively to solve non-familiar, complex problems ☒ Acting in complex and specialised contexts ☒ Functioning completely autonomous, with showing a great deal of initiative ☒ Taking co-responsibility for generating collective results
Behaviour indicators (BI)	☒ BIT0101 The advanced bachelor in bioinformatics applies relevant programming techniques to process (biological) information in an automated manner ☒ BIT0102 The advanced bachelor in bioinformatics programmes as efficient as possible ☒ BIT0301 The advanced bachelor in bioinformatics autonomously selects and deposits biological data from of or in a database, whether or not in an automated manner ☒ BIT0302 The advanced bachelor in bioinformatics develops new database structures to manage existing or acquired biological data ☒ BIT0501 The advanced bachelor in bioinformatics critically evaluates the possibilities to analyse the given or acquired, whether or not in an automated manner, (biological) data ☒ BIT0602 The advanced bachelor in bioinformatics analyses, synthesizes and harmonizes inferences from different disciplines up to a coherent and coordinated level ☒ BIT0701 The advanced bachelor in bioinformatics reports results and information according to the standards valid in the work field
Descriptors structural decree per BI	☒ AC1 thinking and reasoning skills ☒ AC2 acquiring and processing information ☒ AC3 the ability to reflect critically

	☒ AC4 to work project-based ☒ AC5 creativity ☒ AC7 the ability to communicate information, ideas, problems and solutions to both professionals and non-professionals. ☒ ABC1 to be able to work team-oriented ☒ ABC2 to be able to work solution oriented in a way of being able to define independently and analyse complex problem situations in the professional practice and to develop and apply useful solutions strategies (being able to formulate advice)

1.1.3 Learning goals

a) Knowledge

The student gives the different types of public accessible bioinformatics databases and describes some alternative data retrieving methods for these databases (BIT0501)

The student describes the structure of the most common file formats within public databases like the structure of a GFF and a GenBank file (BIT0701)

The student describes the basic concepts and elements concerning databases and the submersive technology (BIT0302)

- The student gives the general structure of a database system
- The student explains the different levels of a database system

The student describes the general features of a database management system and enumerates the differences between a SQL and NoSQL system (BIT0101)

The student enumerates the different components of Structured Query Language (BIT0102, BIT0301)

The student names the different column types within a relational database (BIT0302)

b) Insights

The student selects the appropriate data base system based upon a problem (BIT0101)

The student constructs a relational database model and determines whether a relational or non-relational system is being used (BIT0101)

The student creates a database based upon a series of information (BIT0301, BIT0302)

- The student constructs a database with correlating tables
- The student completes the tables in a database with relevant data

c) Applying knowledge and insights

The student performs search commands within a database databank (BIT0102, BIT0301, BIT0701)

- The student retrieves data from a relational database
- The student retrieves data from a NoSQL database

The student normalizes data to construct a database model and applies correctly the normalisation rules (BIT0102, BIT0501)

The student subjects his files to version control and saves the files in a GIT repository (BIT0101)

The student queries online databases based upon a problem (BIT0101, BIT0301, BIT0602, BIT0701)

- The student retrieves data from the correct bio informatics database
- The student retrieves data using Ensembl REST API

1.1.4 Study material

Study material (learning tool)	Obligatory ☒ presentation ☒ syllabus ☒ software: MySQL; MySQL Workbench; MongoDB; Robo3T
Location	☒ campus

Type of learning goal	☒ Knowledge ☒ Insights ☒ Applying knowledge and insights
Type of evaluation	☒ Knowledge exam ☒ Insight exam ☒ Open book exam ☒ Skills exam
Criteria	☒ Knowledge criteria ☒ Answer key
Evaluator	☒ External experts
When	☒ End of learning process
Why	☒ Summative
Medium	☒ Written ☒ Laptop

1.1.5 Evaluation

End of learning process: exam period	
Type of evaluation	Medium
Knowledge and insight exam	Written
Skills exam	Laptop (Open book exam)

Calculation of course unit score	100% periodic evaluation (second exam chance possible)
Final exam for LR?	☒ No

1.2 An introduction to databases

Moore's law

Moore's law is the observation that the number of transistors in a dense integrated circuit doubles about every two years. A similar conclusion could be drawn for other measures in informatics such as RAM and storage.

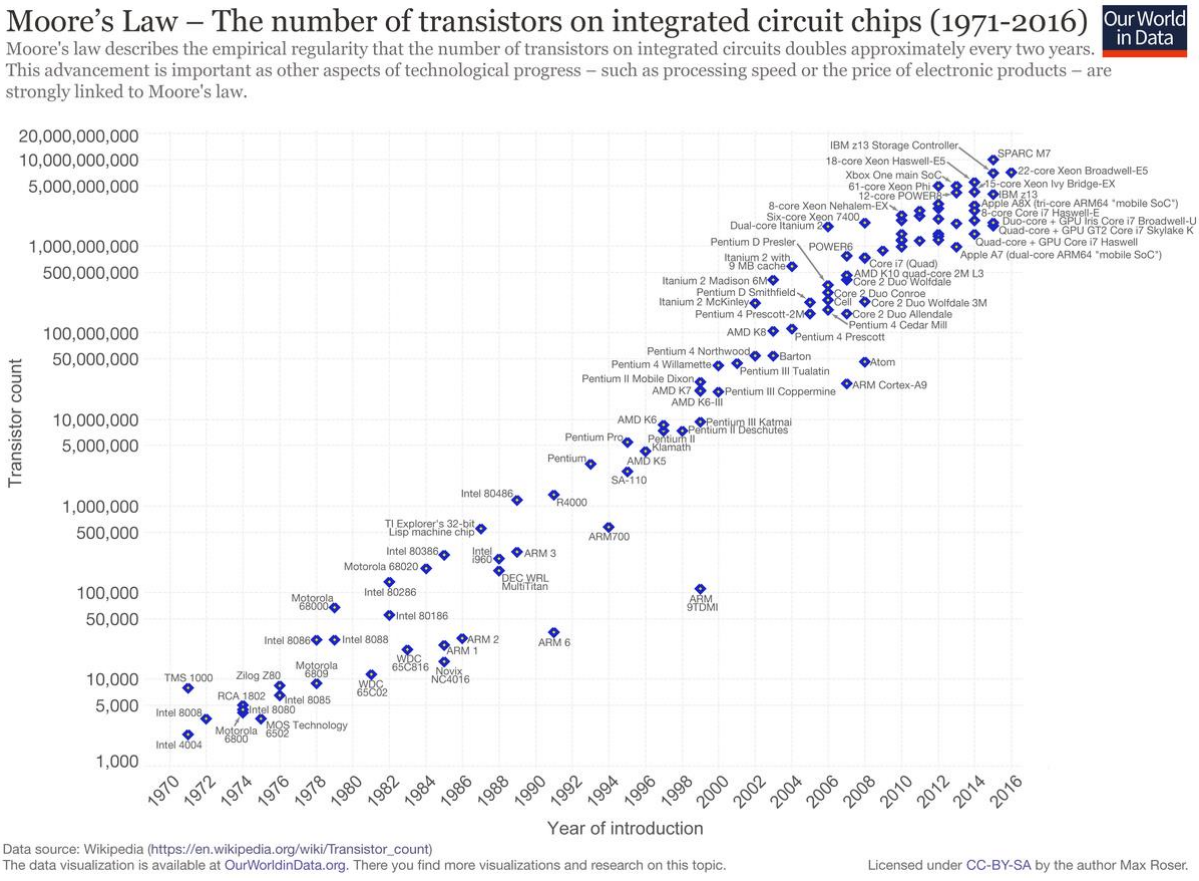

Figure 1: Moore's law (Wikipedia)

Technological advances combined with a lower production cost have led to an exponential growth in amount of processable data. Such an increase in data leads to the need for efficient and structured storage.

1.2.1 What is a database?

A database is a mechanism used to store information electronically. It is a collection of data that can consist of numbers, dates, text, …. A database is used to store data in a structured fashion. It allows efficient interaction with the data. Databases are used everywhere, from your local warehouse (customer card) to your phone (contacts) and everywhere on the world wide web (Facebook, Twitter, …).

In science and research, databases are frequently used. They are classified based on the kind of data they contain (sequences, sequence annotation, experimental data, …) and are often provided by large initiatives (Ensembl, UCSC, Encode project, 1000 genomes project, …).

1.2.2 What is a database system?

A full database system consists of four main parts. The hardware part is a storage media. It possesses processing power (CPU) and temporary memory (RAM) and can range from a smartphone to a high-performance cluster. The second part is the data itself. A low amount of redundancy is being pursued. As mentioned earlier, data is stored in a structured fashion and can be single of multi-user (shared). Another part of a database system is the software (DBMS). One or more users make up the final part of a database system. For each user, access rights are set to determine the amount of actions a user can perform on the database. It goes without saying that the database administrator should and will have more rights than a normal end user.

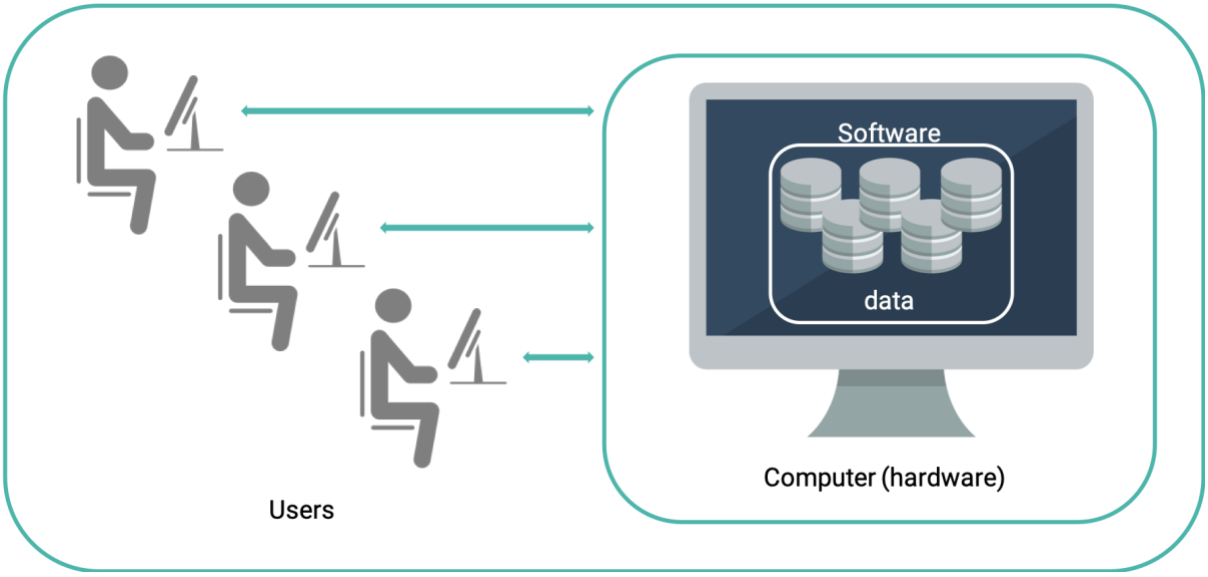

Figure 2: Database system

A good database system has a sufficient amount of storage and is easily accessible through a centralised system. It is secure against non-users. Each user should have his/her own specific set of rights. Furthermore, it should have a clear structure. To allow for easy maintenance, input should be controlled and redundancy reduced to the bare minimum.

Data abstraction

In a typical database, there are mainly 3 levels of data abstraction.

Physical: This is the lowest level of data abstraction. It tells us how the data is actually stored in memory. The access methods like sequential or random access and file organization methods like B+ trees, hashing used for the same. Usability, size of memory, and the number of times the records are factors which we need to know while designing the database.

Suppose we need to store the details of an employee. Blocks of storage and the amount of memory used for these purposes is kept hidden from the user.

Logical: This level comprises of the information that is actually stored in the database in the form of tables. It also stores the relationship among the data entities in relatively simple structures. At this level, the information available to the user at the view level is unknown.
We can store the various attributes of an employee and relationships, e.g. with the manager can also be stored.

View: This is the highest level of abstraction. Only a part of the actual database is viewed by the users. This level exists to ease the accessibility of the database by an individual user. Users view data in the form of rows and columns. Tables and relations are used to store data. Multiple views of the same database may exist. Users can just view the data and interact with the database, storage and implementation details are hidden from them.

Data independence

Physical level data independence: It refers to the characteristic of being able to modify the physical schema without any alterations to the conceptual or logical schema, done for optimisation purposes, e.g., Conceptual structure of the database would not be affected by any change in storage size of the database system server. Changing from sequential to random access files is one such example. These alterations or modifications to the physical structure may include:

- Utilising new storage devices.
- Modifying data structures used for storage.
- Altering indexes or using alternative file organisation techniques etc.

Logical level data independence: It refers characteristic of being able to modify the logical schema without affecting the external schema or application program. The user view of the data would not be affected by any changes to the conceptual view of the data. These changes may include insertion or deletion of attributes, altering table structures entities or relationships to the logical schema etc.

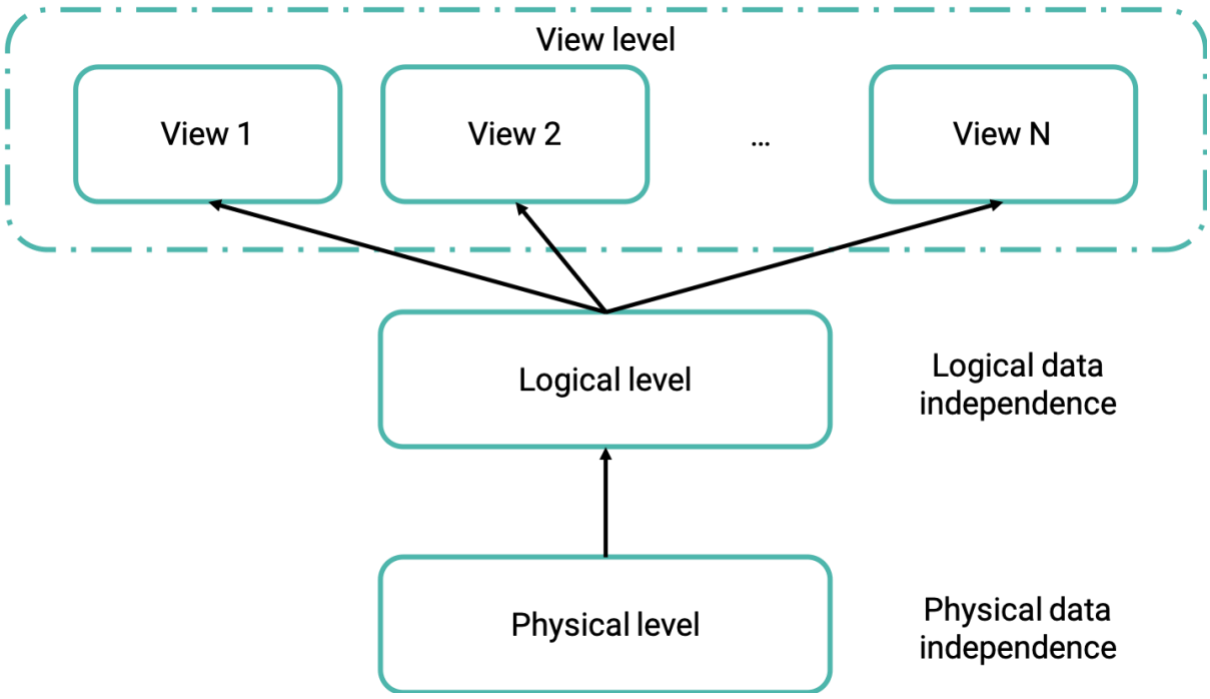

Figure 3: Data abstraction and independence

1.2.3 What is a database management system?

In order to manage a database, a database management system (DBMS) is used. This is a computer software application that interacts with users, other applications, and the database itself to capture and analyse data. It shields users from hardware and storage details and is the most important software component, though not the only one (development tools, …). A database management system is used for data creation, storage, retrieval and manipulation. Authentication and authorization of the database are regulated by the DBMS.

Some examples of database management systems are:

- MySQL
- Oracle
- Access
- NoSQL systems: MongoDB, CouchDB, …
- NewSQL systems: Clustrix, VoltDB, …

2 Relational databases

A typical relational database has a rigid structure and consist of two-dimensional tables. These tables are built out of columns or fields and rows or records. A widely used (and simplified) example of this is a Microsoft Excel worksheet.

The entire collection of model object (entities) and their relationships make up a relational database. In the example *a store sells products to customers*, *customers* and *products* are the entities, both with their own set of attributes (name, address, price, …). *Sale* would then be the relationship with its own set of attributes (quantity, timestamp, …) as well.

In order to manage a relational database, a relational database management system (RDBMS) is used. This enforces data integrity and referential integrity by honouring the constraints set on both columns and rows.

There are commercial (Oracle, MS SQL Server, …) and open-source (MySQL, PostgreSQL, SQLite, …) options for a RDBMS. In this course unit we will make use of the open-source software package MySQL because it is free, but more importantly, because it is widely used and well documented.

2.1 MySQL

MySQL is (one of) the most used relational database management systems. It is open-source and free of charge, although paying versions exist as well. Large companies such as WordPress, Twitter, Facebook and many others make use of (a version of) MySQL.

2.1.1 Installation of MySQL

MySQL can be installed on Windows, Mac and Linux. For each of these operating systems, an installation package is provided on https://dev.mysql.com/downloads/

For continuity reasons (use of Linux in other course units), from now on, we will work with the Linux (Fedora 28) version of MySQL.

All the information to install MySQL on your Linux system can be found on https://dev.mysql.com/doc/refman/8.0/en/linux-installation.html. Be sure to choose the instructions for the correct version of your operating system, e.g. Fedora 28.

After installing the MySQL server on your computer, check if it is running correctly. If not, start it manually. For your convenience change the automatically set temporary password to one of your own choosing. As an extra safety measure, we will secure the installation and limit the access to our localhost only.

```
sudo dnf install https://dev.mysql.com/get/mysql80-community-release-fc28-1.noarch.rpm
sudo yum install mysql-community-server.x86_64
sudo service mysqld start
sudo service mysqld status
```

→ Add to global config file (*/etc/mysql/my.cnf*)

```
[mysqld]
bind-address = 127.0.0.1
```

2.1.2 The MySQL monitor

In order to connect with your recently installed MySQL server, the MySQL monitor is used. This is a command line interface for MySQL. Depending on the user settings, certain options need to be giving to the mysql command.

```
$ mysql [options] [database]
        -u uname | --user=name
                default: UNIX account
        -p [pwd] | --password[=pwd]
                default: <none>
        -h hname | --host=name
                default: localhost
        -P prt | --port=prt
                default: 3306
```

Take in mind that database users and OS users are completely independent from each other. If no user is specified when connecting to the monitor, the OS user is taken. Furthermore, a database superadmin is set by default as root@localhost. This user has all rights, including dropping databases. Generally, it is not a good idea to always connect to the monitor as root.

When logging in as root, the user is allowed to create other users, each with its own specific set of rights. This is done by granting privileges to a user on a certain or all tables in one or more databases. The database(s) and table(s) do not have to exist at the time of the rights granting.

```
mysql> CREATE USER 'newuser'@'localhost' IDENTIFIED BY 'password';
mysql> GRANT prv ON db.tbl TO user@host;
```

So as to avoid retyping of all your credentials, especially your password with every connection attempt to the MySQL monitor, you can create an options file on your OS that contains the necessary login information. This file is typically located in your home directory and is named .my.cnf. To make sure no other OS users can exploit this file, it will need to be protected by setting file permissions to 600. The options file contains *key=value* pairs in *[sections]*. All these pairs are then provided to the mysql command as parameters.

Example options file

```
[client]
password=pwd
user=username
```

2.1.3 SQL

SQL stands for Structured Query Language and is the standard language used for accessing and manipulating databases. It consists of statements and queries. Statements are any text the database engine can recognise as a valid command. Queries are all statements that return a possibly empty record set. There are four type of statements:

- Data Definition Language (DDL) statements: used to design a database
  CREATE TABLE, DROP DATABASE
- Data Manipulation Language (DML) statements: used to manage data in a database
  - **C**REATE: add new data to the database
  - **R**EAD: collect data from the database
  - **U**PDATE: alter data in the database
  - **D**ELETE: remove data from the database

  INSERT, SELECT, UPDATE, DELETE
- Data Control Language (DCL) statements: used to manage database rights
  GRANT, REVOKE
- Transaction Control Language (TCL) statements: used to manage DML tasks (group, undo, …)

A typical SQL statement is built out of clauses containing expressions and/or predicates. An expression produces scalar values, or tables consisting of columns and rows of data. In turn, predicates are conditions or Booleans, used to limit the effects of statements and queries. A statement always ends with a semicolon (;). For readability purposes, whitespace can be used, but it is ignored when interpreting the statement.

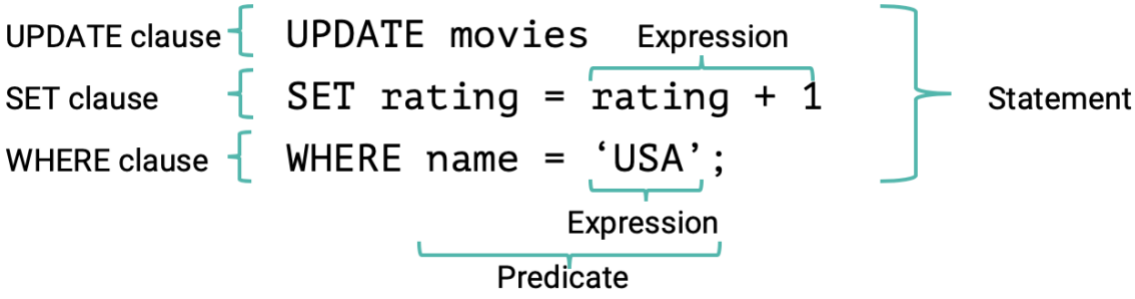

Figure 4: SQL statement

2.1.4 The MySQL monitor – episode 2

There are several ways to execute SQL statements using the MySQL monitor. A first way to do this is by entering the statement directly into the monitor after connecting. This is could the interactive way.

Another way is by giving the statement to the mysql command as an option. A final way, often used to fill or create a database, is by piping a statement file to the mysql command.

```
$ mysql [database]
mysql> stmt

$ mysql [database] –e 'stmt'

$ mysql [database] < stmt_file
$ cat stmt_file | mysql [database]
```

Only database users with significant privileges can create databases. This is either done from the command line with the mysqladmin command or from within the MySQL monitor with a CREATE DATABASE statement.

```
$ mysqladmin [opt] create dbname
mysql> create database dbname
```

A single MySQL service can have multiple databases. A newly created database can be empty or filled with data from the get go, depending on the executed statements in the monitor or the provided statements in a statement file. A particular database can have multiple tables, and a particular table can have multiple columns or fields.

```
SHOW databases;

USE db;
SHOW tables;

SHOW columns FROM tbl;
SHOW create table tbl;
```

A MySQL database is a collection of tables. Each table is a set of columns with specific types. Keep in mind that each row of the table should be in the same format and there can be only one value per field.

Student_number	Name	Last_name	Birthdate	Sex
0293826	John	Doe	1991-10-02	M
0293749	Mel	Trotter	1991-04-11	V
0328273	Bill	Schuette	1990-12-01	M

Figure 5: MySQL table

2.1.5 Column types and constraints

Column types

Each table in a database has one or more columns. These columns have a specific type that indicates the type of data (number, text, …) in that column.

An integer or int(x) is a whole number. It can be negative (SIGNED) or only positive (UNSIGNED). Next to this type, numbers can also be float or double, representing floating point (real) numbers.

A char(x) or varchar(x) defines a string with a certain number of characters. The x stands for the maximum number of characters in the string. A varchar will be more efficient in storage because it only uses the amount of space needed for the length of the string, whereas a char will reserve space for the maximum possible length of the string. Text that are not queried often or do not have to be searchable the text type can be used. For binary data, like images, blob is used.

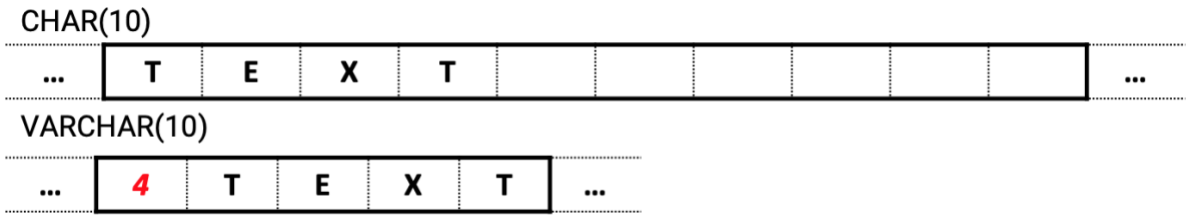

Figure 6: CHAR vs. VARCHAR

If there is only a (short) list of permitted values, the enum type is used. This is very efficient in storage. It also reduced the number of errors for that field, because values are pre-set and typos are not allowed.

To store dates the date or datetime type is used. The default format of this type is YYYY-MM-DD (HH:MM:SS). This is not the same as timestamp, because timestamp can't handle dates before 1970 and after 2038.

Constraints

On top of the column types, other constraints can be put on columns. In order for the DBMS to distinguish separate rows, every row has to be unique. To obtain this, a PRIMARY KEY is set. This is a column, or a set of columns that makes sure every row is unique. Usually, it is the first column and often the int type is chosen together with the auto_increment property, which adds 1 to each value automatically. A table can have only one primary key.

Next to the PRIMARY KEY, a table can contain a FOREIGN KEY as well. It has to have the same constraints as its referenced column. With this foreign key, the possibility to secure removal of linked data is added.

The UNIQUE constraint makes sure all values (or combinations) are unique. When setting the NOT NULL constraint, the database does not accept empty fields when adding data. The DEFAULT constraint sets a default value for a certain field.

2.1.6 INSERT-statement

Usually, after creating a new table in a database, it does not contain data. In order to fill a table with data, the INSERT-statement is used. Not all columns of the table need to be included in the query.

Unmentioned columns are given the default value for that column, empty columns get the NULL-value. For columns with a NOT NULL constraint, a value is required. When writing an INSERT-statement, all strings should be put between quotes, number do not follow that rule.

INSERT INTO *tbl* (*col1*, *col2*) VALUES (*val1*, *val2*);

INSERT INTO students (student_number, name, last_name) VALUES (2654897, 'Glenn', 'Walker');

2.1.7 SELECT-statement

To retrieve rows from a table in a database, the SELECT-statement is used. The user can define a specific set of columns (separated by a comma) to select data from, or use the asterisk (*) as a wildcard for all columns.

SELECT *columns* FROM *tbl;*

SELECT * FROM modorg;

SELECT genus, species FROM modorg;

When using a SELECT-statement, the data is displayed in no particular order. To sort the output the ORDER BY clause is used. This clause can sort the output by a certain column in both ascending (default) or descending order.

SELECT *columns* FROM *tbl* ORDER BY *col1* [*asc*|*desc*] [, *col2* [*asc*|*desc*] ...]*;*

With the SELECT not only existing columns, but also calculated columns (within one row) can be retrieved from a table. There are a lot of functions and operators readily available.

- Numbers
  - Operators: +, -, *, /, %
  - Functions: sqrt(x), power(x, y), round(x), ceil(x), floor(x), …
- Strings
  - Functions: length(s), concat(s1, s2, …), upper(s), lower(s), reverse(s), …
- Dates
  - Functions: now(), year(d), dayofmonth(d), hour(d), …

SELECT 6*7;

SELECT concat(class, " ", genus) FROM modorg;

SELECT now();

The output of a query shows the column names as stated in the SELECT-statement. For calculated columns this can be quite long and unreadable. In order to simplify this, columns can be renamed and the newly created alia can also be used in the ORDER BY clause.

SELECT *col* [AS] *alias* …

Off course, most of the time a user wants to filter the output based on one or more specific conditions. To this purpose, the WHERE clause is used. In this clause, one or more conditions can be specified, combined with AND, OR, NOT or XOR. Only the row(s) for which the conditions evaluates TRUE are selected. It is not possible to use column aliases in the condition(s). To set these conditions, numerical comparison operators can be used. Next to these there are also string comparison operators. Off course, NULL-values require special attention and can be selected or filtered out using the IS (NOT) NULL predicate. There is the possibility to substitute NULL-values directly from the SELECT-statement with the ifnull(col, value) function. This function returns col if col is NOT NULL and value if col is NULL. To combine several conditions, Boolean logic is used.

- not x TRUE if x is FALSE
- x and y TRUE if both x and y are TRUE
- x or y TRUE if x or y is TRUE, or both
- x xor y TRUE if x or y is TRUE, but not both (exclusive or)

SELECT *columns* FROM *tbl* WHERE *condition(s)* [ORDER BY *sortcol*];

SELECT … WHERE *col* IS NULL;
SELECT … WHERE *col* IS NOT NULL;

SELECT ifnull(*col*, *value*) …

To eliminate duplicate rows from the result set, the keyword DISTINCT is added to a column or a set of columns in the query. Each combination of the columns is then set to be unique.

SELECT DISTINCT(*cols*) FROM …

If a user wants to limit the number of result rows, the keyword LIMIT is used. This limits the results set to a predefined number of rows. If an OFFSET is given, the first x number of rows are skipped. This keyword is mostly used in combination with an ORDER BY clause.

SELECT … LIMIT *n* [OFFSET *r*];

Aggregation

Queries are concentrated on particular rows. But what about calculating a single result across multiple rows? SQL allows you to specify criteria to group rows together, calculate a single value per group and filter the grouped data. This is called aggregation. Fortunately, SQL provides a number of predefined functions for this: count(col), count(*), sum(col), min(col), max(col), avg(col), ….

To sort data into groups for aggregation purposes, one can use the GROUP BY clause. All rows with the same value for the specified column are grouped. For each group, the aggregate function is then calculated. Filtering the results based on the results of the aggregate functions can be done with a HAVING clause. Column aliases can be used in this clause. In the query, we type this before ORDER BY.

SELECT [*col*,] *aggregatefunctions* FROM *src* [WHERE *cond*] GROUP BY *col* [ORDER BY …];

SELECT [*col*,] *aggregatefunctions* FROM *src* [WHERE *cond1*] GROUP BY *col* HAVING *cond2* [ORDER BY …];

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [PROCEDURE procedure_name(argument_list)]
    [INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
      | INTO DUMPFILE 'file_name'
      | INTO var_name [, var_name]]
    [FOR UPDATE | LOCK IN SHARE MODE]]
```

Figure 7: SELECT-statement

A SELECT-statement follows a certain execution order. Keep this in mind when executing a query.

1. FROM – input columns are determined
2. WHERE – input columns are filtered
3. GROUP BY – sorting and grouping of filtered input
4. Aggregation functions are calculated
5. HAVING – aggregation functions are filtered
6. ORDER BY – output is sorted
7. LIMIT/OFFSET – output is chopped

2.1.8 Joins

The SELECT-statement is very useful to retrieve data from a single table. But what if a database has multiple tables? Remember that a relational database models entities and their relationships. In order to avoid the storage of redundant information, which is a waste of space and error prone, different entities should be put in different tables. So as to combine information across different tables, a FOREIGN KEY (FK) is added to one of the linked tables. Keep in mind that the data types of the linked columns have to be equals. The FK is typically the PRIMARY KEY (PK) of another table. These FKs can link 2 tables in several ways:

- 1:n one-to-many relationship

A relationship between two entities A and B in which an element of A may be linked to many elements of B, but an element of B is linked to only one element of A

- n:m many-to-many relationship
  A relationship between two entities A and B in which an element of A may be linked to many elements of B, and vice versa. This type of relationship is only possible with a cross-reference table (xref table)

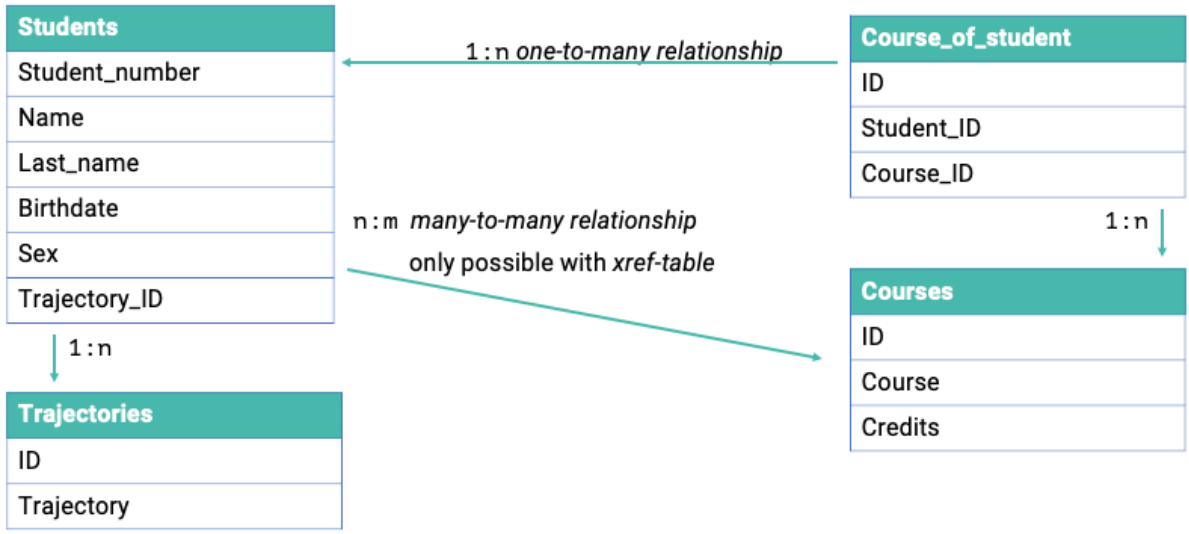

Figure 8: Relationships between tables

These relationships are important when creating a database model or Entity Relationship Diagram (ERD), a graphical representation of a database.

When querying multiple tales in a database at once, the database server generates all possible combinations, making the results comparable to a Cartesian product. Without filtering, these results would be quite uninteresting. That is why a JOIN is used as a replacement for a filtered Cartesian product.

SELECT * FROM modorg, class WHERE modorg.class_id = class.id;
SELECT * FROM modorg [INNER] JOIN class ON modorg.class_id = class.id;

Be aware to avoid ambiguity. It is important that all column names in a query are unique. If this is not the case, these columns must be qualified by adding the table name in front of the column name. Additionally, an alias is chosen. It is also possible to join a table with itself or select data from a subquery. In this case, an alias is used for the data source.

SELECT id, name, genus, species FROM modorg, class WHERE modorg.class_id = class.id;

ERROR 1052 (23000): Column 'id' in field list is ambiguous

SELECT modorg.id as mo_id, name, genus, species FROM modorg, class WHERE modorg.class_id = class.id;

SELECT a.col, b.col FROM src1 [as] a, src2 [as] b WHERE …

There are 4 types of joins:

- INNER JOIN Show rows present in both tables
- LEFT JOIN Show all rows from the left table, even without linked data in the right table
- RIGHT JOIN Show all rows from the right table, even without linked data in the left table
- OUTER JOIN Show all rows from both table (does not exist in MySQL)

When using a JOIN, the asterisk (*) is used to select all columns from all tables. To specify the linked columns (FKs) ON is used. Keep in mind that joins are very inefficient because they use a lot of memory and time. The format in which the data is retrieved is also not necessarily the same as the format in which the data is stored.

The execution order of the SELECT statement is slightly changed by the addition of the JOIN:

1. FROM – input columns are determined
    a. JOIN clause
2. WHERE – input columns are filtered
3. GROUP BY – sorting and grouping of filtered input
4. Aggregation functions are calculated
5. HAVING – aggregation functions are filtered
6. ORDER BY – output is sorted
7. LIMIT/OFFSET – output is chopped

2.1.9 Efficiency and speed

Complex queries tend to lead to a large system load. As a result, the interface becomes slow and the user has to wait. To avoid this, it is important to choose the right column types when designing a database and to include only the necessary columns when querying one. Frequently used queries can be sped up using view and indices. Another way to increase speed is to allow a certain amount of redundancy.

Views

If a query is frequently re-used, it can be saved as a special table-like object. The speed gain depends on the use case. In MySQL a virtual table is created and used to serve up data in an orderly fashion.

CREATE VIEW *viewname* as SELECT …;

SELECT … FROM viewname WHERE … ORDER BY …;

Index

An index is a mechanism to retrieve specific rows very fast. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more resources this uses. If the table has an index for the column(s) in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially. An index is stored separately from the table. The golden rule is to use indices on columns used in the WHERE clause. Keep in mind that only one index per query can be used. On the upside, an index can contain multiple columns.

CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX *index_name* ON *tbl* (*col*);

MySQL uses indexes for these operations:

- To find the rows matching a WHERE clause quickly.
- To eliminate rows from consideration. If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows (the most selective index).
- To retrieve rows from other tables when performing JOINs. MySQL can use indexes on columns more efficiently if they are declared as the same type and size. In this context, VARCHAR and CHAR are considered the same if they are declared as the same size.
- To find the MIN() or MAX() value for a specific indexed column *key_col*. This is optimized by a pre-processor that checks whether you are using WHERE *key_part_N = constant* on all key parts that occur before *key_col* in the index. In this case, MySQL does a single key lookup for each MIN() or MAX() expression and replaces it with a constant. If all expressions are replaced with constants, the query returns at once.
- To sort or group a table if the sorting or grouping is done on a leftmost prefix of a usable index (for example, ORDER BY *key_part1*,*key_part2*). If all key parts are followed by DESC, the key is read in reverse order. (Or, if the index is a descending index, the key is read in forward order.)
- In some cases, a query can be optimized to retrieve values without consulting the data rows. (An index that provides all the necessary results for a query is called a covering index.) If a query uses from a table only columns that are included in some index, the selected values can be retrieved from the index tree for greater speed

Indexes are less important for queries on small tables, or big tables where report queries process most or all of the rows. When a query needs to access most of the rows, reading sequentially is faster than working through an index.

An FK is an index used to improve the speed of JOIN operations. Although it is not required for a JOIN, it is highly recommended to use them.

Redundancy

The schema with the least amount of redundancy is not always the fastest. By introducing some redundancy, the number of JOIN operations can be limited, thus speeding up the query. There are 2 widely used database schemas: snowflake and star. A snowflake schema does not allow redundancy and is easy to maintain and change. Complex queries with more JOINs are needed to retrieve data. It uses less space but is slower than other schemas. A star schema on the other hand, contains (some amount of) redundant data. Its less easy to maintain and change, but a lot faster than the snowflake schema. It also has a lower query complexity, but uses more space because data is stored twice or more. A star schema has a fact table that contains numerical data and has millions of data points and dimension tables which contain attributes of facts.

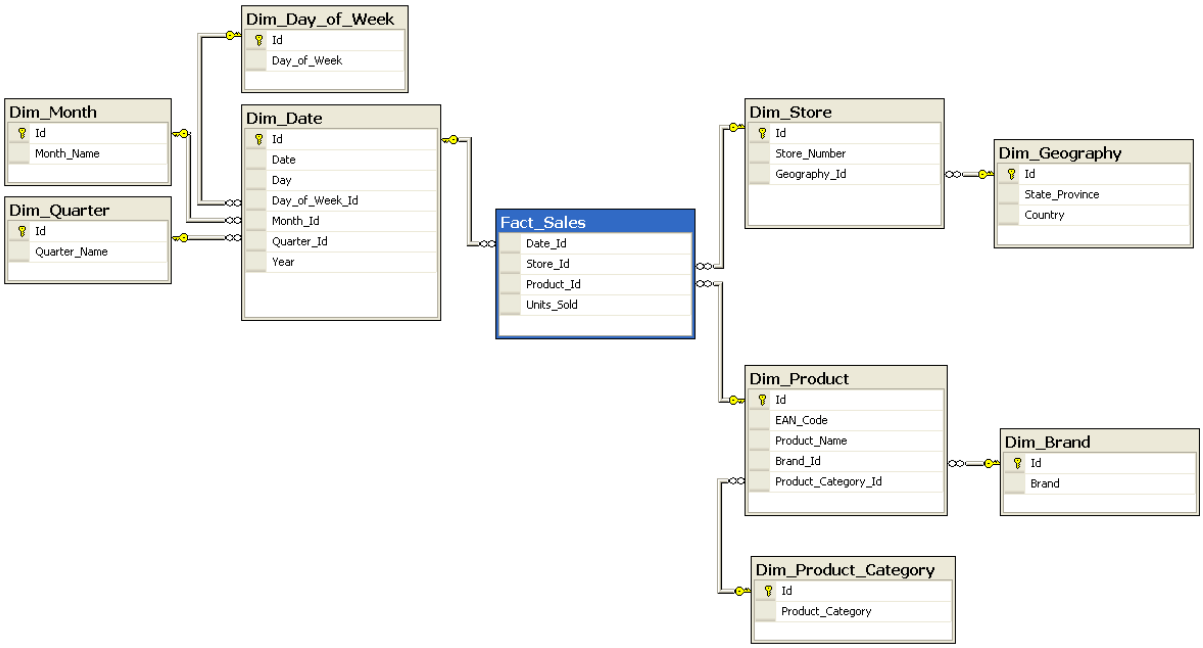

Figure 9: Snowflake schema

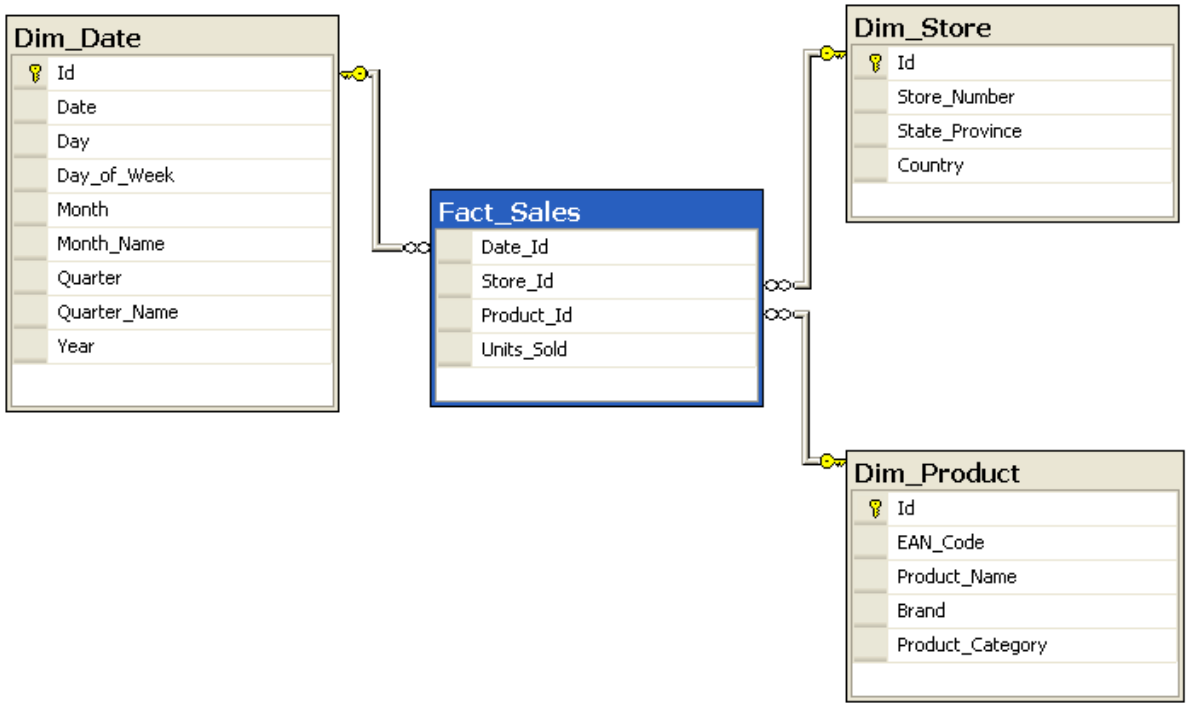

Figure 10: Star schema

2.1.10 Backup

MySQL provides the ability to dump a complete database into a text file. It includes statements for creating the database, tables, views, … and for inserting data. A dump file can be used to restore the database.

$ mysqldump [opt] *db* > *db.sql*

$ mysql *db* < *db.sql*

2.2 MySQL Workbench

More advanced users execute DDL commands through the MySQL monitor. On servers this is the primary method for interacting with a database as well. However, there are some readily available graphical user interfaces (GUIs) to do so, like HeidiSQL or MySQL Workbench.

sudo yum install mysql-workbench-community.x86_64

Instead of using the command line interface of the MySQL monitor, the GUI of MySQL Workbench makes it easier for users to interact with a database. On the welcome screen, users can add a number of connections to different database servers.

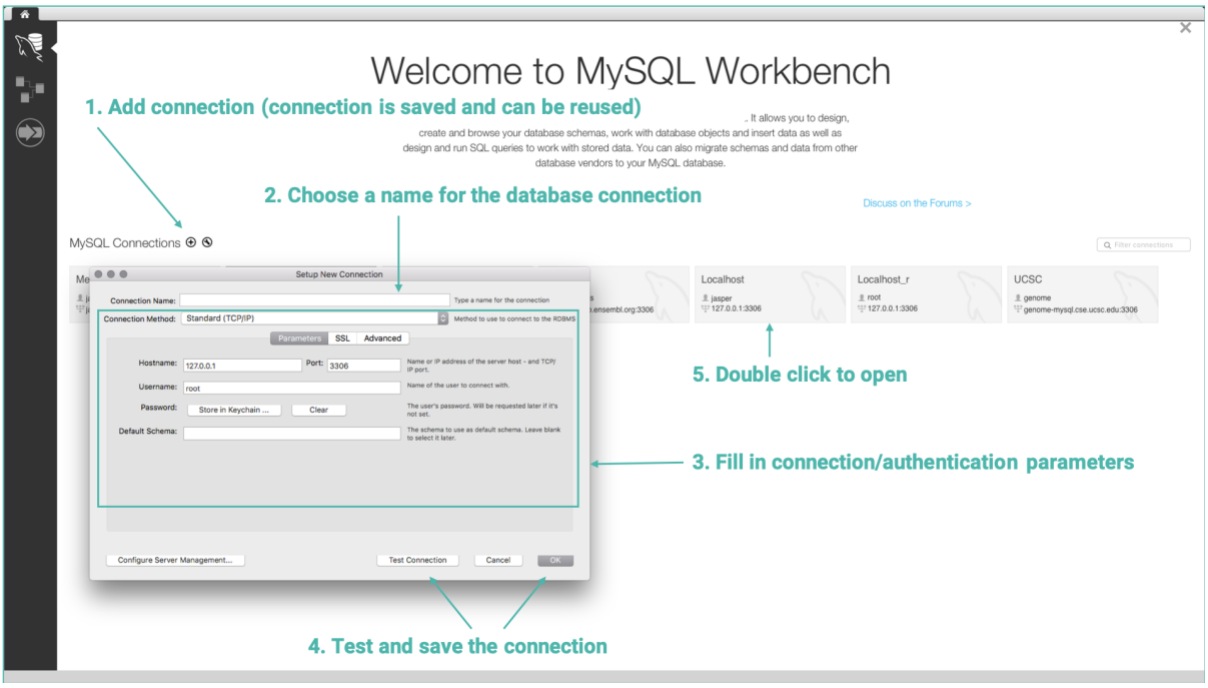

Figure 11: MySQL Workbench welcome screen

After adding a connection, all interaction with the database take places in the main window. This window consists out of (at least) 3 panes: a query pane, a results pane and a navigator pane. In the query pane, one can type and execute all possible SQL statements. The results of the statement are then displayed in the results pane. More information about the query and a history of executed queries can be found in the output pane. The navigator pane shows all information about the database server the user is connected to. The currently selected database is shown in bold. All queries will be executed in this database. A simple double click will change the database. The pane also allows the user to browse the tables in the database. More information on the currently selected item (table, column) is shown as well.

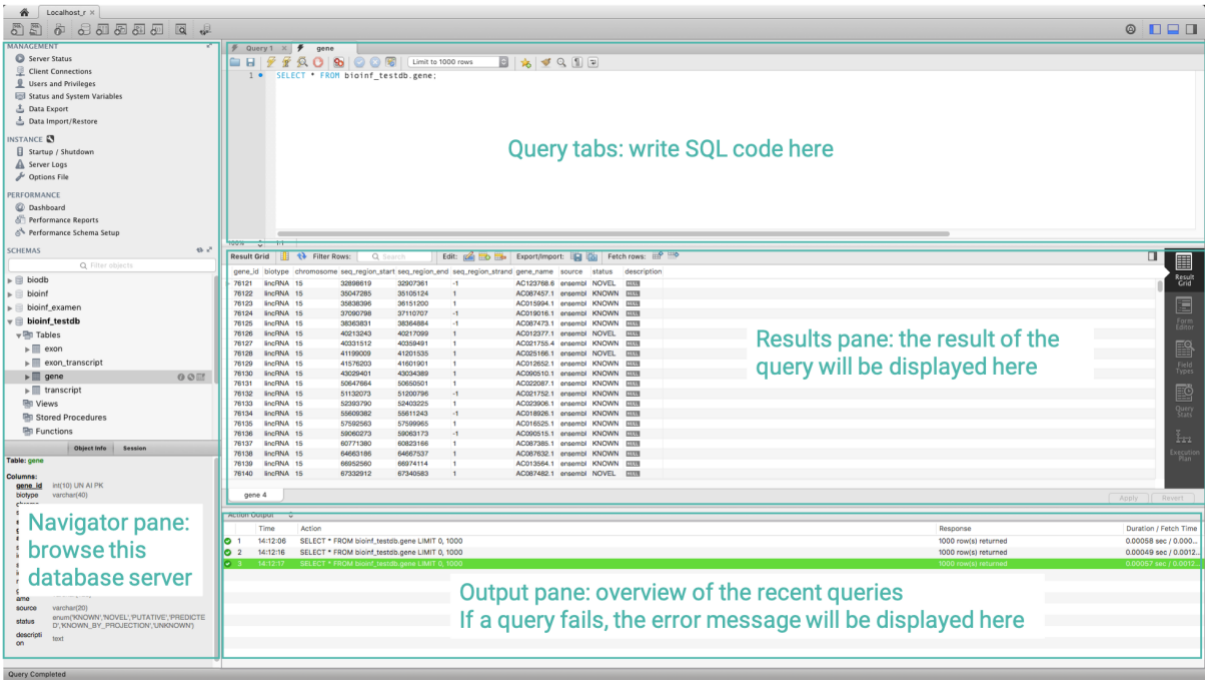

Figure 12: MySQL Workbench main window

2.2.1 Data model

A data model determines the structure of the data. It normally consists out of 3 parts:

- The conceptual data model identifies the highest-level relationships between different entities. No attributes or primary keys are specified (yet).
- The logical data model is more complex than the conceptual data model. It describes the data in as much detail as possible. All attributes as well as the primary key for each entity are specified. Foreign keys are specified too. Normalization occurs at this level.
- The physical data model represents how the model will be built in the database. All tables and columns are specified and foreign keys are used to identify relationships between tables.

All these models can be saved in an Entity Relationship Diagram (ERD). This is a graphical representation of the structure of a database.

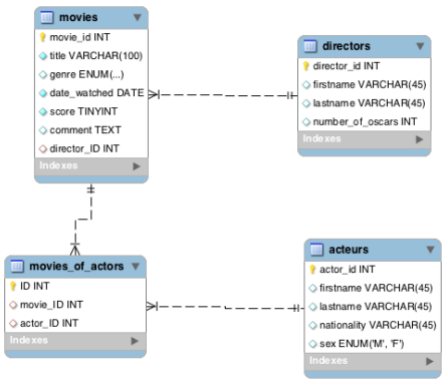

Figure 13: Entity Relationship Diagram

There are several types of database models that can be used to create a database.

- Flat model
  A single two-dimensional array of data elements, e.g. a spreadsheet.
- Hierarchical model
  The data is organised in a tree-like structure. Each child record has only one parent, whereas each parent record can have on or more child records. A record is a collection of fields, with each field containing only one value. They are connected through links. Fields are defined by the entity type.
- Network model
  Each record can have multiple parents and child records.
- Relational model
  Tables are seen as relations and the links between tables are not explicitly defined. Keys are used instead. This is the closest to what we've been using in this course unit (so far).
- Object-relational model
  A relational model with object-oriented features. Mostly used in PostgreSQL.
- Object oriented model
  All data is represented in the form of object. It uses the same model of representation as in most programming languages.

2.2.2 Normalisation

As mentioned before, when creating a database model, at a certain point in the process normalisation comes to place. Normalisation is nothing more than organizing columns and tables in order to reduce redundancy and improve integrity. The standard normalisation procedure follows the 12 rules of Edgar F. Codd and can be split into several Normal Forms (NFs).

- Unnormalized form
  Group all data in one entity
- First normal form (1NF)
  All repeating and calculated groups are eliminated in individual tables, as all attributes should be atomic. A separate table is created for each set of related data. Each of these sets is identified with a primary key.
- Second normal form (2NF)
  Every non-prime attribute of the table is dependent on the whole key of every candidate key. A non-prime attribute of a relation is an attribute that is not a part of any candidate key of the relation.
- Third normal form (3NF)
  Every non-prime attribute of a relation is non-transitively dependent on every key of that relation. Simply put: Every non-key attribute must provide a fact about the key, the whole key, and nothing but the key.
- Boyce-Codd normal form (BCNF)

No transitive dependencies. None of the key attributes provides facts about another key attribute within the same table, except about the entire primary key.

- 4NF– Essential tuple normal form (ETNF) – 5NF – 6NF – Domain/key normal form (DKNF)

2.2.3 Creating tables

Creating tables in MySQL Workbench can easily be done with the model interface. This is a kind of drawing board for your database. It automatically generates the DDL statements. Inside the model interface, tables can be placed and dragged around anywhere. The relations between different tables in the form of FKs will be displayed as lines and arrows, each with a different outlook depending on the type of relation (1:n, n:m).

Figure 14: MySQL Workbench Model Interface

MySQL Workbench makes it possible to create an actual database based on a model. To achieve this, the forward engineering function is used. It generates SQL code to create or modify a database based on a model.

Before starting with creating a database, keep the following questions in mind:

- Which data will be stored in the database,
- What are the constraints (column types, unique values, …)?
- For which application will the database be used?
- What are the relations between the data?

The answers to all these questions need to be considered when creating an ERD.

Another handy functionality of MySQL Workbench is the revere engineering. This function does the exact opposite of the forward engineering function and generates a (graphical) model of a database.

2.2.4 Workbench does it all

MySQL Workbench is capable of doing (almost) everything advanced users can complete with the monitor. It has the ability to dump your database or to export just a fraction of the results in a wide variety of file formats. Next to that, it is also able to import an entire dump file or a previously exported file (in the right file format).

2.3 MySQL and other languages

Off course, it is possible to include MySQL in other (programming) languages.

2.3.1 PHP

PHP can interact with MySQL in several ways. To this end, it uses extensions or adapters. PDO or PHP Data Objects is one of the currently supported adapters. It works with up to 12 different database systems. More specific to MySQL is the MySQLi extension. It can be used in a procedural or object-oriented manner.

```
<?php
// Procedural
$servername = "localhost";
$username = "username";
$password = "password";

// Create connection
$conn = mysqli_connect($servername, $username, $password);

// Check connection
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}
echo "Connected successfully";


// Create database
$sql = "CREATE DATABASE myDB";
if (mysqli_query($conn, $sql)) {
    echo "Database created successfully";
} else {
    echo "Error creating database: " . mysqli_error($conn);
}


// INSERT
$sql = "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('John', 'Doe', 'john@example.com')";
```

```
if (mysqli_query($conn, $sql)) {
    $last_id = mysqli_insert_id($conn);
    echo "New record created successfully. Last inserted ID is: " . $last_id;
} else {
    echo "Error: " . $sql . "<br>" . mysqli_error($conn);
}


// prepare and bind
$stmt = mysqli_stmt_init($conn);
mysqli_stmt_prepare($stmt, "INSERT INTO MyGuests (firstname, lastname, email) VALUES
(?, ?, ?)");
mysqli_stmt_bind_param ($stmt,"sss", $firstname, $lastname, $email);

// set parameters and execute
$firstname = "John";
$lastname = "Doe";
$email = "john@example.com";
mysqli_stmt_execute ($stmt);

echo "New record created successfully";
mysqli_stmt_close($stmt);


// SELECT
$sql = "SELECT id, firstname, lastname FROM MyGuests";
$result = mysqli_query($conn, $sql);

if (mysqli_num_rows($result) > 0) {
    // output data of each row
    while($row = mysqli_fetch_assoc($result)) {
        echo "id: " . $row["id"]. " - Name: " . $row["firstname"]. " " . $row["lastname"]. "<br>";
    }
} else {
    echo "0 results";
}


mysqli_close($conn);
?>

<?php

// Object-oriented
$conn = new mysqli($servername, $username, $password);


$sql = "CREATE DATABASE myDB";

$conn->query($sql);

// INSERT
$sql = "INSERT INTO MyGuests (firstname, lastname, email)
VALUES ('John', 'Doe', 'john@example.com')";

if ($conn->query($sql) === TRUE) {
    $last_id = $conn->insert_id;
    echo "New record created successfully. Last inserted ID is: " . $last_id;
}


// prepare and bind
$stmt = $conn->prepare("INSERT INTO MyGuests (firstname, lastname, email) VALUES (?,
?, ?)");
```

```
$stmt->bind_param("sss", $firstname, $lastname, $email);

// set parameters and execute
$firstname = "John";
$lastname = "Doe";
$email = "john@example.com";
$stmt->execute();

echo "New records created successfully";

$stmt->close();


// SELECT
$sql = "SELECT id, firstname, lastname FROM MyGuests";
$result = $conn->query($sql);

if ($result->num_rows > 0) {
    // output data of each row
    while($row = $result->fetch_assoc()) {
        echo "id: " . $row["id"]. " - Name: " . $row["firstname"]. " " . $row["lastname"]. "<br>";
    }
}


$conn->close();
?>
```

2.3.2 Python

Another common interaction is the one between MySQL and Python. Here there are several modules readily available as well. Both mysql.connector and MySQLdb are valid options. More info on this can be found in the Scripting module of this study program.

```
import mysql.connector
cnx = mysql.connector.connect(user='username', database='myDD')
cursor = cnx.cursor()
query = ("SELECT id, firstname, lastname FROM MyGuests")
cursor.execute(query)
cursor.close()
cnx.close()
```

```
import MySQLdb
db = MySQLdb.connect(host="localhost",user="username",passwd="password",db="myDB")
cur = db.cursor()
cur.execute("SELECT id, firstname, lastname FROM MyGuests")
db.close()
```

3 NoSQL

Next SQL systems, there are also NoSQL systems. These are another kind of database management system that store data in a non-relational manner. They are more flexible than an SQL system and are often used for big-data and real-time web applications. The downside is they lack the ACID characteristics.

- Atomic
  An indivisible and irreducible series of database operations either all occur, or nothing occurs
- Consistent
  A database transaction either creates a new valid state, or restores the previous valid state. This implicates that after the transaction all integrity rules of database are met.
- Isolated
  Transactions are executed isolated from other transactions, meaning that simultaneous transactions do not have insight in each other's intermediate results.
- Durable
  A completed transaction cannot be made invalid (even if the system crashes).

3.1 Categories

NoSQL databases can be divided in several categories, depending on the way the data is stored.

3.1.1 Column store

In the column store variant, tuples consist of 3 elements, a unique name, a value and a timestamp. The number of columns can change from row to row and a tuple does not have to be in every row. An example of this is Apache Cassandra.

3.1.2 Key-value store

Data is stored in a dictionary or hash. This is a collection of objects or records with different fields. There is a unique key per record. Within a single collection, different fields per record are possible. A major advantage of this method is that it is very memory friendly. An example of this is the Oracle NoSQL Database.

3.1.3 Graph store

Data is stored in nodes, edges and properties. Nodes represent entities that are roughly the equivalent of the relation or row in a relational database. Edges, also known as graphs or relationships are the lines that connect nodes to other nodes. They represent the relationship between them. Properties are pertinent information that relate to nodes.

Graph store methods are used for simple and rapid data retrieval based on shortest path queries using SPARQL. An example of this is OrientDB.

3.1.4 Multi model

The multi model method uses multiple data models against a single, integrated backend. Document, graph, relational, and key-value models are supported. An example of this is Couchbase, which uses both a relational and document store.

3.1.5 Document store

Another method is the document store. It contains semi-structured data and is a subclass of the key-value store. Every stored object can be different from every other and all information for a given object is stored in a single instance. Each document is identified by a unique key. The document store method supports CRUD (Create, Read, Update, Delete) and is often used for web applications. Some advantages of this method are the lack of predefined data formats and normalisation. A change in type and form has no effect on the database and existing stored documents.

3.2 MongoDB

One of the most known examples of a document store NoSQL system is MongoDB. It is free and open-source. Data is stored in JSON-like documents. Ad hoc queries, indexing, and real time aggregation (MapReduce) provide powerful ways to access and analyse the data. MongoDB provides the ability to store files in a Grid File System (GridFS). Moreover, it allows for replication and load balancing through sharding.

As mentioned before, MongoDB stored document in a JavaScript Object Notation format. This is a data-interchange format that is easy to parse and generate. It consists out of objects and arrays with attribute-value pairs. They are converted to BSON documents which are stored in collection that are in turn stored in databases.

Figure 15: JSON format document in MongoDB

3.2.1 Installing MongoDB

To install MongoDB on your system (Fedora 28, $releasever = 7), follow the instructions on https://docs.mongodb.com/manual/tutorial/install-mongodb-on-red-hat/

After installing MongoDB, make sure it is running correctly. If not, start in manually.

```
$ sudo service mongod start
$ sudo service mongod stop
$ sudo service mongod restart
$ mongod
```

3.2.2 The mongo shell

Once installed and started MongoDB, connect to the mongo shell.

```
$ mongo
```

The mongo shell is similar to the MySQL monitor. It is a command line interface to interact with a database. Similar to MySQL a database needs to be selected in order to query it. By entering the use myDB command, a database is selected. If the database does not exist, it get created. To add data to a collection the insert() function is used. The insert() operation creates the collection if it does not exist.

```
> use myNewDB
> db.myNewCollection1.insert({x:1})
```

3.2.3 MongoDB documents

In MongoDB data is stored in documents. Each record is a separate document and contains field and value pairs. The values of a field may include other documents, arrays, and arrays of document. Some of the main advantages of documents are the similarity to programming languages, the decreased need for joins and the polymorphism of the documents.

Figure 16: A MongoDB record

Similar to MySQL, a field can have a variety of data types. The ObjectId type is mostly used for unique identifiers. Other types are, amongst others, strings, timestamps, embedded documents, the Date type, arrays and the NumberLong type.

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

Figure 17: A MongoDB document with different data types

There are some restrictions to the use of field names in MongoDB. They cannot start with the dollar sign ($) character and cannot contain the dot (.) or the null character. The field name _id is reserved for use as a primary key. Its value must be unique in the collection, is immutable, and may be of any type other than an array. By default, MongoDB creates a unique index on he _id field during the creation of a collection. It is always the first field in a document.

MongoDB uses the dot notation to access the elements of an array (zero-based index) and to access the fields of an embedded document (field name).

3.2.4 The mongo shell – episode 2

A number of commands to interact with the mongo shell are similar to the ones used by MySQL.

- See all databases on the server
  > show dbs
- List all methods you can use on your db object
  > db.help()
- See all collections in a database
  > show collections
- List all methods you can use on your collection object
  > db.collection.help()
- List all cursor methods
  > db.collection.find().help()

As in MySQL, query optimization is very important. To gain speed indexes can be created and the number of query results can be limited with the limit() method. Query selectivity is another method to optimize the query. Less selective queries match a larger percentage of documents. Next to that, the use of a covered query, where all the fields in the query are part of an index and all the fields returned in the results are in the same index. A final way to reduce memory usage and increase speed is to return only the necessary data by using query projections.

3.2.5 Create operations

To add new documents to a collection there are a number of ways. The insert() method, as well as the insertOne() and insertMany() methods can be used. The first method is used to insert a single or multiple documents into a collection. To insert a single document into a collection insertOne() is used, to insert multiple insertMany() is used. With each of these methods, the collection is created if it does not exist.

> db.collection.insert()
> db.collection.insertOne()
> db.collection.insertMany()

3.2.6 Read operations

To retrieve documents from a collection, one can use either the find() or findOne() method. The first method returns a cursor to the matching documents. It is possible to add optional fields like a query filter or a query projection. A cursor modifier can be used if wanted.

> db.collection.find()
> db.collection.findOne()

A query filter is used to specify an equality condition. Optionally, query operators van be used. By default, AND is used when specifying multiple conditions. If OR is needed, specify it explicitly.

Query operators

- Comparison

{ <field>: { <operator>: <value> } }

$eq	Values that are equal to a specified value
$gt	Values that are greater than a specified value
$gte	Values that are greater than or equal to a specific value
$lt	Values that are less than a specified value
$lte	Values that are less than or equal to a specific value
$ne	Values that are not equal to a specified value

{ field: { <operator>: [<value1>, <value2>, ... <valueN>] } }

$in	Any of the values specified in an array
$nin	None of the values specified in an array

- Logical

{ <operator>: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

$or	All documents that match the conditions of either clause
$and	All documents that match the conditions of both clauses
$nor	All documents that fail to match both clauses

{ field: { $not: { <operator-expression> } } }

$not	All documents that do not match the query expression

- Element

{ field: { $exists: <boolean> } }

$exists	All documents that have the specified field

- Array

{ <field>: { $all: [<value1> , <value2> ...] } }

$all	Arrays that contain all elements specified in the query

{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }

$elemMatch	Documents where the element in the array field matches all the specified $elemMatch conditions

{ field: { $size: 2 } }

$size	Documents if the array field is a specified size

Query projection

A query projection is used to specify which field are shown in the results. Inside the projection a field gets a value, which can be 0 (false) or 1 (true) or an expression using a projection operator.

- Projection operators

db.collection.find({ <array>: <value> … }, { "<array>.$": 1 })

$	First element in an array that matches the query condition

{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }

$elemMatch	First element in an array that matches all the specified $elemMatch conditions

Next to query and projection operators, there are also cursor modifiers. These are methods that modify the way the underlying query is executed.

- sort() Returns results ordered according to a sort specification
- count() Returns the number of documents in the result set
- hasNext() Returns true if the cursor has documents and can be iterated
- next() Returns the next document in a cursor
- limit() Constraints the size of a cursor's result set
- skip() Returns a cursor that begins returning results only after passing or skipping a number of documents
- size() Returns a count of the documents in the cursor after applying skip() and limit() methods
- distinct() Returns only unique values

3.2.7 Update operations

To modify one or more existing documents in a collection, the update or replace commands are used. Similar as with insert operations, there are several possibilities. The update() method updates on or multiple documents at once. The updateOne() method is used to update a single document, whereas updateMany() is used to update multiple documents. Another option is using the replaceOne() method.

> db.collection.update()
> db.collection.updateOne()
> db.collection.updateMany()
> db.collection.replaceOne()

In order to update documents, sometimes update operators are used. These make it easier to update a document following certain rules.

- Update operators

{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }	
$inc	Increments the value of the field by the specified amount
{ <operator>: { <field1>: <value1>, ... } }	
$set	Sets the value of a field in a document
$addToSet	Adds elements to an array only if they do not already exist in the set
$pop	Removes the first or last item of an array

$pull	Removes all array elements that match a specified query
$push	Adds an item to an array

3.2.8 Delete operations

To delete documents from a collection the remove() method is used to delete one or multiple documents at the same time. The deleteOne() method removes a single document and the deleteMany() method removes multiple documents.

```
> db.collection.remove()
> db.collection.deleteOne()
> db.collection.deleteMany()
```

3.2.9 MongoDB vs. SQL

When comparing MongoDB and SQL a lot of similarities can be found. Off course there are some differences as well. A database is called a database in both systems. An SQL table is a collection in MongoDB, a row is a document and a column is a field. Both systems make use of indices and a primary key remains a primary key. Instead of using joins, MongoDB support embedded documents.

3.2.10 Aggregation

To calculate a single result across multiple documents, the aggregate() method can be used. To simplify the process of aggregation, there are many aggregation pipeline operators available, such as $sum, $avg, $group, $match, $limit, $skip, $sort, $concat, $first, $last and many others.

```
> db.collection.aggregate()
```

3.2.11 Robo 3T

Comparable to MySQL Workbench for MySQL, MongoDB has several GUI options as well. One being MongoDB Compass, a software package to visually explore and interact with your data with full CRUD functionality.

Another option is Robo3T, which will be used in this course unit. It has the full power of a MongoDB JavaScript environment as it embeds a complete JavaScript engine. It supports the usage of multiple (isolated) shells at the same time. Autocompletion for all objects and functions make it easy to work with. As a plus, it is cross-platform and open source.

4 Databases in bioinformatics

4.1 Online databases

Because of the large amount of available data, a lot of (online) databases are used in bioinformatics. They make biological data available to a multitude of scientists. Because published data may be difficult to find, it is collected in a single (centralized) place. It also makes the biological data available in a computer-readable form which is needed for analysis.

There are many types of databases, divided based on several properties.

- Type of data
  - Nucleotide sequences
  - Protein sequences
  - Gene expression data
  - Metabolic pathways
  - 3D structure
- Data entry and quality control
  - Directly deposited data
  - Data added and updated by appointed curators
  - Marking or removal of erroneous data
  - Type and degree of error checking
- Primary or derived data
  - Primary: experimental results
  - Secondary: results of analysis on primary databases
  - Aggregate of many databases: collision or combination of data
- Technical design
  - Flat files
  - Relational database
  - Object oriented database
- Maintainer status
  - Large, public institution (EMBL, NCBI)
  - Quasi-academic institute (Swiss Institute of Bioinformatics, TIGR)
  - Academic group or scientist
  - Commercial company
- Availability
  - Publicly available, no restrictions
  - Available, but with copyright
  - Accessible, but not downloadable
  - Academic, but not freely available
  - Commercial

4.1.1 Identifiers and accession codes

In these databases, an entry is identified in two different ways. The first is by using an identifier. This is a string of letters and digits and can usually change in time. The other way is by using an accession code or number. This is a stable number that uniquely identifies an entry in its database.

4.1.2 Primary nucleotide sequence databases

When looking at primary nucleotide sequence databases, one of the most used types of online databases in bioinformatics, three main players come to mind. All of these do little error checking and contain redundant data. They are synchronised on a daily basis and have no legal restrictions.

The European Nucleotide Archive (ENA, http://www.ebi.ac.uk/ena) stores both DNA and RNA sequences in three different databases: the Sequence Read Archive, the Trace Archive and the EMBL Nucleotide Sequence Database. All of these are maintained by the European Bioinformatics Institute (EBI). The data can be accessed in XML, HTML, FASTA and FASTQ format.

GenBank is another open access database with publicly available nucleotide sequences and their protein translations. It contains data about more than 100.000 distinct organisms. The National Center for Biotechnology Information (NCBI) maintains this database. Its entries are retrievable through the NCBI GenBank webpage or via FTP. GenBank uses its own file format which is basically a flat file with three main sections: a header, a features part and a sequence part.

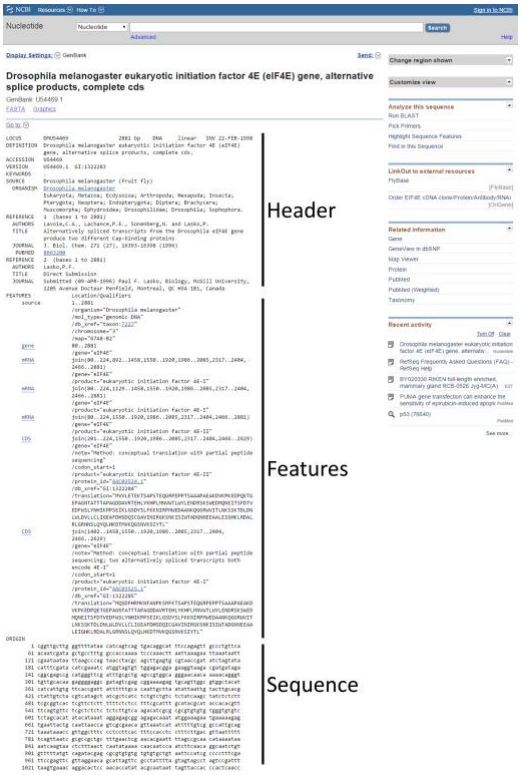

Figure 18: The GenBank file format

The third main primary nucleotide sequence database is the DNA Data Bank of Japan (DDBJ, http://www.ddbj.nig.ac.jp). As stated in its name, it contains only DNA sequences. It is the only nucleotide sequence database in Asia.

4.1.3 Secondary nucleotide sequence databases

These databases contain information from analyses performed on data from primary nucleotide sequence databases.

RefSeq contains sequences of DNA, RNA and their protein products. It is annotated and curated. There is a single record in this database for each natural biological molecule.

OMIM is a catalogue of human genes and genetic disorders and traits. It is based on the selection and review of published peer-reviewed literature.

HapMap is a haplotype map of the human genome. It contains genetic variants affecting health, disease and responses to drugs and environmental factors.

4.1.4 Other nucleic acid databases

Next to primary and secondary nucleotide sequence databases, there is a plethora of other nucleic acid databases.

- Gene expression databases
  Mostly microarray data (Gene Expression Omnibus, Expression Atlas, …)
- Gene ontology
  Relationships between concepts within a domain
- Genome databases
  Annotated and analysed genome sequences (Ensembl Genomes, Flybase, Wormbase, …)
- Phenotype databases
  Phencode
- RNA databases
  miRbase, LNCipedia, …

4.1.5 Sequencing databases

Datasets from sequencing experiments are stored in sequencing database. The Sequence Read Archive (SRA), hosted by NCBI, contains raw data in BAM-format. Experimental metadata is also available. Another one is the European Genome-phenome Archive, hosted by EMBL-EBI. This data is not publicly available.

4.1.6 Protein databases

Protein sequences and protein structures are stored in protein databases. Data derived from translation of nucleotide sequences is collected in secondary databases such as NCBI protein or trEMBL (part of Uniprot). SwissProt contains data from computational analyses which is manually reviewed and annotated. Protein structures are stored in, amongst others, Protein Data Bank or NCBI Structure

4.1.7 The General Feature Format

The features of a particular gene, DNA and protein sequence are kept in a tab-delimited file. Per line, one feature is described. All but the final field in each feature line must contain a value and empty columns should be denoted with a dot (.).

Apart from some general information, preceded by ##, a gff-file has the following columns:

- seqname
  Name of the chromosome or scaffold; chromosome names can be given with or without the 'chr' prefix.
- source
  Name of the program that generated this feature, or the data source (database or project name)
- feature
  Feature type name, e.g. Gene, Variation, Similarity
- start
  Start position of the feature, with sequence numbering starting at 1.
- End
  End position of the feature, with sequence numbering starting at 1.
- score
  A floating point value.
- strand
  Defined as + (forward) or - (reverse).
- frame
  One of '0', '1' or '2'. '0' indicates that the first base of the feature is the first base of a codon, '1' that the second base is the first base of a codon, and so on.
- attribute
  A semicolon-separated list of tag-value pairs, providing additional information about each feature.

```
##gff-version 3
##sequence-region P69905 1 142
P69905  UniProtKB       Initiator methionine    1       1       .       .       .
Note=Removed;Ontology_term=ECO:0000269,ECO:0000269,ECO:0000269,ECO:0000269;evidence=ECO:0000269|PubMed:12665801,ECO:0000269|PubMed:13872627,ECO:0000269|PubMed:13954546,ECO:0000269|PubMed:14093912;Db
xref=PMID:12665801,PMID:13872627,PMID:13954546,PMID:14093912
P69905  UniProtKB       Chain   2       142     .       .       .       ID=PRO_0000052653;Note=Hemoglobin subunit alpha
P69905  UniProtKB       Metal binding   59      59      .       .       .       Note=Iron (heme distal ligand)
P69905  UniProtKB       Metal binding   88      88      .       .       .       Note=Iron (heme proximal ligand)
P69905  UniProtKB       Site    12      12      .       .       .       Note=Not glycated
P69905  UniProtKB       Site    57      57      .       .       .       Note=Not glycated
P69905  UniProtKB       Site    61      61      .       .       .       Note=Not glycated
P69905  UniProtKB       Site    91      91      .       .       .       Note=Not glycated
P69905  UniProtKB       Site    100     100     .       .       .       Note=Not glycated
P69905  UniProtKB       Modified residue        4       4       .       .       .       Note=Phosphoserine;Ontology_term=ECO:0000244;evidence=ECO:0000244|PubMed:24275569;Dbxref=PMID:24275569
P69905  UniProtKB       Modified residue        8       8       .       .       .       Note=N6-succinyllysine%3B alternate;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        9       9       .       .       .       Note=Phosphothreonine;Ontology_term=ECO:0000244;evidence=ECO:0000244|PubMed:24275569;Dbxref=PMID:24275569
P69905  UniProtKB       Modified residue        12      12      .       .       .       Note=N6-succinyllysine;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        17      17      .       .       .       Note=N6-acetyllysine%3B
alternate;Ontology_term=ECO:0000244;evidence=ECO:0000244|PubMed:19608861;Dbxref=PMID:19608861
P69905  UniProtKB       Modified residue        17      17      .       .       .       Note=N6-succinyllysine%3B alternate;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        25      25      .       .       .       Note=Phosphotyrosine;Ontology_term=ECO:0000244;evidence=ECO:0000244|PubMed:24275569;Dbxref=PMID:24275569
P69905  UniProtKB       Modified residue        36      36      .       .       .       Note=Phosphoserine;Ontology_term=ECO:0000244;evidence=ECO:0000244|PubMed:24275569;Dbxref=PMID:24275569
P69905  UniProtKB       Modified residue        41      41      .       .       .       Note=N6-succinyllysine%3B alternate;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        50      50      .       .       .       Note=Phosphoserine;Ontology_term=ECO:0000244;evidence=ECO:0000244|PubMed:24275569;Dbxref=PMID:24275569
P69905  UniProtKB       Modified residue        103     103     .       .       .       Note=Phosphoserine;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        109     109     .       .       .       Note=Phosphothreonine;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        125     125     .       .       .       Note=Phosphoserine;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        132     132     .       .       .       Note=Phosphoserine;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        135     135     .       .       .       Note=Phosphothreonine;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        138     138     .       .       .       Note=Phosphothreonine;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
P69905  UniProtKB       Modified residue        139     139     .       .       .       Note=Phosphoserine;Ontology_term=ECO:0000250;evidence=ECO:0000250|UniProtKB:P01942
```

Figure 19: The General Feature Format (GFF)

4.1.8 Genome browsers

Genome browsers provide a graphical interface for genomic data. The most known are the UCSC genome browser and the Ensembl genome browser. The first one has the ability to search by gene name or location. The second one contains annotated genes aligned to a reference genome. Data can be exported in multiple formats, such as FASTA, GFF, EMBL, ….

4.2 Direct queries, API, REST API

For some of the online available databases, it is possible to query them directly from MySQL. UCSC, Gene Ontology and Ensembl all provide this option.

When looking to the Ensembl database more closely, it is clear that it has quite complex database schemas. On top of this, it is not suited to retrieve sequences.

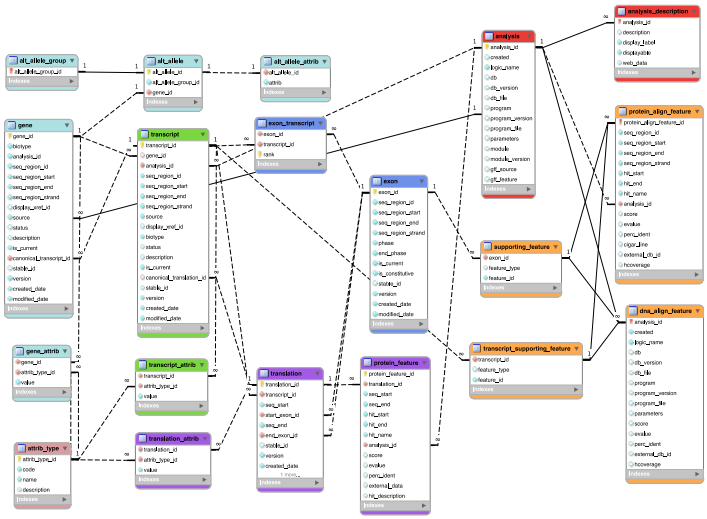

Figure 20: An Ensembl database schema

4.2.1 Ensembl API

To solve this problem and simplify user access to their databases, Ensembl developed an API (Application Programming Interface). This provides a uniform method of access to the data. It is reusable in different systems and insulates developers from underlying database changes. This make an API a very reliable system. The Ensembl API is a Perl API. Installation instructions and different versions, based on Ensembl releases, can be found on the Ensembl website. The basic idea is to use a Registry to find database and connect to them.

```
Bio::EnsEMBL::Registry->load_registry_from_db(
        -host => 'ensembldb.ensembl.org',
        -user => 'anonymous',
        -verbose => '1'
);
```

There are several Ensembl databases that can be accessed with the API. The core database contains annotation information for each organism in Ensembl. The core databases are species specific databases. The compara database is a cross-species database with genome-wide species comparisons on both DNA-sequence (whole genome alignments, synteny regions, conservation scores) and gene level (phylogenetic trees, homology predictions). The variation database contains areas of the genome that differ between individual genomes. It provides associated disease and phenotype information. Both sequence variants (SNPs, insertions, deletions, indels, substitutions) and structural variants (CNV, inversions, translocations) are kept. The last Ensembl database is the regulation database. It holds gene expression data and its regulation in human and mouse. It focusses on transcriptional and post-transcriptional mechanisms.

4.2.2 Ensembl REST API

Because the Ensembl API is a Perl API it is not suited for every scientist. To that account, Ensembl developed a REST API (Representational state transfer/RESTful). This is basically the Ensembl API made available using a base URL and standard HTTP methods, such as options, GET, PUT, POST and DELETE. The main advantage is that the REST API provides language agnostic bindings to Ensembl data. A user is able to create a REST client in JAVA, Perl, Python and Ruby.

```python
#!/usr/bin/env python

import sys
import urllib
import urllib2
import json
import time

class EnsemblRestClient(object):
    def __init__(self, server='http://rest.ensembl.org', reqs_per_sec=15):
        self.server = server
        self.reqs_per_sec = reqs_per_sec
        self.req_count = 0
        self.last_req = 0

    def perform_rest_action(self, endpoint, hdrs=None, params=None):
        if hdrs is None:
            hdrs = {}

        if 'Content-Type' not in hdrs:
            hdrs['Content-Type'] = 'application/json'

        if params:
            endpoint += '?' + urllib.urlencode(params)

        data = None

        # check if we need to rate limit ourselves
        if self.req_count >= self.reqs_per_sec:
            delta = time.time() - self.last_req
            if delta < 1:
                time.sleep(1 - delta)
            self.last_req = time.time()
            self.req_count = 0

        try:
            request = urllib2.Request(self.server + endpoint, headers=hdrs)
            response = urllib2.urlopen(request)
            content = response.read()
            if content:
                data = json.loads(content)
            self.req_count += 1
```

```python
        except urllib2.HTTPError, e:
            # check if we are being rate limited by the server
            if e.code == 429:
                if 'Retry-After' in e.headers:
                    retry = e.headers['Retry-After']
                    time.sleep(float(retry))
                    self.perform_rest_action(endpoint, hdrs, params)
            else:
                sys.stderr.write('Request failed for {0}: Status code: {1.code} F

        return data

    def get_variants(self, species, symbol):
        genes = self.perform_rest_action(
            '/xrefs/symbol/{0}/{1}'.format(species, symbol),
            params={'object_type': 'gene'}
        )
        if genes:
            stable_id = genes[0]['id']
            variants = self.perform_rest_action(
                '/overlap/id/{0}'.format(stable_id),
                params={'feature': 'variation'}
            )
            return variants
        return None


def run(species, symbol):
    client = EnsemblRestClient()
    variants = client.get_variants(species, symbol)
    if variants:
        for v in variants:
            print '{seq_region_name}:{start}-{end}:{strand} ==> {id} ({consequen

if __name__ == '__main__':
    if len(sys.argv) == 3:
        species, symbol = sys.argv[1:]
    else:
        species, symbol = 'human', 'BRAF'

    run(species, symbol)
```

Figure 21: Ensembl REST API in Python

The basic structure of the REST API contains zero or more required parameters, zero or more optional parameters. In a standard URL, required parameters are flagged with a double-point (:). Optional parameters should go into the request body if performing a POST or as key-value pairs after the question mark (?) if performing a GET. Parameters specify what is required and indicate the type of returned data from the REST API. Some common parameters are: id, region, species, symbol, external_db, object_type and callback.

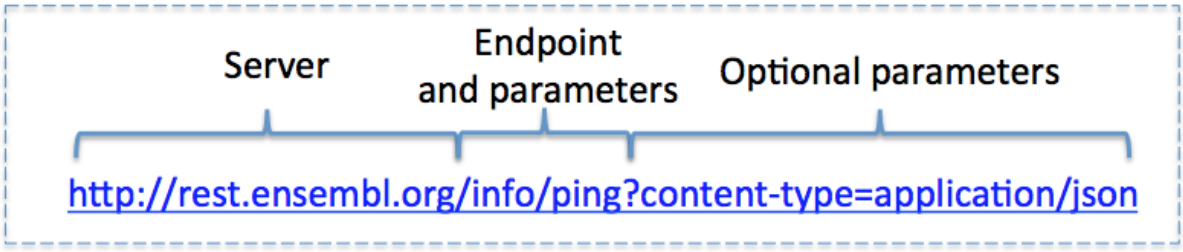

Figure 22: Ensembl REST API URL structure

The REST API can be accessed through a web browser, but often command line is used as well. To do so, the curl command is needed. This is a command line tool and library for transferring data with URLs.

curl '*<url>*'
[-H '*<header>*']
[-X *<request_method>*]
[-d '*<data>*']

It is possible to output results from the REST API in a number of formats (JSON, FASTA, BED, XML, …). Specifying the value depends heavily on your client and the type of operation you are executing. GET requests support 4 ways of specifying the format. POST only supports one.
The Ensembl REST API has several endpoints. Depending on the type of data searched for, a different endpoint should be used.

- Archive

GET archive/id/:id	Uses the given identifier to return the archived sequence
POST archive/id	Retrieve the archived sequence for a set of identifiers

- Comparative genomics

GET genetree/id/:id	Retrieves a gene tree for a gene tree stable identifier
GET genetree/member/id/:id	Retrieves the gene tree that contains the gene / transcript / translation stable identifier

GET genetree/member/symbol/:species/:symbol	Retrieves the gene tree that contains the gene identified by a symbol
GET alignment/region/:species/:region	Retrieves genomic alignments as separate blocks based on a region and species
GET homology/id/:id	Retrieves homology information (orthologs) by Ensembl gene id
GET homology/symbol/:species/:symbol	Retrieves homology information (orthologs) by symbol

- Variation

GET variation/:species/:id	Uses a variant identifier (e.g. rsID) to return the variation features including optional genotype, phenotype and population data
POST variation/:species	Uses a list of variant identifiers (e.g. rsID) to return the variation features including optional genotype, phenotype and population data

- Sequence

GET sequence/id/:id	Request multiple types of sequence by stable identifier. Supports feature masking and expand options.
POST sequence/id	Request multiple types of sequence by a stable identifier list.
GET sequence/region/:species/:region	Returns the genomic sequence of the specified region of the given species. Supports feature masking and expand options.
POST sequence/region/:species	Request multiple types of sequence by a list of regions.

5 Version control

In (bio)informatics a special kind of database with history tracking uses GIT. It is a software package that is used to track and store revisions or versions of files. An online platform to use this tool is GitHub. A great tutorial on how to use this, can be found on https://try.github.io.

The most used commands are

- Initialize

  $ git init

- Show status

  $ git status

- Track files

  $ git add <*filename*>

- Commit changes

  $ git commit [-m "<*commit_message*>"]
  $ git reset

- Show logs

  $ git log

- Checkout a commit

  $ git checkout <*checksum*>

- Show differences between revisions

  $ git diff [<*checksum1*> [<*checksum2*>]]

- Branching

  $ git branch
  $ git branch <'*new_branch*'>

- Merging

  $ git merge <*new_branch*>

  - Merge conflicts: same file modified on 2 seperate branches

- Delete branch

  $ git branch -d <*new_branch*>

- Remotes

  - GitHub (public repositories)

- Clone repository

  $ git clone <*repository_name*> <*local_dir*>

- Update repository

  $ git pull <*remote_name*> <*branch_name*>

- Submit changes

  $ git push <*remote_name*> <*branch_name*>

6 Exam insights

Note: These questions are intentionally kept vague. For the theoretical part, the slides about this course unit are used. The exercises part is similar to the exercises done in class and the rehearsal exercises.

6.1 Theoretical part (closed book)

1. A question about database management systems (what, characteristics, examples)
2. A question about SQL (components, actions)
3. A general question about a relational database system (structure, relationships, …)
4. A question about column types
5. A question about relational database schemas
6. A question about MongoDB (usage, (dis)advantages, …)
7. A question about the APIs

6.2 Practical part (open book)

Every solution is added to a git repository.

8. Normalisation: 1 exercise
9. Creation of relational database model based on given data
10. MySQL queries: from normal to a bit harder
11. NoSQL queries: from normal to a bit harder
12. Ensembl, Ensembl API, Ensembl REST API

7 References list

This course material was based on the following references:

- Specialised bioinformatics course by Pieter-Jan Volders (Advanced bioinformatics for masters in Biomedical Sciences)
- Basics of databases and MySQL by Luc Ducazu and Gerben Menschaert (Training at VIB Bioinformatics Core)
- Sequence databases by Janick Mathys (Training at VIB Bioinformatics Core)