Database normalisation

- The learning objectives for this week are:
  - Knowing what the purpose of database normalisation
  - Knowing what is a functional dependency, a partial dependency and a transitive dependency
  - Knowing how to identify functional dependencies in a relation or table
  - Knowing the different normal form rules
  - Knowing how to formally check if a relation is in the Boyce-Codd normal form (BCNF)
  - Knowing how to decompose a relation into smaller relations if it is not in BCNF

Database normalisation

- *Database normalisation* is a formal technique of organizing data in a database in a way that *redundancy* and *incosistency* within the data is eliminated
- The objective of database normalisation is to ensure that:
  - Attributes with a *close logical relationship* (functional dependency) are found in the *same relation*
  - The relations do not display *hidden data redundancy*, which can cause update anomalies that violate database integrity
- The technique involves a set of normalisation rules that are defined as *normal forms* (1NF, 2NF, 3NF, BCDF...)

Database normalisation

- In a case of fixing an identified structural problem, normalisation involves *decomposing a relation into less redundant (and smaller) relations* without losing information
- When an *ER model is well designed*, the resulting correctly derived relations won't normally have such structural problems and they will meet the criteria of database normalisation
- Normalisation of candidate relations derived from ER diagrams is accomplished by analysing the *functional dependencies* (FDs) associated with those relations

Functional depedency

- *Functional dependency* (FD) describes the *relationship between attributes* in a relation
- With functional dependencies, we are interested in properties of the data that are true for *all the time*
- For example, if the *student number is unique*, the following property is true all the time:

  > The surname for a student whose student number is "a12345" is "Smith"

- So, *all the time* it is true that there is only one surname for each student
- By contrast, the following property might to be true for a sample set of students, but it is not true for all the time:

  > There is exactly one student whose surname is "Smith"

Functional dependency

- A functional dependency occurs when attribute A in a relation *uniquely determines* attribute B
- In other words: for each value of A there is *exactly one value* of B and that *holds all the time*. This can be written as `A → B`
- The *determinant* of a functional dependency refers to the attribute, or group of attributes, on the *left-hand side* of the arrow. In `A → B`, A is the determinant of B.
- On the *right-hand side*, there's the *dependent*. In `A → B`, B is the dependent of A.

Example of functional dependency

- Let's suppose that each student has a unique student number. In the relation below, *studentnumber* uniquely determines *surname* and *firstname*. That is, *studentnumber is the determinant of surname and firstname*:

```
Student (studentnumber, surname, firstname)
```

- In this example, there are the following two functional dependencies:
    - `studentnumber → surname`
    - `studentnumber → firstname`

Example of functional dependency

- Let's suppose the following table occurrence:

studentnumber	surname	firstname
a12345	Smith	John
a14444	Smith	Susan
a15555	Jones	Susan

- The *functional dependency* `studentnumber → surname` guarantees that the query below (that uses an existing student number) returns exactly one surname and that holds all the time:

```
SELECT surname FROM Student WHERE studentnumber = 'a12345'
```

Example of functional dependency

- By contrast, the query below may return several student numbers:

  ```sql
  SELECT studentnumber FROM Student WHERE surname = 'Smith'
  ```

- The latter type of dependency is called *multi-valued dependency* and it can be written as follows: `surname ->> studentnumber`

Identifying undesired data redundancy

- Relations that *do not have* undesired data redundancy ✅, *each determinant is a candidate key* (an unique attribute that is suitable for being the primary key)
- In such case *all arrows are arrows out of whole candidate keys* (simple or composite key)
- Let's consider the following relation *without data redundancy*:

```
CourseOffering (coursecode, offeringnumber, startdate, teachernumber)
```

- In this relations there's for example the following functional depedency:
  - ✅ `{coursecode, offeringnumber} → startdate, teachernumber`

Identifying undesired data redundancy

- Relations that *have* undesired data redundancy ❌, *there is a determinant that is not a candidate key*
- In such case *there is on arrow that is not an arrow out of a whole candidate key*
- Let's consider the following relation *with data redundancy*:

```
CourseOffering (
      coursecode,
      offeringnumber,
      coursename,
      startdate,
      teachernumber,
      surname
  )
```

Identifying undesired data redundancy

- In this relations there's for example the following functional depedencies:
  - ✅ `{coursecode, offeringnumber} → coursename, startdate, teachernumber, surname`
  - ❌ `coursecode → coursename`
  - ❌ `teachernumber → surname`
- In functional dependencies `coursecode → coursename` and `teachernumber → surname`, the determinants are not candidate keys
- With such functional dependencies, the relation has redundant data
- For example the teacher's surname is repeated unnecessarily, which can cause consistency issues for example when a teacher's surname is updated
- Instead, the teacher's information should be in a *separate relation*

Calculated attributes

- We *should not include* attributes in a relation that we can *derive* from other relations or *calculate*
- For example, let's suppose that the firm's total budget is the total of department budgets
- Therefore, *totalbudget* is a calculated attribute in the *Firm* relation
- The value of *totalbudget* should change whenever any department budget is changed in the firm
- From the data redundancy and integrity viewpoint, we have a problem here because total budget exists twice in the design:

```
Firm (firmno, firmname, totalbudget ❌)
Depertmant (deptno, deptname, deptbudget, firmno)
    FK (firmno) REFERENCES Firm (firmno)
```

Calculated attributes

- We shouldn't have the *totalbudget* attribute in the *Firm* relation, instead we can calculate it with the following query:

```
SELECT SUM(deptbudget) as totalbudget FROM Department
WHERE firmno = 'a1122'
```

Different kind of functional dependencies

- Functional dependencies can be categorized in the following categories:
  - *Non-trivial* and *trivial* functional dependencies
  - *Partial* and *full* functional dependencies
  - *Transitive* and *non-transitive* functional dependencies

Non-trivial and trivial functional dependencies

- `A → B` is *trivial functional dependency* if B is a subset of A
- `A → B` is non-trivial functional dependency if B is not a subset of A
- Let's consider the *CourseOffering* relations:

```
CourseOffering (coursecode, offeringno, startdate)
```

- In the relation, `{coursecode, offeringno} → startdate` is a *non-trivial functional dependency*, because `startdate` is not a subset of `{coursecode, offeringno}`
- These, on the other are *trivial functional dependencies* of the relation:
  - `{coursecode, offeringno} → coursecode`
  - `{coursecode, offeringno} → {coursecode, offeringno}`
- In normalisation considerations we are only focusing on non-trivial functional dependencies