ME314 Introduction to Data Science and Big Data Analytics

Working with databases

Day 3 Lecture, 1 August 2018 - Kenneth Benoit
Plan for today

- Database systems
- JSON
- Relational data
- Normalisation
- SQL and SQLite
Database systems

Relational databases

- Transactions are always ACID (atomic, consistent, isolated, durable)
- Data needs to be defined

Non-relational databases

- Key-value storage types (e.g. Amazon DynamoDB) or document storage types (e.g. CouchDB, MongoDB)
- Sometime labelled as providing ACID transactions but often only eventually consistent
**JSON**

- a lightweight data-interchange format that is (supposedly!) easy for humans to read and write, and easy for machines to generate or parse
- follows conventions from Javascript, but is language-independent
- Example: Twitter data ([https://github.com/lse-st445/lectures/blob/master/week03/JSONexample.json](https://github.com/lse-st445/lectures/blob/master/week03/JSONexample.json))
- built on two structures:
  - A collection of name/value pairs
  - An ordered list of values
**object**

- unordered set of name/value pairs. An object begins with { and ends with }
- each name is followed by : and the name/value pairs are separated by ,

**array**

- an ordered collection of values
- begins with [ and ends with ]
- array values are separated by ,

**value**

- can be a "string", a number, or true, false, or null, or an object or array
- can be nested
strings in JSON

- a sequence of zero or more Unicode characters, wrapped in double quotes
- uses backslash escapes, e.g.
  - "\u2708\ufe0f" represents a ➥
  - "this is ""quoted"" represents "quoted"

In [10]: print("It's a bird, it's a \u2708\ufe0f!!")

It's a bird, it's a ➥!!
Relational data structures

- invented by E. F. Codd at IBM in 1970
- A relational database is a collection of data organized as a set of formally defined tables
- These tables can be accessed or reassembled in many different ways without having to reorganize the underlying tables that organize the data
- RDBMS: a relational database management system. Examples include: MySQL, SQLite, PostgreSQL, Oracle. MS Access is a lite version of this too.
- The standard user and application programmer interface to a relational database is structured query language (SQL)
Example: from Database of Parties, Elections, and Governments (DPEG) relational database

```sql
SELECT c.countryName, c.countryAbbrev, p.* FROM party AS p
LEFT JOIN country AS c ON p.countryID = c.countryID
```
Basic database concepts

(following on from last week)

- The basic unit is the database
- It might be stored on disk in a single file or a range of files managed by a server
- The database consists of tables which store actual data
- A table consists of at least one column whose name and data type need to be declared
- Data is stored in the rows of a table
Basic relational structures

- tables
  - also known as “relations”
  - tables define the forms of the data that are linked to other data through key relations

- keys: how tables are cross-referenced
  - primary key: an column in a table that uniquely identifies the remaining data in the table
  - foreign key: a field in a relational table that matches the primary key column of another table
  - join operations link tables in a structured query
Normalization

- Refers to the process of organizing a database into tables and columns, so that each table records a specific type of data, and only those columns that pertain to its topic are included
- Prevents duplication, and allows easier guarantee of data integrity
- Is more efficient
- Allows joined data to consist of merged tables, constructed through queries (more on this later)
Normal forms 1

“Normalizing” a database means creating a proper set of relations

First normal form: No Repeating Elements or Groups of Elements

<table>
<thead>
<tr>
<th>countryname</th>
<th>partyname</th>
<th>date</th>
<th>per108</th>
<th>per110</th>
</tr>
</thead>
<tbody>
<tr>
<td>175</td>
<td>Austria</td>
<td>FPÖ Freedom Party</td>
<td>199010</td>
<td>3</td>
</tr>
<tr>
<td>176</td>
<td>Austria</td>
<td>GA Green Alternative</td>
<td>199010</td>
<td>0</td>
</tr>
<tr>
<td>177</td>
<td>Austria</td>
<td>SPÖ Social Democratic Party</td>
<td>199010</td>
<td>5</td>
</tr>
<tr>
<td>178</td>
<td>Austria</td>
<td>ÖVP People's Party</td>
<td>199010</td>
<td>8</td>
</tr>
<tr>
<td>179</td>
<td>Austria</td>
<td>FPÖ Freedom Party</td>
<td>199410</td>
<td>1</td>
</tr>
<tr>
<td>180</td>
<td>Austria</td>
<td>LF Liberal Forum</td>
<td>199410</td>
<td>0</td>
</tr>
</tbody>
</table>

Here, this is violated because of the wide format of per108 and per110. To solve this, we have to move this to "long" format.
Normal forms 2

Second normal form: No Partial Dependencies on a Concatenated Key

<table>
<thead>
<tr>
<th>countryname</th>
<th>party</th>
<th>date</th>
<th>category</th>
<th>catcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Austria</td>
<td>42420</td>
<td>199010</td>
<td>per101</td>
<td>0</td>
</tr>
<tr>
<td>Austria</td>
<td>42110</td>
<td>199010</td>
<td>per101</td>
<td>0</td>
</tr>
<tr>
<td>Austria</td>
<td>42320</td>
<td>199010</td>
<td>per101</td>
<td>0</td>
</tr>
<tr>
<td>Austria</td>
<td>42520</td>
<td>199010</td>
<td>per101</td>
<td>5</td>
</tr>
<tr>
<td>Austria</td>
<td>42420</td>
<td>199410</td>
<td>per101</td>
<td>0</td>
</tr>
<tr>
<td>Austria</td>
<td>42421</td>
<td>199410</td>
<td>per101</td>
<td>0</td>
</tr>
</tbody>
</table>

Here, the format is still violated, because party 42420 is repeated. To solve this we need to create a party table and link to it using a foreign key.
Normal forms 3

Third normal form: No Dependencies on Non-Key Attributes. Every non-prime attribute of data in a table must be dependent on a primary key.

<table>
<thead>
<tr>
<th>countryname</th>
<th>party</th>
<th>date</th>
<th>category</th>
<th>catcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Austria</td>
<td>42420 199010</td>
<td>per101</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Austria</td>
<td>42110 199010</td>
<td>per101</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Austria</td>
<td>42320 199010</td>
<td>per101</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Austria</td>
<td>42520 199010</td>
<td>per101</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>Austria</td>
<td>42420 199410</td>
<td>per101</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Austria</td>
<td>42421 199410</td>
<td>per101</td>
<td>0</td>
</tr>
</tbody>
</table>

Here, this is violated because election data repeats across multiple values of the category count table, when it should have its own table.

Normalization: Example

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>SalesPerson</th>
<th>SalesOffice</th>
<th>OfficeNumber</th>
<th>Customer1</th>
<th>Customer2</th>
<th>Customer3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>Mary Smith</td>
<td>Chicago</td>
<td>312-555-1212</td>
<td>Ford</td>
<td>GM</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>John Hunt</td>
<td>New York</td>
<td>212-555-1212</td>
<td>Dell</td>
<td>HP</td>
<td>Apple</td>
</tr>
<tr>
<td>1005</td>
<td>Martin Hap</td>
<td>Chicago</td>
<td>312-555-1212</td>
<td>Boeing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table serves multiple functions:

- identifies the organization's salespeople
- listing the sales offices and phone numbers
- associates a salesperson with a sales office
- lists each salesperson's customers
Problems

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>SalesPerson</th>
<th>SalesOffice</th>
<th>OfficeNumber</th>
<th>Customer1</th>
<th>Customer2</th>
<th>Customer3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>Mary Smith</td>
<td>Chicago</td>
<td>312-555-1212</td>
<td>Ford</td>
<td>GM</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>John Hunt</td>
<td>New York</td>
<td>212-555-1212</td>
<td>Dell</td>
<td>HP</td>
<td>Apple</td>
</tr>
<tr>
<td>1005</td>
<td>Martin Hap</td>
<td>Chicago</td>
<td>312-555-1212</td>
<td>Boeing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What if we move the Chicago office to San Francisco? We would then need to update SalesOffice for every row of this table.

- What if John Hunt leaves the firm? Removing that employee means we lose the information about New York.

- What if Dell moves to having Mary Smith as a salesperson? Do we shift Customer2 and Customer3 left for John Hunt?
Problem: "insert anomaly"

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>SalesPerson</th>
<th>SalesOffice</th>
<th>OfficeNumber</th>
<th>Customer1</th>
<th>Customer2</th>
<th>Customer3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>Mary Smith</td>
<td>Chicago</td>
<td>312-555-1212</td>
<td>Ford</td>
<td>GM</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>John Hunt</td>
<td>New York</td>
<td>212-555-1212</td>
<td>Dell</td>
<td>HP</td>
<td>Apple</td>
</tr>
<tr>
<td>1005</td>
<td>Martin Hap</td>
<td>Chicago</td>
<td>312-555-1212</td>
<td>Boeing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>???</td>
<td>???</td>
<td>Atlanta</td>
<td><strong>312-555-1212</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We cannot create a new sales office, until we also know the name of a sales person, because EmployeeID is the **primary key**
- The primary key uniquely identifies the data item, so we must have one
Problem: "update anomaly"

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>SalesPerson</th>
<th>SalesOffice</th>
<th>OfficeNumber</th>
<th>Customer1</th>
<th>Customer2</th>
<th>Customer3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>Mary Smith</td>
<td>Chicago</td>
<td><strong>312-555-1212</strong></td>
<td>Ford</td>
<td>GM</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>John Hunt</td>
<td>New York</td>
<td>212-555-1212</td>
<td>Dell</td>
<td>HP</td>
<td>Apple</td>
</tr>
<tr>
<td>1005</td>
<td>Martin Hap</td>
<td>Chicago</td>
<td><strong>312-555-1212</strong></td>
<td>Boeing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If we have to update an attribute not uniquely associated with the primary key, then we have to change the entry multiple times
- Failure to make this update uniformly means we have inconsistent data (in this case, about the phone number of the sales office)
Problem: "deletion anomaly"

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>SalesPerson</th>
<th>SalesOffice</th>
<th>OfficeNumber</th>
<th>Customer1</th>
<th>Customer2</th>
<th>Customer3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>Mary Smith</td>
<td>Chicago</td>
<td>312-555-1212</td>
<td>Ford</td>
<td>GM</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>John Hunt</td>
<td>New York</td>
<td>212-555-1212</td>
<td>Dell</td>
<td>HP</td>
<td>Apple</td>
</tr>
<tr>
<td>1005</td>
<td>Martin Hap</td>
<td>Chicago</td>
<td>312-555-1212</td>
<td>Boeing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Deletion of one row removes more information than the item indexed by the primary key
- Here, removing John Hunt also removes information about the New York sales office, and his clients
**Problem: Searching or sorting**

<table>
<thead>
<tr>
<th>SalesStaff</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EmployeeID</strong></td>
</tr>
<tr>
<td>1003</td>
</tr>
<tr>
<td>1004</td>
</tr>
<tr>
<td>1005</td>
</tr>
</tbody>
</table>

- If we want to search for a specific customer, such as Ford, we have to search three fields (Customer1 - Customer3)
- How could we sort the list of customers?
- Where would we record client data, or link to the clients?
Example: Normalization

Non-Normalised Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>Movie</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Tom Cruise</td>
<td>53</td>
<td>Mission Impossible, Top gun</td>
<td>English</td>
</tr>
<tr>
<td>002</td>
<td>Brad Pitt</td>
<td>50</td>
<td>Seven, Troy</td>
<td>English</td>
</tr>
<tr>
<td>003</td>
<td>George Clooney</td>
<td>54</td>
<td>ER, Gravity</td>
<td>English</td>
</tr>
</tbody>
</table>

Example: solution (2NF)
Second Degree Normalised Table

**ACTOR**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Tom Cruise</td>
<td>53</td>
</tr>
<tr>
<td>002</td>
<td>Brad Pitt</td>
<td>50</td>
</tr>
<tr>
<td>003</td>
<td>George Clooney</td>
<td>54</td>
</tr>
</tbody>
</table>

**MOVIES**

<table>
<thead>
<tr>
<th>ID</th>
<th>Movie</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Mission Impossible</td>
<td>English</td>
</tr>
<tr>
<td>001</td>
<td>Top gun</td>
<td>English</td>
</tr>
<tr>
<td>002</td>
<td>Seven</td>
<td>English</td>
</tr>
<tr>
<td>002</td>
<td>Troy</td>
<td>English</td>
</tr>
<tr>
<td>003</td>
<td>ER</td>
<td>English</td>
</tr>
<tr>
<td>003</td>
<td>Gravity</td>
<td>English</td>
</tr>
</tbody>
</table>
Basic SQL syntax

- Defining data: CREATE TABLE
- Adding, changing, and deleting data: INSERT, UPDATE, DELETE
- Accessing data: SELECT
- Most functionality is part of the SELECT statement:
  - Filter: SELECT ... WHERE
  - Sort: SELECT ... ORDER BY
  - Aggregate: SELECT ... GROUP BY
  - Combining data: SELECT ... JOIN
- Adding constraints: CREATE CONSTRAINT
- Adding indexes: CREATE INDEX
SQL Syntax caveats

- SQL syntax is case-insensitive
- ; has to be added at the end of a line to terminate it (as in C-family languages, Javascript, ...)
- There is no implicit ordering of rows in SQL
Setting up the SQLite command line

Installation via anaconda
conda install sqlite

Connecting to a database
sqlite st445-week03.db

SQLite version 3.20.1 2017-08-24 16:21:36
Enter ".help" for usage hints.
sqlite>
Creating a table

CREATE TABLE table_name (column_name column_type [, column_name column_type])
;

Creating a table involves two things:

1. Giving the table a name consisting of alphanumeric characters and _
2. Giving each column a name and a data type

The SQL Standard defines several common data types and most SQL implementations provide additional ones:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT, BIGINT</td>
<td>Integer (4- and 8-byte).</td>
</tr>
<tr>
<td>FLOAT, DOUBLE</td>
<td>Single or double precision floating point number (4 or 8 bytes).</td>
</tr>
<tr>
<td>TEXT</td>
<td>String, stored using the database encoding (UTF-8, ...).</td>
</tr>
<tr>
<td>BLOB</td>
<td>Raw binary data.</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>True or false.</td>
</tr>
<tr>
<td>DATE, TIMESTAMP</td>
<td>Date and date-time.</td>
</tr>
</tbody>
</table>

Example
CREATE TABLE my_table (my_integer INT, my_float FLOAT, my_text TEXT);
Adding data to a table

You probably will not do this by hand but rather via a script. This is the topic of the class.

```
INSERT INTO table_name [(column_name [, column_name])] VALUES (value1 [, value 2]);
```

Example
```
INSERT INTO my_table VALUES (1, 1.3, 'abc') ;
INSERT INTO my_table (my_integer, my_float, my_text) VALUES (4, 4.3, 'def') ;
INSERT INTO my_table (my_integer, my_float) VALUES (8, 8.3) ;
```

INSERT documentation (https://www.postgresql.org/docs/current/static/dml-insert.html)
Changing or deleting data in a table

As above, you probably do not want to do this by hand.

```
UPDATE table_name SET column_name = value [, column_name = value] WHERE ... ;
DELETE FROM table_name WHERE ... ;
```

Example

```
UPDATE my_table SET my_float = 1.5 WHERE my_integer = 1 ;
UPDATE my_table SET my_integer = my_integer * 2 WHERE my_text = 'def' ;
DELETE FROM my_table WHERE my_integer = 8 ;
```
Retrieving data

Whenever data should be retrieved, the statement starts with SELECT.

```
SELECT * | column_name [AS new_name] {, column_name} FROM table_name [AS other_name] ;
```

* selects all columns of the table

**Example 1: Dummy table created above**
```
SELECT * FROM my_table ;
```

```
1|1.3|abc
4|4.3|def
```

**Example 2: Actual data**
```
SELECT * FROM lecture_TBD ;
```

TBD
• This will return all the rows in the table which is not practical for tables with many rows
• To just have a glance at the first X rows of the data:
  \[
  \text{SELECT * FROM lecture_TBD LIMIT 20 ;}
  \]

• This works the same as the head( dat , \ n = 20 ) function in R from last week
• **NB: There is no implicit ordering of rows in SQL**
• If the data is not explicitly ordered (will be explained shortly) there is no order whatsoever
• This is one of the main differences between SQL tables and tabular data in, e.g., an Excel sheet
Filtering data

`SELECT ... WHERE condition ;`

The `WHERE` clause is part and parcel of the power of SQL (`WHERE`, `JOIN`, and indexing)

You can test for many conditions:

- Equality: `WHERE column_name = value` (as used in the example above)
- Interval: `WHERE column_name BETWEEN lower_value AND higher_value`
- Set membership: `WHERE column_name IN (value1, value2, ...)`
- The set can be generated on the fly: `WHERE column_name IN (SELECT myvar FROM mytable)`
- Simple pattern match in strings: `WHERE string_column LIKE '_ab%'` (_ denotes one character, % denotes zero or more characters, see also GLOB and REGEX)
- Missing values: `WHERE column_name IS NULL`

Individual conditions can be composed with the logic operators AND, OR, and NOT
Sorting data

As mentioned previously:

*SQL does not implicitly order data.*

This means data can and will be returned in an arbitrary fashion (last accessed, query optimization, ...)

Work around this by explicitly ordering on variables.

```
SELECT ... ORDER BY column_name [DESC] [, column_name, ...] ;
```

**Example**

```
SELECT * FROM lecture_TBD WHERE TBD ;
```

Only works in SQLite because the variable rowid is created by default.

```
SELECT * FROM lecture_TBD ORDER BY rowid LIMIT 20 ;
```

¹ SQLite tables have a primary key by default called rowid (see here [https://sqlite.org/lang_createtable.html#primkeyconst](https://sqlite.org/lang_createtable.html#primkeyconst)). However, the data is not implicitly sorted according to the rowid.
Aggregating data

SELECT ... GROUP BY column_name;

- Data can be aggregated directly within SQL
- This works by defining a grouping variable
- The number of distinct values of the grouping variable defines the number groups
- All rows which share a value on the grouping variable are grouped together
- If there are several grouping variables, the unique combinations across all grouping variables defines the groups

- This statement puts some limits on the variables that you can select in the ... part
- Bare variables must be included in the GROUP BY part
- Other variables need to be aggregated with the help of a function
- The functions gets the all the rows in the group and returns a single, aggregated value
Aggregating data (cont’d)

SELECT ... GROUP BY column_name ;

Example

Count the number of rows per group:

SELECT country, COUNT(country) AS n FROM lecture_TBD GROUP BY country ;

Return minimum/maximum value per group:

SELECT country, min(XXX) AS XXX_min, max(XXX) AS XXX_max FROM lecture_TBD GROUP BY column_name ;
Joining tables

- The second pillar of SQL's (or more generally RDBM's) success
- Allows to combine two tables on the fly
- A single table thus does not need to contain all the relevant information
- If you know where

Types of JOINs:

- **INNER** JOIN retains only matching rows from tables 1 and 2
- **LEFT**  **OUTER** JOIN retains all rows from table 1 and matching rows from table 2
- **RIGHT**  **OUTER** JOIN retains matching rows from table 1 and all rows from table 2
Joining tables: **INNER JOIN**

<table>
<thead>
<tr>
<th>Student ID</th>
<th>Student name</th>
<th>Programme</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017829384</td>
<td>Anne</td>
<td>MSc in Research</td>
</tr>
<tr>
<td>2017891623</td>
<td>Tobias</td>
<td>MSc in DB Management</td>
</tr>
<tr>
<td>2017530293</td>
<td>Mette</td>
<td>MSc in DB Management</td>
</tr>
<tr>
<td>2017539281</td>
<td>Adam</td>
<td>MSc in Statistics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programme</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>MSc in Statistics</td>
<td>Department of Statistics</td>
</tr>
<tr>
<td>MSc in Social Research Methods</td>
<td>Department of Methodology</td>
</tr>
</tbody>
</table>

```sql
SELECT a.id, a.name, a.programme, b.department
FROM students AS a
INNER JOIN programmes AS b
ON a.programme = b.programme ;
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>programme</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017891623</td>
<td>Tobias</td>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>2017530293</td>
<td>Mette</td>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>2017539281</td>
<td>Adam</td>
<td>MSc in Statistics</td>
<td>Department of Statistics</td>
</tr>
</tbody>
</table>
Joining tables: LEFT OUTER JOIN

<table>
<thead>
<tr>
<th>Student ID</th>
<th>Student name</th>
<th>Programme</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017829384</td>
<td>Anne</td>
<td>MSc in Research</td>
</tr>
<tr>
<td>2017891623</td>
<td>Tobias</td>
<td>MSc in DB Management</td>
</tr>
<tr>
<td>2017530293</td>
<td>Mette</td>
<td>MSc in DB Management</td>
</tr>
<tr>
<td>2017539281</td>
<td>Adam</td>
<td>MSc in Statistics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programme</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>MSc in Statistics</td>
<td>Department of Statistics</td>
</tr>
<tr>
<td>MSc in Social Research Methods</td>
<td>Department of Methodology</td>
</tr>
</tbody>
</table>

The OUTER keyword is optional and often left out.

```
SELECT
    a.id, a.name, a.programme, b.department
FROM students AS a
LEFT JOIN programmes AS b
ON a.programme = b.programme;
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>programme</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017829384</td>
<td>Anne</td>
<td>MSc in Research</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>name</td>
<td>programme</td>
<td>department</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-----------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>2017891623</td>
<td>Tobias</td>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>2017530293</td>
<td>Mette</td>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>2017539281</td>
<td>Adam</td>
<td>MSc in Statistics</td>
<td>Department of Statistics</td>
</tr>
</tbody>
</table>
Joining tables: **RIGHT OUTER JOIN**

<table>
<thead>
<tr>
<th>Student ID</th>
<th>Student name</th>
<th>Programme</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017829384</td>
<td>Anne</td>
<td>MSc in Research</td>
</tr>
<tr>
<td>2017891623</td>
<td>Tobias</td>
<td>MSc in DB Management</td>
</tr>
<tr>
<td>2017530293</td>
<td>Mette</td>
<td>MSc in DB Management</td>
</tr>
<tr>
<td>2017539281</td>
<td>Adam</td>
<td>MSc in Statistics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programme</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>MSc in Statistics</td>
<td>Department of Statistics</td>
</tr>
<tr>
<td>MSc in Social Research Methods</td>
<td>Department of Methodology</td>
</tr>
</tbody>
</table>

SQLite does not support **RIGHT OUTER JOIN** so the actual syntax used for the output uses **LEFT JOIN** with rearranged tables.

```sql
SELECT  
a.id, a.name, b.programme, b.department  
FROM students AS a  
RIGHT JOIN programmes AS b  
on a.programme = b.programme ;
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>programme</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>name</td>
<td>programme</td>
<td>department</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>--------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>2017530293</td>
<td>Mette</td>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>2017891623</td>
<td>Tobias</td>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>2017539281</td>
<td>Adam</td>
<td>MSc in Statistics</td>
<td>Department of Statistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSc in Social Research Methods</td>
<td>Department of Methodology</td>
</tr>
</tbody>
</table>
Keys, constraints and indexes
• A primary or foreign key (as mentioned at the beginning of the lecture) is a unique identifier
• In SQL, keys are mainly made of constraints
• A constraint defines a check on data entered into a table
• If a row fails the check, SQL will signal an error and no data will be inserted in the table
• The main constraints used to build indexes are:
  - UNIQUE forces the values of a given column to be distinct
  - NOT NULL disallows NULL, i.e. missing, values
  - PRIMARY KEY is a shortcut for UNIQUE NOT NULL in most SQL implementations but not in SQLite (for historical reasons)
  - FOREIGN KEY ... foreign constraints in most SQL implementations and recent SQLite with foreign keys turned on
  - CHECK (EXISTS(SELECT 1 FROM foreign_table WHERE foreign_key=local_key)) foreign key constraint implemented by hand

• Declaring a primary key also automatically creates an index in most SQL implementations (though not in SQLite)
• An index is, generally speaking, a lookup table for a given column which allows sorting and filtering data to be quicker
• It comes with a maintenance cost because it needs to be updated whenever new data is inserted
Putting it all together to create normalised DBs

Setting up tables

```
CREATE TABLE students (id INT PRIMARY KEY NOT NULL, name TEXT);
CREATE TABLE programmes (id INT PRIMARY KEY NOT NULL, name TEXT, department TEXT);
CREATE TABLE allocation ()
    student_id INT NOT NULL,
    programme_id INT NOT NULL,
    FOREIGN KEY(student_id) REFERENCES students(id),
    FOREIGN KEY(programme_id) REFERENCES programme(id)
);
```
Putting it all together to create normalised DBs

Adding data

```
INSERT INTO students VALUES
    (2017829384, 'Anne'),
    (2017891623, 'Tobias'),
    (2017530293, 'Mette'),
    (2017539281, 'Adam');

INSERT INTO programmes VALUES
    (1, 'MSc in DB Management', 'Department of Databases'),
    (2, 'MSc in Statistics', 'Department of Statistics'),
    (3, 'MSc in Social Research Methods', 'Department of Methodology'),
    (4, 'MSc in Research', 'Department of Research');

INSERT INTO allocation VALUES
    (2017829384, 4),
    (2017891623, 1),
    (2017530293, 1),
    (2017539281, 2);
```

The programmes table contains the missing row for an "MSc in Research"
Putting it all together to create normalised DBs

Joining

```sql
SELECT 
s.id AS student_id,
s.name AS student_name,
p.name AS programme_name,
p.department 
FROM 
allocation AS a,
students AS s,
programmes AS p
WHERE 
a.student_id = s.id
AND 
a.programme_id = p.id
;
```

<table>
<thead>
<tr>
<th>student_id</th>
<th>student_name</th>
<th>programme_name</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017829384</td>
<td>Anne</td>
<td>MSc in Research</td>
<td>Department of Research</td>
</tr>
<tr>
<td>2017891623</td>
<td>Tobias</td>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>2017530293</td>
<td>Mette</td>
<td>MSc in DB Management</td>
<td>Department of Databases</td>
</tr>
<tr>
<td>2017539281</td>
<td>Adam</td>
<td>MSc in Statistics</td>
<td>Department of Statistics</td>
</tr>
</tbody>
</table>

(Implicit CROSS JOIN with manual implementation of the ON ... )
Putting it all together to create normalised DBs

Aggregating

```sql
SELECT
    p.department,
    COUNT(1) AS students
FROM allocation AS a
LEFT JOIN programmes AS p
    ON (a.programme_id = p.id)
GROUP BY p.department
;
```

<table>
<thead>
<tr>
<th>Department</th>
<th>students</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department of Databases</td>
<td>2</td>
</tr>
<tr>
<td>Department of Research</td>
<td>1</td>
</tr>
<tr>
<td>Department of Statistics</td>
<td>1</td>
</tr>
</tbody>
</table>

NB:

- Departments without students are omitted here
- Just to reiterate: The order of the rows is completely arbitrary
DB Browser (http://sqlitebrowser.org) to the rescue
DB Browser (http://sqlitebrowser.org): Executing queries