Databases

TDA357/DIT622– LP3 2024

Lecture 2

Ana Bove

(much of the material is based on material from
both Thomas Hallgren and Jonas Duregård)

January 17th 2024

Overview of Today's Lecture

- Basic **SQL** and PostgreSQL :
  - Working with PostgreSQL;
  - Create, delete and alter database tables;
  - Types and constraints;
  - Relational schemas;
  - Insert, update and delete data in tables;
  - Database queries involving one table;
  - Group-by and aggregations.
- We leave queries involving several tables for next lecture.

Relational Databases

Tables are a basic data structure in databases.

They can be seen as a collection of data.

A database consists of one or more tables.

Tables are also known as *relations*.

Example: A database with a single table and information about countries.
In bold face the name of the *attributes*:

Name	**Abbr**	**Capital**	**Area**	**Population**	**Continent**	**Currency**
Denmark	DK	Copenhagen	43094	5484000	EU	DKK
Estonia	EE	Tallinn	45226	1291170	EU	EUR
Finland	FI	Helsinki	337030	5244000	EU	EUR
Norway	NO	Oslo	324220	5009150	EU	NOK
Sweden	SE	Stockholm	449964	9555893	EU	SEK
...	...	...	...	...	...	...

SQL: Structured Query Language

A standardised language for relational databases.
Easy to read by non-experts.

There many implementations (DBMS), each of them following the standard to different degrees.

We will use PostgreSQL which is fairly good at following the standard.

SQL is case insensitive and we will use the following conventions:

- UPPERCASE for **SQL** keywords;
- Capitalised for the name of the tables;
- lowercase for the name of the attributes.

Spaces and line breaks can be inserted freely which allows to format the code nicely.

Working with **SQL** in the Course

- *Changes are persistant!*
  When you run an **SQL** statement that modifies the database, the changes remain until they are altered.
- When running your **SQL** statements, *the state of the database plays a role*:
  - Old stuff may cause problems;
  - Running the statements in another database might not give the results you expect since they might depend on previous things you have done in your database.
- While working on your tasks, *you might want to start on a clean database* until you are satisfied with the results.
- *Work incrementally*:
  - Write a (long) query bit by bit and test each step;
  - Write one query and (re)run it until it works without errors, only then start writing on the next one.
- There might be *several ways to do the same*. Check the manual/test!

Working with PostgreSQL

- Creates a new database called MyDataBase:

```
> createdb MyDataBase
```

- Connects to the database MyDataBase and starts the **SQL** interpeter:

```
> psql MyDataBase
psql (16.0)
Type "help" for help.

MyDataBase=#
```

- Disconnects from the database, all data will persist!

```
MyDataBase=# \q
```

- Deletes the database called MyDataBase and all its data:

```
> dropdb MyDataBase
```

- More generally: accesses your local PostgreSQL installation:

  (get help on Friday's lab session)

```
psql -U <username> <dbname>
```

Working PostgreSQL (Cont.)

- Runs a file in psql and executes its content in the database "portal":

```
portal=# \i <file-name>.sql
```

- Shows the description of a table:

```
portal=# \d <table-name>
```

- Copies data from tab-separated file into the table Countries, which should exist in the database:

```
portal=# \copy Countries FROM 'countries.tsv'
```

- Runs a command in a database (more on commands will come). One can use many lines, tabs and space, and needs to finish with ";":

```
portal=# INSERT INTO ...
portal=#    VALUES (...) ;
```

Comments in **SQL**

- Short comments:

```
-- All this line is a comment.
-- We need to start the line again with a double dash.
```

- Longer comments:

```
/* Here starts the comment
and it continues here.
Even this is part of the comment
but we will end it in the next line.
*/
```

Creating Tables

```
CREATE TABLE Countries (
    name TEXT,
    abbr CHAR(2),
    capital TEXT,
    area FLOAT,
    population INT,
    continent VARCHAR(10),
    currency CHAR(3) );
```

Adds a new (empty) table to the database.

We give the name of the table, its attributes, and their types.

Most Common Types

- INT/INTEGER - for 32 bit signed integers;
- REAL/FLOAT - for 32 bit floating point values;
- NUMERIC(p,s) - numbers with p digits before and s digits after '.';
- SERIAL - the value increases by one on each insert;
- BOOLEAN - for boolean constants TRUE and FALSE;
- TEXT - for variable sized strings;
- VARCHAR(n) - for variable sized strings with max size n;
- CHAR(n) - for fixed size strings of size n (like character arrays);
- TIMESTAMP - for date+time (microsecond resolution);
- DATE, TIME - for dates and times of days independently.

Constraining the Data

We can add *constraints* to disallow certain values and/or duplicates.

```
CREATE TABLE Countries (
    name TEXT PRIMARY KEY,
    abbr CHAR(2) NOT NULL UNIQUE,
    capital TEXT NOT NULL,
    area FLOAT NOT NULL,
    population INT NOT NULL CHECK (population >= 0),
    continent VARCHAR(10) NOT NULL,
    currency CHAR(3),
    CONSTRAINT none_sense CHECK (area > population * 10) );
```

- PRIMARY KEY - uniquely identifies each row, cannot be empty/null;
- NOT NULL - disallows empty/null values;
- UNIQUE - secondary key, disallows repetition;
- CHECK - sets a constraint in/among the values of a row;
- CONSTRAINT - sets a constraint and gives it a name.

Note: How much should we constraint the data?

Primary Keys (and a bite more on Contraints)

Each table should have a *single* primary key.

The primary key can however be *compound*, that is, consists of several attributes.

- Introduces a single but compound PRIMARY KEY

```
CREATE TABLE CourseGrades (
  student TEXT,
  course CHAR(6),
  grade INT DEFAULT 0,
  CONSTRAINT okgrade CHECK (grade IN (0,3,4,5)),
  PRIMARY KEY (student, course) );
```

- *Error:* multiple PRIMARY KEYs!

```
CREATE TABLE CourseGrades (
  student TEXT PRIMARY KEY,
  course CHAR(6) PRIMARY KEY,
  grade INT DEFAULT 0 CHECK (grade IN (0,3,4,5)) );
```

Database/Relational Schema

A *database/relational schema* is a compact way to describe a database.

It consists of a *relation schema* for each of the tables/relations in the database.

In each relation schema:

- name of the table and of its attributes are stated;
- the primary key is underlined (can consist of several attributes!);
- types are missing but ...
- ... constraints are stated.

Example of Relation Schemas

A country has the following *attributes*:

name, abbr, capital, area, population, continent, currency

A course grade has the following *attributes*: student, course, grade

The corresponding *relation schemas* are:

Countries (<u>name</u>, abbr, capital, area, population, continent, currency)
- Unique abbr
- population $\geqslant 0$
- area $>$ population * 10

CourseGrades (<u>student</u>, <u>course</u>, grade)
- grade $\in \{0, 3, 4, 5\}$
- Default grade is 0

Selecting Primary Keys

What is the right level of *identification*?

Too many/few attributes are problematic when choosing primary keys.

Quiz: What are the problems/advantages of these primary keys?

CourseGrades (student, course, grade)	Only one course/grade per student
CourseGrades (student, course, grade)	Only one student/grade per course
CourseGrades (student, course, grade)	Only one student/course per grade
CourseGrades (student, course, grade)	Only one student per course and grade
CourseGrades (student, course, grade)	A student can have several grade in a course
CourseGrades (student, course, grade)	Correct: Only one grade per student and course

Altering Tables

Many ways to alter a table; see documentation for more possibilities.

- Changes the type of a column: (what will happen with the data? test it!)

```
ALTER TABLE Countries ALTER COLUMN continent TYPE CHAR(2);
```

- Adds a new column:

```
ALTER TABLE Countries ADD language TEXT;
```

- Disallows empty values in a cloumn:

```
ALTER TABLE Countries ALTER COLUMN language SET NOT NULL;
```

- Deletes a column:

```
ALTER TABLE Countries DROP COLUMN language;
```

- Removes a constraint by its name:

```
ALTER TABLE Countries DROP CONSTRAINT none_sense;
```

Deleting Tables

- Deletes a table with all its data!

```
DROP TABLE CourseGrades;
```

Note: Gives an error if the table doesn't exist.

- Doesn't give an error:

```
DROP TABLE IF EXISTS Contacts;
```

Note: Deleting a table will fail if other tables have references to it.

(More on references to other tables next lecture!)

Inserting Data in Tables

- These insert will work: (observe the difference empty string vs. null value!)

```
INSERT INTO Countries
VALUES ('Denmark', 'DK', 'Copenhagen', 43094, 5484000, 'EU', 'DKK');
INSERT INTO Countries
VALUES ('Sweden', 'SE', '', -449964, 9555893, 'EU', NULL);
```

- These will not work after the inserts above: (can you see why?)

```
INSERT INTO Countries
VALUES ('Sweden', 'SE', 'Stockholm' , 449964, 9555893, 'EU', 'SEK');
```

```
INSERT INTO Countries
VALUES (NULL, 'SA', BsAs , 1780400, 44938712, 'AM', 'SAR');
```

```
INSERT INTO Countries
VALUES ('SmallArgentina', 'SA', 'BsAs' , 1780400, 44938712, NULL, 'SAR');
```

```
INSERT INTO Countries
VALUES ('BigDen', 'DK', 'BigCop', 43094000, 5484000, 'EU', 'DKK');
```

```
INSERT INTO Countries
VALUES ('Perú', 'PE', 'Lima' , 1285216, -32824358, 'AM', 'SOL');
```

Pattern Matching

Useful to check that strings follow a given format (see documentation):

```
CREATE TABLE Teachers (
   idnr TEXT PRIMARY KEY
   CHECK (idnr LIKE '______-____'),
   name TEXT
   CHECK (name LIKE '% %'),
   phone TEXT NOT NULL
   CHECK (phone SIMILAR TO '[0-9]{10}') );
```

LIKE:
- '_' means any character;
- '%' means any sequence of characters;

SIMILAR TO: uses regular expressions.

```
INSERT INTO Teachers
VALUES ('123456-7890', 'Ana Bove', '0123456789');
```

Queries

To retrieve information from the tables in a database.

- Selects everything:

```
SELECT * FROM Countries;
```

- Selects only some columns:

```
SELECT name, capital FROM Countries;
```

- Selects only some rows:

```
SELECT * FROM Countries
WHERE name='Sweden' OR name='Uruguay';
```

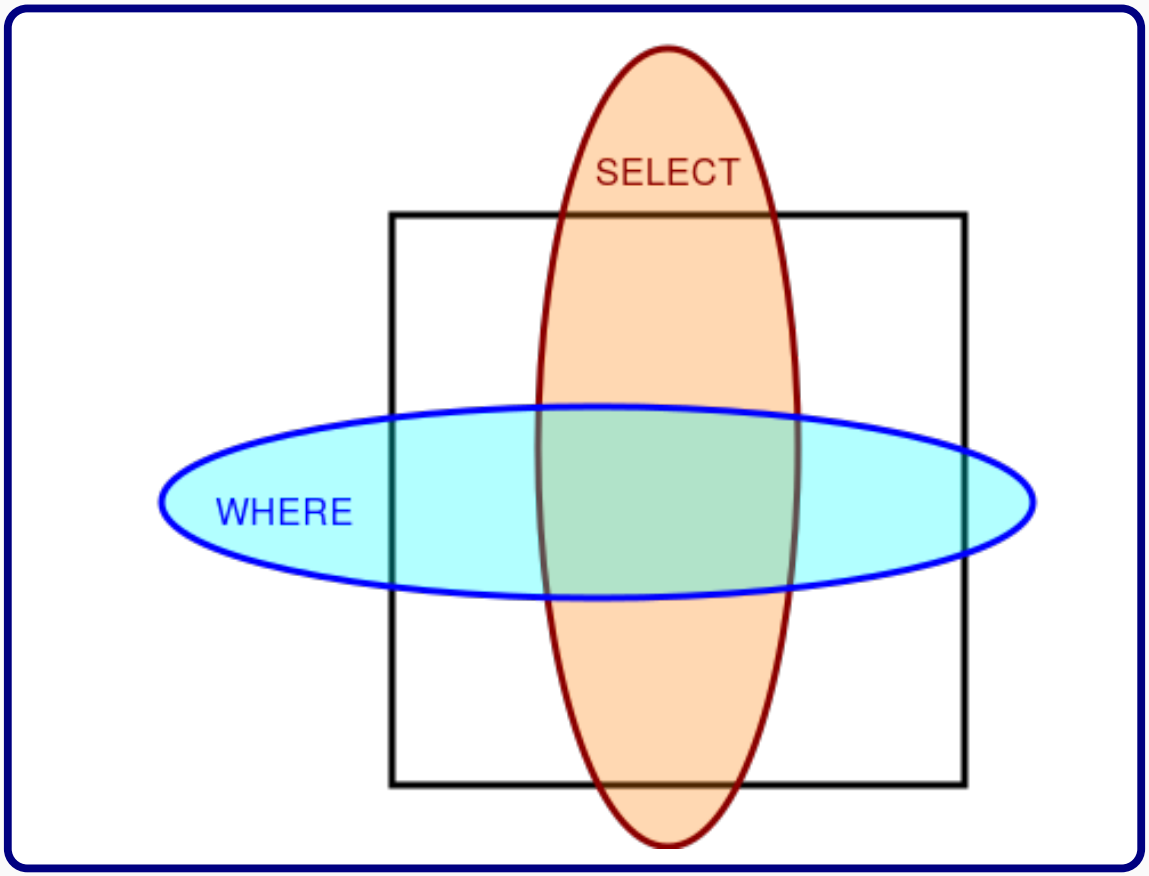
- Selects some columns in some rows:

```
SELECT name, capital
FROM Countries
WHERE area > 0 AND name IN ('Sweden', 'Uruguay');
```

Queries

The result of a query is another table, containing:

- a subset of the columns (as specified after SELECT);
- a subset of the rows (determined by the condition after WHERE).



Playing with the Output

- Orders the output (ascending is the default):

```
SELECT name, population FROM Countries
ORDER BY population;
```

- Indicates the order:

```
SELECT name, population FROM Countries
ORDER BY population ASC;
```

```
SELECT name, population FROM Countries
ORDER BY population DESC;
```

- Limits the number of rows in the outcome:

```
SELECT name, population FROM Countries
ORDER BY population DESC
LIMIT 5;
```

- Selects all outputs vs. only distinct outputs:

```
SELECT continent FROM Countries;
```

```
SELECT DISTINCT continent FROM Countries;
```

Quiz

- Do these two queries give the same result?

```
SELECT name, area, population FROM Countries
WHERE continent != 'EU'
ORDER BY area, population;
```

```
SELECT name, area, population FROM Countries
WHERE continent != 'EU'
ORDER BY population, area;
```

- Do these two queries give the same result?

```
SELECT name, abbr FROM Countries
WHERE continent != 'EU';
```

```
SELECT name, abbr FROM Countries
WHERE continent != 'EU'
ORDER BY population, area;
```

Modifying Data in Tables

We can set a new value for some columns in particular rows:

```
UPDATE Countries
SET continent = 'AM'
WHERE name = 'Uruguay';
```

```
UPDATE Countries
SET area = -area, capital = 'Stockholm', currency = 'SEK'
WHERE name = 'Sweden';
```

```
UPDATE Countries
SET population = population + 10
WHERE continent = 'EU';
```

Quiz: How many rows were updated in each case?

Deleting Data from Tables

- Deletes certain rows:

```
DELETE FROM Countries
WHERE name IN ('SmallArgentina', 'NewUruguay');
```

```
DELETE FROM Countries
WHERE continent = 'AM' AND area <= 0;
```

- Deletes every row!
  But the table will still exist...

```
DELETE FROM Countries;
```

Computing while Querying

- Performs a computation in a column:

```
SELECT name, population/area
FROM Countries;
```

- Gives the column a better name (aliases a column):

```
SELECT name, FLOOR(population/area) AS density
FROM Countries;
```

- Combines with other features:

```
SELECT name, ROUND(population/area) AS density
FROM Countries
WHERE continent = 'EU'
ORDER BY density DESC
LIMIT 3;
```

Aggregations

- Counts the countries in the table:

```
SELECT COUNT(*) FROM Countries;
```

```
SELECT COUNT(name) FROM Countries;
```

- Counts the number of larger countries:

```
SELECT COUNT(name) FROM Countries
WHERE population > 10000000;
```

- Counts the total number of countries and the sum of their population:

```
SELECT COUNT(name), SUM(population) FROM Countries;
```

Combining Group By and Aggregations

- Combines all rows with the same continent:

```
SELECT continent FROM Countries
GROUP BY continent;
```

- Computes the number of countries per continent:

```
SELECT continent, COUNT(name)
FROM Countries
GROUP BY continent;
```

- Computes the sum and average of the populations per continent:

```
SELECT continent, SUM(population),
       AVG(population)::Numeric(10,2) AS average
FROM Countries
GROUP BY continent;
```

Note: All attributes we select need to be used in the group-by or in an aggregate function!

More on Aggregations

- Counts countries using Euro:

```
SELECT COUNT(*) FROM Countries
WHERE currency = 'EUR';
```

- Counts number of small countries with the same currency:

```
SELECT currency, COUNT(*) FROM Countries
WHERE population < 10000000
GROUP BY currency;
```

WHERE works on individual rows, comes before GROUP BY (if any).

- Selects the currencies used in more than one country:

```
SELECT currency, COUNT(*) FROM Countries
GROUP BY currency
HAVING COUNT(name) > 1;
```

HAVING works on aggregated values, comes after GROUP BY.

Expressions in SELECT Fields

All the parts in the SELECT statement are optional:

```
SELECT 'Hello world!';
```

```
SELECT 2+3;
```

```
SELECT 2+3 AS answer;
```

```
SELECT 2+3 AS sum, 2*3 AS product;
```

```
SELECT 2+3 WHERE 2+2 = 5;
```

```
SELECT 2+3 WHERE true;
```

```
SELECT ;
```

Quiz: How many rows we get in each case?

Overview of Next Lecture

- More **SQL**:
  - Local definitions;
  - Views;
  - Set operations (unions, intersections, excepts);
  - Databases with several tables;
  - Database queries involving several tables;
  - Cross products and joins;
  - In and not in, Exists and not exists;
  - Dealing with empty values.

Reading:

Book: chapters 2, 6.1–6.5, 7.1–7.4, 8.1–8.2 and 8.5

Notes: chapter 2, 7.4.1–7.4.3 and 4.9