

Database normalisation

- The learning objectives for this week are:
 - Knowing 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*

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 dependency

- *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 $A \rightarrow B$
- The *determinant* of a functional dependency refers to the attribute, or group of attributes, on the *left-hand side* of the arrow. In $A \rightarrow B$, A is the determinant of B.
- On the *right-hand side*, there's the *dependent*. In $A \rightarrow B$, 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


- $\{A, B\} \rightarrow C$ means that *A and B together uniquely determine C*. For example,
 $\{\text{course_code}, \text{implementation_number}\} \rightarrow \text{start_date}$
- $A \rightarrow B, C, D$ means that *A uniquely determines B, C, and D*. For example,
 $\text{course_code} \rightarrow \text{course_name}, \text{language}, \text{credits}$

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 dependency:
 -  {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 an 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 dependencies:
 -  `{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

- $A \rightarrow B$ is *trivial functional dependency* if B is a subset of A
- $A \rightarrow B$ 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, $\{\text{coursecode}, \text{offeringno}\} \rightarrow \text{startdate}$ 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:
 - $\{\text{coursecode}, \text{offeringno}\} \rightarrow \text{coursecode}$
 - $\{\text{coursecode}, \text{offeringno}\} \rightarrow \{\text{coursecode}, \text{offeringno}\}$
- In normalisation considerations we are only focusing on *non-trivial functional dependencies*

Partial and full functional dependencies

- Let's consider the following relation schema:

```
ClubMembership (empno, clubno, clubname, joindate)
```

- In this relation the functional dependency $\{\text{empno}, \text{clubno}\} \rightarrow \text{joindate}$ is a *full functional dependency*, because none of the determinant attributes `empno` or `clubno` determine `joindate` *individually*
- That is, functional dependencies $\text{empno} \rightarrow \text{joindate}$ or $\text{clubno} \rightarrow \text{joindate}$ *do not exist* in the relation
- In contrast, the functional dependency $\{\text{empno}, \text{clubno}\} \rightarrow \text{clubname}$ is a *partial functional dependency* because relation $\text{clubno} \rightarrow \text{clubname}$ *does exist* in the relation

Transitive and non-transitive functional dependencies

- Let's consider the following relation schema:

```
Employee (empno, surname, firstname, deptno, deptname)
```

- In this relations there's functional dependencies $\text{empno} \rightarrow \text{deptno}$ and $\text{deptno} \rightarrow \text{deptname}$
- This means, that deptname is *transitively dependent* on empno via deptno
- In contrast, $\text{empno} \rightarrow \text{surname}$ is a *non-transitive dependency*, because surname is *not a determinant of any functional dependency* in the relation

Boyce-Codd Normal Form (BCNF)

- *Normal form* refers to a set of normalisation rules that a database relation should follow in order to be considered "normalized" and thus *well-organized*
- The *Boyce-Codd Normal Form (BCNF)* is one such normal form
- We simplify the rules of BCNF we will have the following limitations during the course:
 - We only focusing on *non-trivial functional dependencies*
 - Instead of including any superkeys in our analysis, we narrow the analysis to *candidate keys*
 - We do not allow any attribute that *does not have a determinant* within the relation




Boyce-Codd Normal Form (BCNF)

- With these limitations the BCNF has the following rules for a relation:
 - Each determinant is a candidate key
 - All attribute values are atomic (single values)
 - There is a determinant for each attribute that is not contained in a candidate key

Boyce-Codd Normal Form (BCNF)

- Let's consider the following relation:

Teacher (teacherno, firstname, surname)

- `teacherno → firstname, surname` is the only *non-trivial functional dependency* in the relation
-  Each determinant is a candidate key
-  All attribute values are atomic (single values)
-  There is a determinant for each attribute that is not contained in a candidate key
- Thus, *the relation is in BCNF*

Boyce-Codd Normal Form (BCNF)

- Let's consider the following relation:

CourseGrade (course_code, studentno, firstname, surname, grade)

- `studentno → firstname, surname` is one of the *non-trivial functional dependencies* in the relation
- ✗ `studentno` is *not a candidate key* in the relation (so each determinant is *not* a candidate key)
- Thus, *the relation is not in BCNF*

Turning a relation into Boyce-Codd Normal Form

- To convert a *non-BCNF relation to BCNF*, we must decompose the relation in two steps
- Step 1: Find a *non-trivial functional dependency* $X \rightarrow Y$ which violates the BCNF rule (find a determinant that is *not a candidate key*)
- Step 2: Split the original relation in two relations as follows:
 - Create a new relation with all attributes (for example both X and Y) from the dependency. X will be the primary key in the new relation
 - Remove Y attribute(s) from the original relation and leave X in the original relation to act as a foreign key.
- We repeat the steps above until all of our relations are in BCNF

Turning a relation into Boyce-Codd Normal Form

- Let's consider the following relation candidate:

```
CourseOffering (coursecode, offeringno,  
                coursename, startdate, teacherno, surname, firstname)
```

- In the first step, we identify the *non-trivial functional dependencies*:
 - {coursecode, offeringno} → coursename, startdate, teacherno, surname, firstname
 - coursecode → coursename
 - teacherno → surname, firstname
- Then, we identify functional dependencies where the determinant is *not a candidate key*
- There's two such cases: coursecode → coursename and teacherno → surname, firstname

Turning a relation into Boyce-Codd Normal Form

- In the second step, to solve these two cases we split the original relation two times
- With `coursecode → coursename` we create a new relation Course with attributes `coursecode` and `coursename`
- The determinant, the `coursecode` will be the primary key for the relation. We'll get the following relation:

```
Course (coursecode, coursename)
```

- Finally, we remove the `coursename` from the CourseOffering relation and leave `coursecode` as a foreign key:

```
CourseOffering (coursecode, offeringno,  
                startdate, teacherno, surname, firstname)  
    FK (coursecode) REFERENCES Course(coursecode)
```


Turning a relation into Boyce-Codd Normal Form

- We will repeat the same process with `teacherno → surname, firstname` and the final relations are the following:

```
Course (coursecode, coursename)
Teacher (teacherno, surname, firstname)
CourseOffering (coursecode, offeringno, startdate, teacherno)
    FK (coursecode) REFERENCES Course(coursecode)
    FK (teacherno) REFERENCES Teacher(teacherno)
```

- Finally, we *check the decomposed relations*
- In each relation above each determinant is a candidate key and each attribute non-candidate-key attribute has a determinant
- Therefore, the *relations are in BCNF* and we have successfully removed all the undesired redundancy from the design

Normal form rules (1NF, 2NF, 3NF, BCNF)

- During the course we will cover the most common normal forms: *first normal form* (1NF), *second normal form* (2NF), *third normal form* (3NF) and *Boyce-Codd normal form* (BCNF)
- Each normal form from 1NF to BCNF adds more rules to the previous normal form
- For example, the 2NF includes all rules of the 1NF and additional rules

First normal form (1NF)

- A relation is in the *first normal form* (1NF) if the following rules apply:
 - All attributes in a relation *must have atomic values*. No multi-valued attributes are allowed
 - A relation *must have a primary key* and all its *attributes must be dependent on the primary key*

Second normal form (2NF)

- A relation is in the *second normal form* (2NF) if the following rules apply:
 - Relation is in 1NF
 - Relation has no *partial functional dependencies*, meaning that there is no *part of a candidate key* that uniquely determines a *non-candidate-key* attribute
- Let's consider the following relation:

```
ClubMembership (empno, clubno, clubname, joindate)
```

- The relation has a *partial functional dependency* $\{\text{empno}, \text{clubno}\} \rightarrow \text{clubname}$, because the functional dependency $\text{clubno} \rightarrow \text{clubname}$ exists in the relation
- That is, the relation *is not in 2NF*

Third normal form

- A relation is in the *third normal form* (3NF) if the following rules apply:
 - Relation is in 1NF
 - Relation has no functional dependency between two *non-candidate-key* attributes, meaning no *non-candidate-key* attribute is allowed to be *transitively* dependent on any *candidate key* within the relation
- Let's consider the following relation schema:

```
Employee (empno, surname, firstname, deptno, deptname)
```

- The relation has a functional dependency `deptno → deptname` between two *non-candidate-key* attributes, causing `deptname` to be *transitively dependent* on `empno` via `deptno`
- That is, the relation *is not in 3NF*

Summary

- *Database normalisation* is a formal technique of organizing data in a database in a way that *redundancy* and *incosistency* within the data is eliminated
- We analyze a set *normalisation rules* to determine if a relation is in a certain *normal form* (1NF, 2NF, 3NF, BCNF)
- Normalisation rules determine what kind *functional dependencies* the relation can have
- We can turn a non-BCNF relation into BCNF relations by decomposing the relation