Introduction

Aim of the course

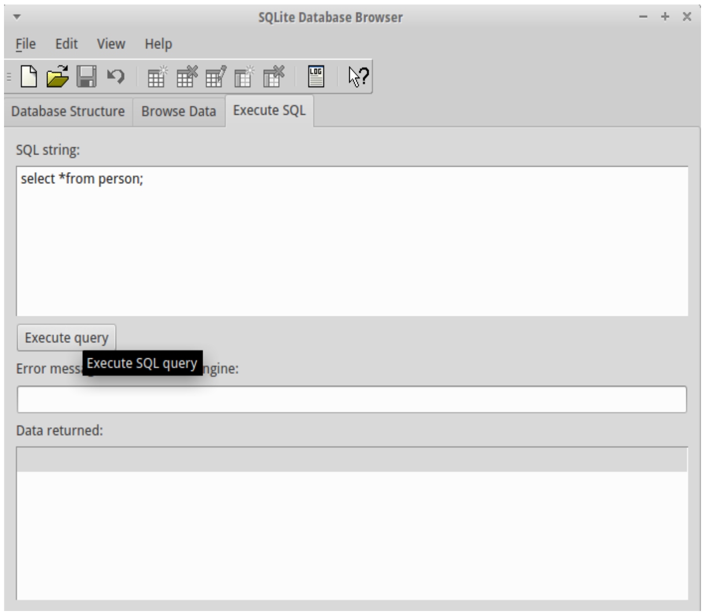
After the completion of this course, you will be able to use an SQL client to obtain data stored in an SQL database such as MySQL, MS-SQL, or Oracle DB. You will learn how to write queries which give you exactly the information you require and to bring them into the form most suitable to answer the given questions.

SQL databases

SQL, being an acronym for ***S**tructured* ***Q**uery* ***L**anguage,* does not denote a data base system but merely a *database language* used to access databases. However, we speak of *SQL databases* when we mean a database system which provides an SQL interface. Such databases are called *relational databases* as the data is stored in a set of table between which relationships exist (the primary-key / foreign-key relation described below).

Software used

In this tutorial, we work with the file-based database system *sqlite* and the corresponding client *sqlitebrowser*. S*qlite* (http://www.sqlite.org/) is simple and quite powerful and allows to store a database in an single file. Besides many other applications it is also used in the android operating system to store application data on phones and tablets. How to install the require software on Linux, MacOS, and Windows is detailed in the provided installation guide.



```
select *from person;
```

To perform a SQL query to a database, enter the command in the text field „SQL string“ and click on „Execute Command“. As an alternative to *Sqlitebrowser* you can use *Sqliteman* which is a good alternative especially for Debian and Ubuntu Linux users. Please refer to the installation guide for details.

Sample database

As the focus of this tutorial lies on the retrieval of data from an existing database rather than creating new databases, a file *databaseOTS.sqlite* is provided which contains all the data required later on in the course.

General hints

Write all sql-Statements in an text editor first. This way it is much easier to reuse statements formulated before. All text in `fixed font` are SQL statements. Note that SQL key words such as `select` or `from` are *not* case sensitive. All <u>underlined</u> text denotes questions. The answer to the questions (or a SQL statement which yields the answer) is found at the very end of this document. You are strongly encouraged not only to answer the questions but also to try out all presented commands with some examples of you own.

SQL Basics

Simple queries

In a SQL database, data is stored in a set of tables. A table contains a collection of entries (="lines"), information of the same type.

We want to see what is in the table person:

```
select * from person;
```

This is too much information as we are only interested in the name:

```
select name from person;
```

In general:

```
select col1, col2, ... from tablename;
```

We can get the number of lines of results by using the `count` command. To get the number of entries in the person table we use

```
select count(*) from person;
```

Results might contain duplicates (they form a multi-set). We can get only the distinct results by

```
select distinct col1,col2, ... from tablename;
```

1.) How many different cities of residence do the people in the persons table have?

Types

Columns in databases can have different types. The most important ones are `varchar(n)` for text data with up to `n` characters, `integer` for integer numbers (e.g. 12 or -4), `real` for floating point numbers (e.g. 23.3434), `date` for dates (e.g. "12/12/2014"), `time` for value denoting a time of the day, and `boolean` for entries which can be either true or false (e.g. does the phone number belong to an active contract?).

Constraints

Quite often we are not interested in all of the entries of the database, especially because it might contain a huge number of lines. To get only the lines fulfilling some constraints, the `where` clause is used, e.g.

```
select name, firstname from person where age > 30;
```

For the different SQL data types, different constrains make sense. For all types, we can check for *equal* and *not equal*.

```
=,!=
```

For numbers, one can also compare for *larger, larger or equal, smaller, smaller or equal*

```
>,>=,<,<=
```

E.g.

```
age > 30;
```

For text, we use `like`, to find entries where the text value in a column contains a string, where you can use the % as a wild-card for unknown parts of a word. E.g. if you search for `'%an%'`, the following words would fit: B**an**ana, f**an**tasy, but not Anna, since this is an capital A.

In *sqlite* the `>,<,=` operators for dates, do an *lexicographical* comparison, not a *numeric* one. This means that if there is an inconsistent representation of dates, the comparison might fail even though the two values represent the same dates. E.g. if once the format "2014-10-23" and once the format "2014/10/16" is used. Note that here double quotes " are used instead of single quotes '.

Furthermore, the operator `in` checks if the left-hand-side is a member of the set on the right-hand-side. The two following statements are equivalent (try yourselves).

```
name in ('Philipp', 'Carlos')

name = 'Philipp' or name = 'Carlos'
```

2.) Get all people whose names contains "oh".

Constraints can be combined by the logical operator `and` and the operator `or`.

3.) Find all people whose first name is Carlos and who are older than 20 years.

There is another table which is called *phone_contract*. The column containing the status (active, non-active) is called *status*.

4.) Find out, how many contracts there are and how many accounts are active.

Results from more than one table

Very often, it is necessary to get information which is spread over more than one table. For example, you may want to know the names of the people who have an active phone contract.

When you know what you are looking for you are already able to formulate an concrete condition in the `where` clause. With SQL you can combine all information from various tables, e.g.

```
select person.name, phone_contract.phone_number from person,
     phone_contract;
```

Note that we could also write `phone_contract` instead of `phone_contract.phone_number` as the table person does not contain a column `called phone_number` and, therefore, no ambiguity could arise.

5.) What does this query return?

To output the matching names and phone numbers, we can use a where clause again

```
select p.name, phone_number from person p, phone_contract c
     where c.name = p.name;
```

As we did not want to write the complete table name, we have defined and abbreviation in the from-clause (person p).

6.) Find all names who have an active phone_contract.

7.) Find all names and phone numbers of people who are older than 30.

Sub-queries
The methods described above allow to get information from an arbitrary number of tables. But if you want to formulate an condition which depends on values of certain data entries, you need sub-queries. Assume, we want to get all names and phone numbers of all people who are older than Carlos:

```
select person.name, phone_number from person, phone_contract
     where person.name = phone_contract.name and age >
     (select age from person where name = 'Carlos');
```

The part in the last line in parentheses is called a *sub-query. S*uch sub-queries can be a little bit confusing at first sight so you might want to pause for a moment. An operator which is very useful when working with sub-queries is `in`. Keep this in mind when solving the following criminal case.

The theft of the Mona Lisa

Now we can start to use our new skills to solve a criminal case: one of the most startling art thefts in history: on October the 23rd, 2014 the Mona Lisa was stolen from the Louvre in Paris. There is no trace of the thieves. You are working for Interpol and will find out who is responsible for this crime. Due to new regulations you have access to any data of any international companies and states, like register of residents, mobile phone text messages or bank data. So, using these resources, find the thieves!

Which tables are there? - System tables

So, first you need to get an overview of what kind of data is actually available. i.e. you want to know what tables are in the database. The sqlite database contains are table named `sqlite_master` which holds meta information. We can get the names of the tables by an ordinary SQL query:

```
select name from sqlite_master where type = 'table';
```

To get information about a specific table use the pragma command:

```
pragma table_info (tablename);
```

8.) Use the two commands above to get information about the columns in the various tables.

Remarks: In other databases (e.g. Oracle, DB2 etc.) this meta information is stored differently, i.e. the system tables are always called differently. Sometimes there is more than one system table. When working with another database than sqlite you need to google for the names of the system tables of that database.

How to find the thief

For finding the thief we have the following plan:

- Find out, who was in Paris at that time.
- The thief was probably not working alone. Is there any suspicious communication during the time in question?
- If you find the people responsible, who was the actual wire-puller and who was "only" executing henchman?

Who was in Paris?

When you try to transform the following questions into sql queries, you might want to check for the actual column names again using the pragma command!

9.) Look at the system table. What table could contain information regarding traveling?

10.) Get an idea what kind of information is stored in the travel-table? (Hint: have a look at the actual data, and a second look on the table information)

11.) Get the name of all persons who live in Paris.

12.) Get all names that did a journey to Paris before 23.10.2014.

13.) Get all names that did a journey from Paris after 23.10.2014.

14.) Get all names, that did a journey to Paris before 23.10.2014, where this name is also in an entry for a journey from Paris after the 23.10.2014.

15.) Get all names of persons who live in Paris or spent their time in Paris on 23.10.2014 (according to the travel data).

Congratulations! You have reduced the number reduced the number of suspicious tremendously!

How information of different tables are cleanly connected: Key constraints

The local police will pay these people a visit to ask for an alibi for the time in question. So you need a list of names and their residence. Repeat the select from before, but this time query for name as well as for residence.

Now, check the result. There is something wrong! In the list of conspicuous persons, there is one, who actually never was in Paris, i.e. who lives neither in Paris nor was traveling there.

16.) Who is it?

Hint: Perform a select on the flight table where the name is one of those who do not live in Paris, and check, and then check for every person-residence pair if there is a corresponding flight. For one is not. Who? Why is this person turning up in that list? Look at your where statement.

To prevent mistakes like this, data is usually more structured: People who create a table, usually define a column (or the combination of several columns) which is unique for every entry. This column is called the *primary key*. Other tables which are referencing entries of this table have a corresponding column, containing the same value as the primary key column of the referenced table. This corresponding column is called *foreign key*. For example, in our person table the field `ID` is the primary key. To find out about the primary keys of a table we can use the pragma command as

```
pragma table_info (tablename)
```

This returns a list of columns where the last number is non-zero exactly if the column is a primary key. However, to also see foreign keys you have to use the column `sql` in the table `sqlite_master`.

17.) What foreign keys are used in the table containing the messages and which is the referenced table?

Since the primary key is unique, it is also unique which entry is referenced. Often, primary key columns contain the suffix "id" (=identifier) and foreign key columns contain the name of the referenced table, like names_id. However, this is only a *convention* which might not be fulfilled for all databases you ever encounter.

18.) Try the query from the last section again, but this time use as the connection column not the name, but the foreign key-column.

For this, but also for other reasons not discussed here: If you query data spreading over more than one table, and if there exists a connection between a *primary key* and a *foreign key, always use this connection! Do not use other columns, which seem to serve the same purpose!*

Who is the thief? - Order by and group by

To find out, who is the thief check the text messages stored by the mobile phone providers.

19.) How is the name of the table containing the text messages and the one containing phone contracts?

20.) Get all text messages which where sent between 2010-10-20 and 2010-10-25.

21.) Get all contract ids where the contract.name_id is equal to one of the persons you get with the result set of the last section.

22.) Get all text messages where the sent date is the 23.10.2014 and the `contract_sender_id` is equal to the contract ids where the `contract.name_id` is equal to one of the persons you get with the result set of the last section.

You see that you got all the required information but the output looks kind of chaotic. You can order an result set according to a column with an order-by-phrase. The query to get all text messages from 21.10.2014 ordered by time reads

```
select message from messages where sent like '2014-10-21%'
      order by sent;
```

Remember date comparison is in sqlite a lexicographical comparison. So, if we would use sent = '2014-10-21' we got an empty result set. Instead we must use the % wild-card for the time information as shown above.

23.) Get a list of messages from our suspects from the given time period and sort the messages.

24.) Who are the thieves?

Hint: Read the conversations as they a give a clear trace. Check the ids of sender and receiver of the message and look it up in the person tables!

Who was the string puller? - Data quality and aggregation functions

A week later the local police found the thieves and had them arrested. The picture was already sold on the black market two days after the coup, but for a reasonable chance for mitigation of punishment they disclosed the buyers and the picture could be brought back to the museum where it belongs.

Data quality

But for the court trial, further evidence needs to be collected, and the question who actually initiated the fraud is open. For this, one may conclude, that the original initiator kept more money than the henchmen. So, have a look at the bank data of our thieves.

25.) Which tables are containing bank data?

26.) How are these table connected with each other, and with the names table? Which primary/foreign keys are there? Hint: Often it is a good idea to use pen and paper to draw a diagram denoting the connections. One possible way to to this is to represent to represent tables by a list of their columns and to draw a line for every primary key-foreign key relationship.

27.) Which columns can you use to get the connection between names and the account table? Hint: Again, using pen and paper might be very helpful.

28.) Get the account numbers of our thieves.

Depending on the actually query, you might be surprised: Philipp does not have a bank account? But you are almost sure he has! This is a very common situation, that you do not get the results you expect. There are some possibilities:

- The SQL-query was not formulated in a suitable way
- The data does not exists
- The data is corrupt

Our experience with large databases is that most common reason is the first one, followed by the third reason. So unclean data is a common issue.

What can in this situation? Try not to compare the exact names with =, but try the like-comparator.

29.) Get a list of account details taking into account the different ways to spell Sarah or Philipp.

To deal with unclean data one sometimes has to be creative and to try different things to find the data one is looking for. Once again: If there is a primary key/ foreign key connection, use it! These columns have extra checks by the database, so the probability that these are corrupt is much, much lower than for other columns. If there are no such primary-key columns, it is usually a bad database design which should be a quite rare exception.

Who got the money? - Aggregate Functions

If you want to calculate a quantity from several values from a column you use aggregate functions, e.g. average values or sums. One aggregate function we have already learned about: The count function. Important other functions are:

• `AVG()`	Returns the average value
• `COUNT()`	Returns the number of rows
• `FIRST()`	Returns the first value

- `LAST()` Returns the last value
- `MAX()` Returns the largest value
- `MIN()` Returns the smallest value
- `SUM()` Returns the sum

Even if it does not make much sense, we compute the average of all phone numbers:

```
select avg(phone_number) from phone_contract;
```

30.) What is the sum of all transactions of Sarah?

31.) What is the sum of all transactions of Sarah between 23.10. and 25.10.?

We could do now the same for Philipp. But we want to learn a few more useful commands first. SQL has a nice command for cases like the one, where you want to use an aggregate function for subsets according to values in another column:

```
select column_name, aggregate_function(column_name)
     from table_name
     where column_name operator value
     group by column_name;
```

For instance, if you want to know the sum of all transaction per account number, you can use

```
select account_number_fk, sum(amount)
     from bank_transaction
     where date > "2014-10-22" and date < "2014-10-26"
     group by account_number_fk;
```

Note that `group by` can also be used for more than one column.

These information would have provided an alternative way to identify suspected persons: Anyone who has gotten more than, for instance 20.000€, during the time period in question on their bank account would have been a suspect. In this case, we want to formulate a condition on the result of an aggregated function. Here, we cannot use a `where` clause, instead we use the `having`-clause:

```
select column_name, aggregate_function(column_name)
     from table_name
     where column_name operator value
     group by column_name
     having aggregate_function(column_name) operator value;
```

32.) Get a list of account numbers and sums of transactions for all accounts where the amount is greater than 10.000 € in the time in question. Who is probably the initiator of the crime?

Advanced Topics

Nullable columns

There are situations where not all fields of a row have a value. For example, in the bank_transaction table, the date might be missing because the intelligence service did not obtain this information from the bank. To find the flights without a date we can use the command

```
select * from bank_transaction where date is null;
```

33.) What is the percentage of flights where the date information is missing?

In practice, there are usually columns which are created to be *not nullable.* In this case, the database refuses to add rows which contain null values in the respective fields. This is always true for primary keys.

Joins

Assume we are looking all information on passengers who have send or received a bank transfer on the date of their flights.

34.) What is the SQL-query?

This kind of query is called an *inner join* on the two tables. An alternative syntax making this a little bit more explicit is

```
select * from flight inner join bank_transaction
     on flight.date = bank_transaction.date;
```

Now, we might want to see all flight data even if no bank transaction is performed on the same date while still showing the corresponding transaction data if there is one. This is called an *left outer join* or simply *left join*:

```
select * from flight left join bank_transaction
     on flight.date = bank_transaction.date;
```

35.) What happens when no bank transactions occurs at the flight date?

In a *right join* the roles of the two tables are just interchanged while in a *full outer join* all rows of both tables are shown. However, they are both not supported by sqlite.

Answers

1. `select count(distinct residence) from person;`
2. `select * from person where name like '%oh%';`
3. `select * from person where name = 'Carlos' and age > 20;`
4. a) `select count(*) from phone_contract;`
   b) `select count(*) from phone_contract where status = 'active';`
5. We all combinations of names and phone numbers.

6.
   ```
   select person.name from phone_contract where status = 'active'
         and person.name = phone_contract.name;
   ```

7.
   ```
   select p.name, phone_number from person p, phone_contract where
         p.name = c.name and age > 30;
   ```
8. ...
9. flight

10.
    ```
    select * from flight;
          pragma table_info(flight);
    ```

11.
    ```
    pragma table_info (person);
          select * from person where residence = 'Paris';
    ```

12.
    ```
    select name from flight where dest_city = 'Paris'
          and date < "2014-10-23";
    ```

13.
    ```
    select distinct name from flight where start_city = 'Paris'
          and date > "2014-10-23";
    ```

14.
    ```
    select distinct name from flight where dest_city = 'Paris'
          and date < "2014-10-23"
          and name in (select name from flight where dest_city
          = 'Paris' and date < "2014-10-23");
    ```

15.
    ```
    select distinct person.name from person, flight
          where residence = 'Paris'
          or (flight.name = person.name and dest_city = 'Paris'
    ```

```
      and date < "2014-10-23" and flight.name in
      (select flight.name from flight
      where dest_city = 'Paris' and date < "2014-10-23"));
```

16. `select * from flight where name in ('Philipp', 'Kesia', 'Sarah');`

```
select distinct person.name, residence from person, flight
      where residence = 'Paris' or
      (flight.name = person.name and dest_city = 'Paris' and
      date < "2014-10-23" and flight.name in
      (select flight.name from flight
      where dest_city = 'Paris' and date < "2014-10-23"));
```

17. Foreign keys: contract_sender_id, contract_receiver_id
    Table: phone_contract

```
      select sql from sqlite_master
            where type = ‘table’ and name = ‘messages’;
```

18.
```
select distinct person.name, residence from person, flight
      where residence = 'Paris' or
      (flight.person_id = person.id and dest_city = 'Paris'
      and date < "2014-10-23" and flight.name in
      (select flight.name from flight where dest_city = 'Paris'
      and date < "2014-10-23"));
```

19. `select name from sqlite_master where type = 'table';`

20.
```
select * from messages where sent > "2014-10-20"
      or sent < "2014-10-25";
```

21.
```
select id from phone_contract where phone_contract.person_id
      in (select distinct person.id from person, flight
      where residence = 'Paris' or
      (flight.person_id = person.id
      and dest_city = 'Paris' and date < "2014-10-23"
      and flight.name in (select flight.name from flight
      where dest_city = 'Paris' and date < "2014-10-23")));
```

22.
```
select * from messages where sent > "2014-10-20"
      or sent < "2014-10-25" and contract_sender_id in
      (select id from phone_contract where
      phone_contract.person_id in
      (select distinct person.id from person, flight
      where residence = 'Paris' or (flight.person_id = person.id
      and dest_city = 'Paris' and date < "2014-10-23"
```

```
     and flight.name in (select flight.name from flight
     where dest_city = 'Paris' and date < "2014-10-23"))));
```

23.
```
select * from messages
     where sent > "2014-10-20"  or sent < "2014-10-25"
     and contract_sender_id in (select id from phone_contract
     where phone_contract.person_id in
     (select distinct person.id from person, flight
     where residence = 'Paris'
     or (flight.person_id = person.id
     and dest_city = 'Paris' and date < "2014-10-23"
     and flight.name in (select flight.name from flight
     where dest_city = 'Paris'
     and date < "2014-10-23")))) order by sent;
```

24. Philip and Sarah

25. account and bank_transaction

26. Via foreign key on bank_transaction referencing the table `account`, both have a primary key and connection to table name via the column `name` *without* foreign constraint.

27. Via the column `name` in the tables `person` and `account`.

28. 203993 and 203987

29.
```
select * from account where name
     like 'Sara%' or name like 'Philip%';
```

30.
```
select sum(amount) from bank_transaction
     where account_number_fk = 203987;
select  sum(amount) from bank_transaction
     where account_number_fk = (select account_number_pk
     from account where name like 'Sara%');
```

31.
```
select sum(amount) from bank_transaction
     where account_number_fk = 203987
     and  date > "2014-10-22" and date < "2014-10-26";
```

32.
```
select account_number_fk, sum(amount)
     from bank_transaction where date > "2014-10-22";

select account_number_fk, sum(amount)
     from bank_transaction where date > "2014-10-22"
     and date < "2014-10-26"
     group by account_number_fk having sum(amount) > 10000;
```

33. `select count(*) from flight where date is null;`

34.
```
select flight.*, bank_transaction.*
        from flight, bank_transaction
        where flight.date = bank_transaction.date;
```

35. The corresponding banking details are filled with null values.