Ian Kenny

November 7, 2016

# Databases

# Lecture 6

# In this lecture

- Database design theory: functional dependencies

# Introduction

To date we have looked at how a database might be designed using a process of ER modelling.

The focus of this process is to produce a design that meets an enterprise's needs for a database system. Hence, it is based on their business rules (enterprise constraints) and on their other requirements.

It is hoped that this will produce a design that does suit the enterprise's needs, but it is possible that it may produce a sub-optimal logical design, or one that creates the potential for database anomalies to be introduced. That is, it may permit errors or be a wasteful design in some way.

# Introduction

We will now consider how such waste or anomalies can be avoided, by using ideas from database design theory. In particular, we will look at *functional dependencies*.

We will consider a formal definition but, briefly, a functional dependency is a relationship between attributes in a relation. A functional dependency exists between two sets of attributes if a certain value in one set is always associated with a certain value in the other. A *genuine* functional dependency is a constraint on a relation. We will return to the meaning of the word 'genuine' shortly.

# Introduction

The existence of *undesirable* functional dependencies in a relation can cause problems. Not all functional dependencies are undesirable, as we shall see. Indeed, functional dependencies form the basis of keys, the existence of which is highly desirable.

The process of database *normalisation* consists of removing undesirable functional dependencies. However, sometimes a database will be *denormalised*, reintroducing functional dependencies, usually for reasons of efficiency.

# Example problem

In order to motivate this discussion, let us consider an example enterprise.

Consider an enterprise in which employees work on projects. Each employee has an id number, name, job title and salary. They can work on multiple projects. Each project has one manager. Employees can, therefore, have a manager for each project they work on. These managers don't need to be the same person. Managers manage one project. Each project has a length (its duration in time) and a cost.

Our first attempt at capturing this information in a relation might look like this

**employee**(empid, name, job, salary, project, manager, prlength, prcost)

# A sample relation

**employee**(empid, name, job, salary, project, manager, prlength, prcost)

Of course, you may already have noted that this relation will have some issues ... For example, it will lead to redundant information being stored. If an employee starts work on more than one project, their empid, name, job and salary will be duplicated in the table. This means that, currently, the single attribute *empid* cannot be a primary key for the relation as might be generally expected.

The schema is currently just a list of attributes. No other information about this schema is stated, e.g. primary key, etc.

# A sample instance of the relation

| empid | name | job | salary | project | manager | prlength | prcost |
|---|---|---|---|---|---|---|---|
| 1 | Jones | engineer | 35000 | Gherkin | Davis | 5 | 500m |
| 2 | Wilson | sales | 28000 | Tunnel | Fulton | 6 | 250m |
| 1 | Jones | engineer | 35000 | Tunnel | Fulton | 6 | 250m |
| 3 | Peters | tech | 24000 | Gherkin | Davis | 5 | 500m |
| 3 | Peters | tech | 24000 | Dome | Mandelson | 2 | 2000m |
| 4 | Price | runner | 17500 | Dome | Mandelson | 2 | 2000m |

Employee

We can see that *empid* is not a key for this relation. No single attribute is. How do we decide on a key? Functional dependencies will help with that.

# Problems with this relation

We will now consider problems with this relation design. They are usually categorised as

- Update anomalies.
- Insertion anomalies.
- Deletion anomalies.

# Update anomalies

Update anomalies occur when the data is changed erroneously in an existing tuple or tuples leading to anomalous data in the database.

In our current design this can happen due to the existence of *redundant* data.

# Update anomalies

Consider the case in which Jones gets a pay increase.

| empid | name | job | salary | project | manager | prlength | prcost |
|---|---|---|---|---|---|---|---|
| 1 | Jones | engineer | *40000* | Gherkin | Davis | 5 | 500m |
| 2 | Wilson | sales | 28000 | Tunnel | Fulton | 6 | 250m |
| 1 | Jones | engineer | 35000 | Tunnel | Fulton | 6 | 250m |
| 3 | Peters | tech | 24000 | Gherkin | Davis | 5 | 500m |
| 3 | Peters | tech | 24000 | Dome | Mandelson | 2 | 2000m |
| 4 | Price | runner | 17500 | Dome | Mandelson | 2 | 2000m |
| 5 | Dollis | designer | 45000 | Airport | Craig | 5 | 1000m |

Employee

Because the information about Jones' salary is stored once for each row he is involved in, if any of his attribute values changes, it must be changed in all of the rows that involve him otherwise we end up with anomalous data. In this case we have an incomplete update that could cause problems.

# Update anomalies

Consider the case in which Mandelson is sacked.

| empid | name | job | salary | project | manager | prlength | prcost |
|---|---|---|---|---|---|---|---|
| 1 | Jones | engineer | 35000 | Gherkin | Davis | 5 | 500m |
| 2 | Wilson | sales | 28000 | Tunnel | Fulton | 6 | 250m |
| 1 | Jones | engineer | 35000 | Tunnel | Fulton | 6 | 250m |
| 3 | Peters | tech | 24000 | Gherkin | Davis | 5 | 500m |
| 3 | Peters | tech | 24000 | Dome | *null* | 2 | 2000m |
| 4 | Price | runner | 17500 | Dome | Mandelson | 2 | 2000m |
| 5 | Dollis | designer | 45000 | Airport | Craig | 5 | 1000m |

Employee

Again, the update is incomplete.

# Insertion anomalies

Insert anomalies can occur when we don't have all of the information needed at the point at which we need to enter the record in the relation, or when that information is not required for a particular entry. We are forced to specify values or specify null (if allowed) for the attributes for which we don't have data.

We could also enter the information incorrectly.

# Insertion anomalies

Consider the case of a new employee called Johnson. We don't yet know which project he will work on.

| empid | name | job | salary | project | manager | prlength | prcost |
|---|---|---|---|---|---|---|---|
| 1 | Jones | engineer | 35000 | Gherkin | Davis | 5 | 500m |
| 2 | Wilson | sales | 28000 | Tunnel | Fulton | 6 | 250m |
| 1 | Jones | engineer | 35000 | Tunnel | Fulton | 6 | 250m |
| 3 | Peters | tech | 24000 | Gherkin | Davis | 5 | 500m |
| 3 | Peters | tech | 24000 | Dome | Mandelson | 2 | 2000m |
| 4 | Price | runner | 17500 | Dome | Mandelson | 2 | 2000m |
| 5 | Dollis | designer | 45000 | Airport | Craig | 5 | 1000m |
| 6 | Johnson | enginer | 33000 | null | null | 0 | null |

Employee

It is perhaps correct that we specify null for the project if we don't know which one Johnson will be working on yet, but we are forced to specify null for some of the other values simply because we need to specify some value or null for those attributes. For example, we need to state that we don't know the project cost every time we add a new employee who is not yet assigned to a project.

# Insertion anomalies

| empid | name | job | salary | project | manager | prlength | prcost |
|---|---|---|---|---|---|---|---|
| 1 | Jones | engineer | 35000 | Gherkin | Davis | 5 | 500m |
| 2 | Wilson | sales | 28000 | Tunnel | Fulton | 6 | 250m |
| 1 | Jones | engineer | 35000 | Tunnel | Fulton | 6 | 250m |
| 3 | Peters | tech | 24000 | Gherkin | Davis | 5 | 500m |
| 3 | Peters | tech | 24000 | Dome | Mandelson | 2 | 2000m |
| 4 | Price | runner | 17500 | Dome | Mandelson | 2 | 2000m |
| 5 | Dollis | designer | 45000 | Airport | Craig | 5 | 1000m |
| 6 | Johnson | enginer | 33000 | null | null | 0 | null |

Employee

Note we also made a spelling mistake: *enginer*. This would obviously mess up SQL queries. We could be forced to specify arbitrary values too, if the NOT NULL constraint is specified (as shown here for *prlength*).

# Deletion anomalies

Deletion anomalies occur when data is lost from a relation when a row is deleted - data that shouldn't be lost.

# Deletion anomalies

What happens if Dollis leaves the company and we delete that record? Currently, that record is the only place that the data about the Airport project is stored. It is also the only place where the salary of a 'designer' is stored, which would be relevant if the salaries are standardised and **all** designers would receive that salary.

| empid | name | job | salary | project | manager | prlength | prcost |
|---|---|---|---|---|---|---|---|
| 1 | Jones | engineer | 35000 | Gherkin | Davis | 5 | 500m |
| 2 | Wilson | sales | 28000 | Tunnel | Fulton | 6 | 250m |
| 1 | Jones | engineer | 35000 | Tunnel | Fulton | 6 | 250m |
| 3 | Peters | tech | 24000 | Gherkin | Davis | 5 | 500m |
| 3 | Peters | tech | 24000 | Dome | Mandelson | 2 | 2000m |
| 4 | Price | runner | 17500 | Dome | Mandelson | 2 | 2000m |
| 5 | Dollis | designer | 45000 | Airport | Craig | 5 | 1000m |

Employee

# Anomalies

We are getting these anomalies because of the redundancy created by the relational schema for Employee.

We will now consider a theoretical approach that enables us to analyse our designs for some of the issues just raised.

# Functional dependencies

As stated earlier, informally, a functional dependency exists between two sets of attributes if a certain value in one set is always associated with a certain value in the other. We can set that the first set of attributes *determines* the second. For two attributes A and B we write

$A \rightarrow B$

To mean that attribute A determines B, i.e. A is the determinant of B.

In the language of functional dependencies (FDs), we say B is functionally dependent on A.

# Functional dependencies

If we need to consider sets of attributes we can write

$\bar{A} \rightarrow \bar{B}$

Where $\bar{A}$ and $\bar{B}$ are sets of attributes.

If a set of attributes functionally determines a single attribute, we can write

$\bar{A} \rightarrow B$

And, if a single attribute functionally determines a set of attributes, we can write

$A \rightarrow \bar{B}$

# Definition

For two tuples $t$ and $u$ in relation $R$, $\bar{A} \rightarrow \bar{B}$ means

$\forall t, u \in R$

$$t[A_1, A_2, ..., A_n] = u[A_1, A_2, ..., A_n] \implies t[B_1, B_2, ..., B_m] = u[B_1, B_2, ..., B_m]$$

If the tuples have equal values for attributes $\bar{A}$ then they will have equal values for attributes $\bar{B}$.

# Functional dependencies

If we believe, using knowledge of the domain, that we have the following functional dependency

project $\rightarrow$ manager

Hence, every time we have the same value for project we will also have the same value for manager, then we can write

$\forall t, u \in \mathit{employee}$

$t.\mathit{project} = u.\mathit{project} \implies t.\mathit{manager} = u.\mathit{manager}$

# Functional dependencies

If we believe, using knowledge of the domain, that we have the following functional dependency

project $\rightarrow$ prlength, prcost

Hence, every time we have the same value for project we will also have the same value for prlength and prcost, then we can write

$\forall t, u \in \mathit{employee}$

$t.\mathit{project} = u.\mathit{project} \implies t[\mathit{prlength}, \mathit{prcost}] = u[\mathit{prlength}, \mathit{prcost}]$

# Types of functional dependency

We can specify three types of functional dependency (FD).

- Trivial FDs.
- Non-trivial FDs.
- Completely non-trivial FDs.

# Trivial FDs

Trivial FDs are not interesting.

They can be expressed as

$\bar{A} \rightarrow \bar{B}, \bar{B} \subseteq \bar{A}$

For example: project, manager $\rightarrow$ manager

This includes the case

$\bar{A} \rightarrow \bar{A}$

For example: project$\rightarrow$ project

# Non-trivial FDs

Non-trivial FDs can be expressed as

$\bar{A} \rightarrow \bar{B}, \bar{B} \nsubseteq \bar{A}$

For example: project, manager $\rightarrow$ manager, prlength

# Completely non-trivial FDs

These are the most interesting cases.

They can be expressed as

$\bar{A} \rightarrow \bar{B}, \bar{A} \cap \bar{B} = \varnothing$

For example: project $\rightarrow$ manager, prlength

# Partial FDs

A partial functional dependency is a functional dependence where attributes in the determinant can be removed but the FD still holds, for example.

$\bar{A}\bar{B} \rightarrow \bar{C}, \bar{A} \rightarrow \bar{C}$

$\bar{A}\bar{B} \rightarrow \bar{C}$ is a partial dependency.

For example, empid, manager$\rightarrow$ project, manager$\rightarrow$ project

# Rules for functional dependencies

Certain rules can be applied to FDs

- Splitting rule.
- Combining rule.
- Transitive rule.

# Splitting rule

The splitting rule states that if set of attributes A functionally determines a set of attributes B then A also functionally determines the individual elements of B.

$\bar{A} \rightarrow B_1, B_2, ..., B_m$

$\Longrightarrow$

$\bar{A} \rightarrow B_1$
$\bar{A} \rightarrow B_2$
...
$\bar{A} \rightarrow B_m$

This makes sense. If $\bar{A}$ functionally determines all of the set $\bar{B}$ then it must functionally determine each element of $\bar{B}$.

For example if we have the FD

project $\rightarrow$ prlength, prcost

Then we also have

project $\rightarrow$ prlength and project $\rightarrow$ prcost

# Splitting rule

What about splitting the left-hand side? Would that also work?

$A_1, A_2, ..., A_n \rightarrow \bar{B}$

$\Longrightarrow$

$A_1 \rightarrow \bar{B}$
$A_2 \rightarrow \bar{B}$
...
$A_3 \rightarrow \bar{B}$

No, this won't work. For example, it would be like saying that if

studentid, studentname $\rightarrow$ studentaddress

Then

studentname $\rightarrow$ studentaddress. Which may not be true.

# Combining rule

The combining rule states that if a set of attributes functionally determines individual attributes then it functionally determines a combination of those attributes.

$\bar{A} \rightarrow B_1$
$\bar{A} \rightarrow B_2$
...
$\bar{A} \rightarrow B_m$

$\Longrightarrow$

$\bar{A} \rightarrow B_1, B_2, ..., B_m$

For example, if

studentid, studentname $\rightarrow$ studentaddress

And

studentid, studentname $\rightarrow$ tutorgroup

Then

studentid, studentname $\rightarrow$ studentaddress, tutorgroup.

# Transitive rule

The transitive rule states that if a set of attributes A functionally determines a set of attributes B, and B functionally determines set of attributes C, then A functionally determines C.

$\bar{A} \rightarrow \bar{B}, \bar{B} \rightarrow \bar{C} \implies \bar{A} \rightarrow \bar{C}$

For example, if

studentname $\rightarrow$ studentid

And

studentid $\rightarrow$ tutorgroup

Then studentname$\rightarrow$tutorgroup.

# Armstrong's axioms

Armstrong's axioms are a sound and complete set of axioms for deriving FDs from other FDs (they will generate only correct FDs and will generate all of them). They are

- The Axiom of Reflexivity.
- The Axiom of Augmentation.
- The Axiom of Transitivity.

We have already seen two of these axioms but they are presented again here for completeness.

# The Axiom of Reflexivity

We have actually already seen this one. It is the axiom that gives us the trivial dependencies and is stated as

$\bar{A} \rightarrow \bar{B}, \bar{B} \subseteq \bar{A}$

# The Axiom of Augmentation

This axiom states that we can add the same attributes to both side of an FD

$\bar{A} \rightarrow \bar{B}$

$\Longrightarrow$

$\bar{A}\bar{C} \rightarrow \bar{B}\bar{C}$

# The Axiom of Transitivity

Again, we have already seen this axiom

$$\bar{A} \to \bar{B}, \bar{B} \to \bar{C} \implies \bar{A} \to \bar{C}$$

# Finding functional dependencies

How do we identify FDs?

We can analyse the data itself for dependencies.

Consider again the employee table.

# Finding FDs

Which FDs can you see?

| empid | name | job | salary | project | manager | prlength | prcost |
|---|---|---|---|---|---|---|---|
| 1 | Jones | engineer | 35000 | Gherkin | Davis | 5 | 500m |
| 2 | Wilson | sales | 28000 | Tunnel | Fulton | 6 | 250m |
| 1 | Jones | engineer | 35000 | Tunnel | Fulton | 6 | 250m |
| 3 | Peters | tech | 24000 | Gherkin | Davis | 5 | 500m |
| 3 | Peters | tech | 24000 | Dome | Mandelson | 2 | 2000m |
| 4 | Price | runner | 17500 | Dome | Mandelson | 2 | 2000m |
| 5 | Dollis | designer | 45000 | Airport | Craig | 5 | 1000m |

Employee

# Finding FDs

We appear to have (at least)

empid→ name
empid→ job
empid→ name, job
empid→ salary
empid→ name, job, salary
name → empid
name → job
name → salary
name, job→ salary
name, job, salary→empid
job → salary
job → name
empid, manager → project
project → manager

project → prlength
project → prcost
project → manager, prlength
project → manager, prlength, cost
prlength→ prcost
prlength→ manager
prlength→ project
manager → prlength
manager → project
manager → prcost
manager → project
prcost → project
prcost → prlength
prcost → manager

# Finding FDs

We could apply the rules and axioms we have already seen in order to derive other FDs from this list: FDs that are implied by this list of FDs.

We can also use the rules to remove FDs from the list because there is more than likely a large degree of redundancy in the list.

# Finding functional dependencies

Do these FDs make sense to you?

Some of them do. Many don't.

The issue is that, yes, we can formulate all of the FDs this way but many won't actually reflect the enterprise being modelled.

Some of the apparent dependencies may arise by 'accident'. They just happen to be implied by the data. It is possible that we could add some data that makes the 'FD' no longer apply. In which case, it wasn't an FD for the relation.

Some dependencies are real in the sense that they say something about the enterprise but they don't capture what we are actually looking for. Some of them follow from other FDs, using the rules. Others are real dependencies.

# Finding functional dependencies

We might find other FDs using the rules. Consider the FDs

1. empid$\rightarrow$ name
2. empid$\rightarrow$ job
3. empid$\rightarrow$ name, job

3. follows from 1. and 2. through the combination rule.

Conversely, 1. and 2. follow from 3. through the splitting rule.

# Finding functional dependencies

Consider the FDs

1. name$\rightarrow$ job
2. job$\rightarrow$ salary
3. name$\rightarrow$ salary

3. follows from 1. and 2. through the transitive rule.

We are not claiming at this point that this results in an FD that is saying something interesting about the relation.

# Finding functional dependencies

Many of these FDs don't seem to say much in the context of this enterprise but they are saying which attributes depend on other attributes, in the data. These FDs may not be real, or they may not be FDs that we want. At least we get the opportunity to review them and see if we want them to exist or not.

In order to determine which FDs we want and which we don't, we need to consider information from the enterprise domain itself. That will determine if the FDs are genuine and desirable, etc.

The FDs we have found seem to coalesce around 'employees' and 'projects'. This tells us something. It says that the relation might benefit from being decomposed into at least two different relations. One to describe employees and one to describe projects.