Relational databases

Relational databases

Last week we saw that much data, small and big, is stored in a database. We also saw that there are two kinds of databases: relational and non-relational. This week we focus on relational databases.

In a **relational database** your data is stored in a set of **tables**. You might sometimes hear tables referred to as "relations" (which is why relational databases are called "relational").

Typically each table stores data about **entities** of a particular kind, such as people, departments, books, and so on. So the database will have a table for each kind of entity about which you have data.

The table for a particular kind of entity has **columns** (also called **fields**), each of which stores a particular property of those entities. When designing the table you must specify what columns it has, and what type of data can be stored in that column (text, number, date, etc.).

Each **row** of a table (also called a **record**) contains data about an entity of that kind.

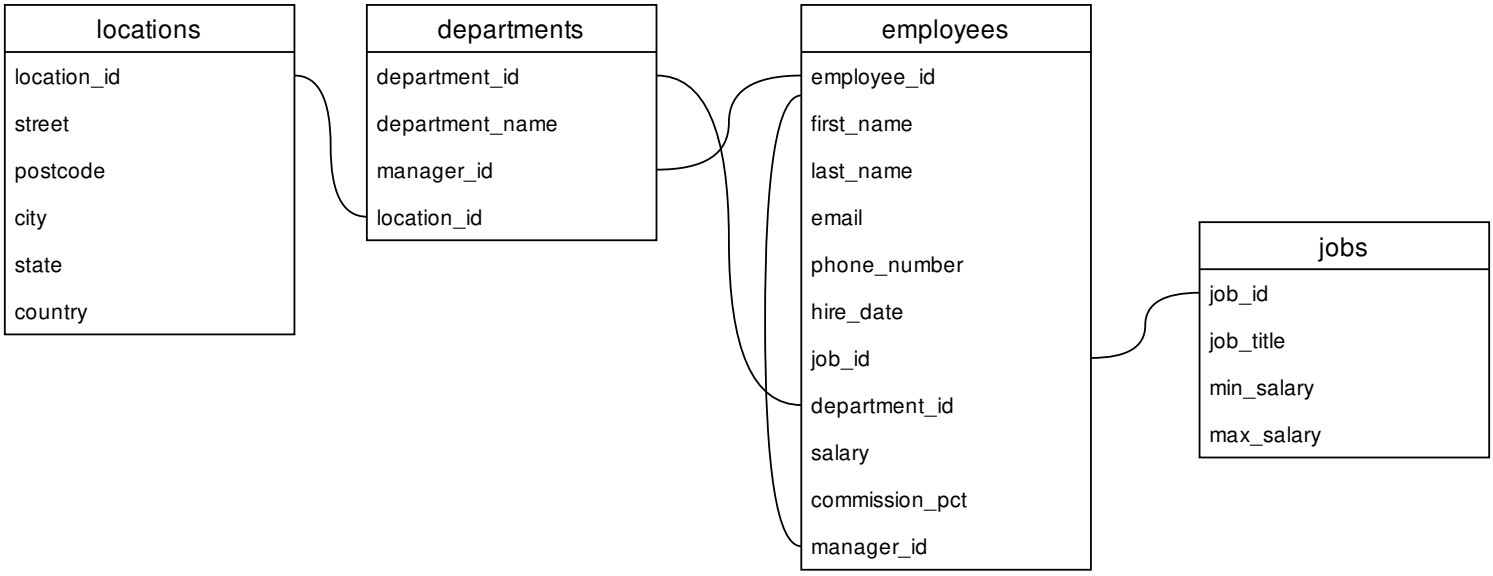
As you can see, a table is highly structured. The structure of a table - what the columns are, and what kind of data can go in each column - is called its **schema**.

You can also impose further constraints on the data in a table, such as requiring that a certain column contains **unique values**, or that a certain combination of columns contains **unique combinations of values**. This is a way to help stop the same piece of a data being adding twice to your database, such as the same person being added twice. Accidental duplications can lead to errors, and are best avoided.

Every row of a table has to conform to the tables's schema. Because of this, a relational database is sometimes said to be **schema on write**.

You will typically have many tables in your database. And these will typically be related. We will discuss these relations in the next slide.

Here is a diagram of an example database (the connecting lines show relations between tables - more about these in the next slide):


locations
location_id
street
postcode
city
state
country
departments
department_id
department_name
manager_id
location_id
employees
employee_id
first_name
last_name
email
phone_number
hire_date
job_id
department_id
salary
commission_pct
manager_id
jobs
job_id
job_title
min_salary
max_salary

Designing a relational database

When you keep your data in a relational database you first need to **design** your database - that is, to decide what tables you should have and what the schema of each should be (i.e. what columns it should have, what types of data can be stored in those columns, and what other constraints there should be on the data in the table, such as columns or combinations of columns being unique).

There are better and worse ways to do this. In this slide we'll go through the general principles that can help you to design your database well. In general, a well-designed relational database is one in which its tables are **normalised**. We'll see what that means, and how these principles help achieve normalisation.

Tables

What tables should you have? In general, for each kind of entity that you have data about you should have a table in which to store data about entities of that kind. So, if you have data about people and departments, then you should have a table in which to store data about people ("People" is a good name for it), and a table in which to store data about departments ("Departments" is a good name for it).

Columns

What columns should a table have? Suppose a table is storing data about people. In general, you should have a column for each property that a person has, that you want to store as data. A person has a first name, and a last name - here are two properties that a person has. If you want to store data about people's first and last names then you should have a column for each of these. "FirstName" and "LastName" are good names for those columns. So your table might look like this:

Atomic values

The columns in a table should contain **atomic values** - single values, with no structure. To see what this means, consider a couple of examples.

First, suppose we want to store a person's address, so add an Address column:

The values in the address column have **structure** - each address has meaningful parts, such as the street, suburb, state, etc. If we store all of these parts in one column it can be hard to impose constraints on those individual parts when storing the data, and hard to use them when retrieving data later. It's easy enough to search for a particular word - to search, for example, for addresses that contain the word "Hay". But what if you wanted to only find addresses whose *town* is Hay? Then just searching for the word "Hay" might not work - you might find addresses whose street name is "Hay". And what if you wanted to produce a list of towns? That would be difficult, because the town name might appear in very different locations within the address. Or what if you wanted to sort by town? That would also be difficult.

What we should do instead is break the address into its meaningful parts and have a column for each:

Second, suppose we want to store a person's phone number, or phone numbers, if she has more than one. We might try adding a single column for these:

But here we are doing the same thing as above - each number has two meaningful parts (number and type), and these are better separated. And we are doing something else as well - storing multiple items of the same kind in the one cell.

One way to do better is to have columns called "WorkPhone", "HomePhone", "MobilePhone", and store the phone numbers for each person in their appropriate column. What about other types of phone number? We would have to add a column for each of them too. How do allow that someone might have more than one phone number of the same type, such as two mobile numbers, or two work numbers? To avoid having multiple numbers in one cell we would have to add more columns: "MobilePhone2", "MobilePhone3", "WorkPhone2", "WorkPhone3", and so on. To allow for all of the possibilities we would end up with a lot of columns, many of which would contain mostly empty cells.

Another way, and probably better, is to separate the "Phone" column into just two columns, "PhoneType" and "PhoneNumber", and put each phone number in its own row. So our table would look like this:

In this way we can avoid all those extra columns with mostly empty cells - we only have a row when there is a phone number, in which case it will have both a type and a number.

So now we have atomic values in our columns.

No redundancy

But now our table suffers from a different kind of problem - it has **redundancy.** Notice that we are repeating a person's name in multiple rows. This creates a few problems:

- It increases the amount of data that we are storing and working with in the database
- It can create problems when **inserting data**. Suppose that Peter Simons gets another phone number, a home number. To add this to the database we need to add a new row, and enter his first and last names again, even though we already have that data in the database - this is extra work that is not needed, and if we are not careful then we might end up entering his names in a slightly different way, giving rise to inconsistent data (this is called an **insertion anomaly**).
- It can create problems when **updating data**. Suppose we discover that "Simons" should be spelled "Symons". We have to fix this error in multiple rows, which means extra work and time, and if we are not careful then we might end up with inconsistent data (this is called an **update anomaly**).
- It can create problems when **deleting data**. Suppose Peter Simons no longer has a mobile phone - he just has a work phone. To reflect this fact we can just delete that row in the table. Now suppose he no longer has a work phone either. If we reflect that fact in the same way, by deleting the corresponding row in the table, then notice what happens: we lose *all* data about Peter Simons, including his names - there is no longer any row in the table for him. In this case we have to not delete the row, but just make his phone columns blank. If we are not careful, we can end up losing data (this is called a **deletion anomaly**).

So what should we do? The problem is that we are mixing together two kinds of entities in the one table - people, and phone numbers. What we should have, instead, is a table for each.

A table for people:

And a table for phone numbers:

Now we have no redundancy, and we can avoid the problems listed above.

Linking and keys

But how do we know which phone numbers belong to which people? By **linking** the tables.

Here's a good way to do it.

First, we add a new column to the people table, called "PersonId", which is an automatically-generated number that is unique to a row. This is called a **primary key** for the table. It serves as a way of uniquely identifying a row in the table.

Next, we add a new column to the phone numbers table, also called "PersonId", which contains the id number of the person whose phone number it is. This is called a **foreign key** for the table. It serves as way of connecting a row in the phone numbers table with a row in the people table.

Now we have recorded which phone numbers belong to which people.

Note that the values in this foreign key column do not need to be unique - they are not serving to uniquely identify a row in the phone numbers table, but rather to identify a row in the people table.

Primary keys

In the above we added a foreign key column to the people table so that we could link phone numbers to people. But even without the need for linking we still need the people table to have a primary key. In fact, every table in our database should have a primary key - a column, or combination of columns, that uniquely identifies the rows in the table.

Why? Return to when our people table just had two columns - FirstName and LastName. Suppose that we have data about two people with the same first and last name. Suppose, for example, that our table looks like this:

This means we have two rows with exactly the same values in each column. This is a problem.

Why? Suppose we discover that one of their names is actually June, not Jane. We need to tell our database management system to update the name from Jane to June. But how do we tell it which of those two rows to update? We have no way of doing so. There is a similar problem if we want to delete one of those two people from the database - how do we specify which one to delete? That's why we need there to be a column, or combination of columns, that uniquely identifies the rows (i.e. no two rows can have the same value in that column or columns).

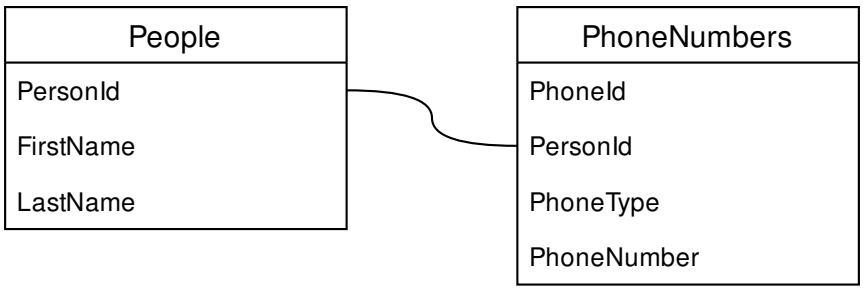
Does our phone numbers table have a primary key? It depends on whether we want to allow a person to have more than one phone number of the same type (e.g. two mobile numbers). If don't want to allow that then yes, we have a primary key: the combination of columns PersonId and PhoneType is a primary key - that combination of values uniquely identifies a row in the table (no two rows can have the same combination of values for those two columns).

What if we *do* want to allow multiple phone numbers of the same type? Perhaps the combination of *three* columns PersonId, PhoneType, and PhoneNumber would then be a primary key. But now it's getting complicated, and it's often better to do the same thing that we did with the people table - introduce a new column which contains an automatically-generated unique number for each row, to serve as the table's primary key:

The final result

So here are our tables and their data:

And here is a diagram of the schema of our database (i.e. the design of the tables):


People
PersonId
FirstName
LastName
PhoneNumbers
PhoneId
PersonId
PhoneType
PhoneNumber

Another example

Let's consider another example.

Suppose we want to store data about books and their authors. Let's design a relational database to do so.

We have data about books - their titles and year of publication, for example. So we should have a table for them - call it "Books". Let's suppose we have the following data:

Note that we have added a column called "BookId" - it contains an automatically generated unique number, to serve as the table's primary key.

We have data about people - their names, for example. So we should have a table for them too - call it "People". Let's suppose we have the following data:

Note again that we have added a column called "PersonId" - it contains an automatically generated unique number, to serve as the table's primary key.

We also have data about who wrote which book. Suppose that Jane Johnson wrote "DIY Candles" and Peter Simons wrote "How to Bake". How should we store that data?

We could think of who wrote a book as being a property of the book, and add a column called "AuthorId" to the Books table, in which we store the id number of the person who wrote it (thus making this column a foreign key):

That would be okay for the data that we currently have, but not for data that we might get in the future. Books can have multiple authors, and this design does not allow for that.

Alternatively, we could think of which book she wrote as being a property of a person, and add a

column called "BookId" to the People table, in which we store the id number of the book she wrote (thus making this column a foreign key):

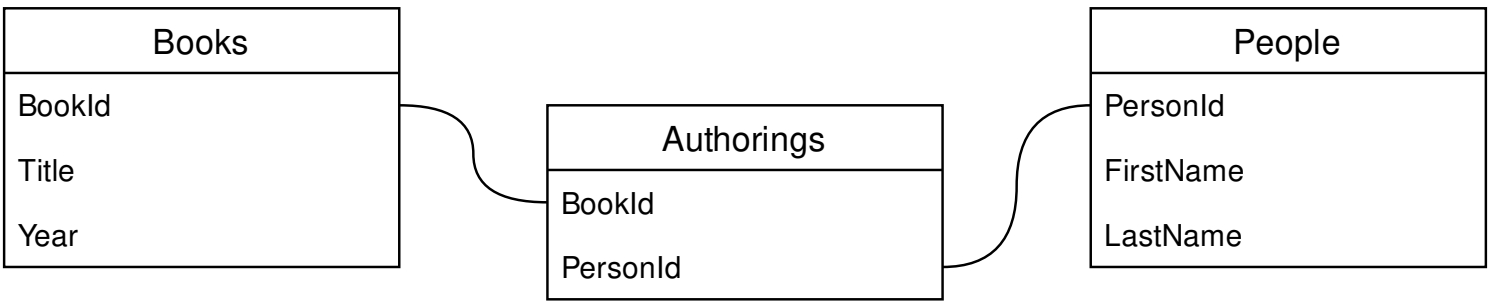
Again, that would be okay for the data that we currently have, but not for data that we might get in the future. A person might write multiple books, and this design does not allow for that.

What we should do is think of *authorings* as entities in their own right, and put data about them in their own table - let's call it "Authorings". An authoring is the writing of a book by someone. It has two properties: the person who did the writing, and the book that she wrote. So our authorings table should have a column in which to store the id number of the person who did the writing - call it "AuthorId" - and one in which to store the id number of the book that she wrote - call it "BookId" (each column is a foreign key). Then the table would look like this:

This design can accommodate a book having multiple authors - just add a row to this table for each author (with the appropriate book id). And it can accommodate a person writing multiple books - just add a row to the table for each book (with the appropriate person id).

What about a primary key? We could add another column, call it "AuthoringId", which contains an automatically generated unique number, to serve as primary key. But we don't need to - the combination of the two existing columns can serve as primary key, because each combination of values in those two columns is unique (we will never have the same person id and book id combination in more than once).

Here is a diagram of the schema of our database (i.e. the design of the tables):



Extending the database

As we get new kinds of data we can extend the database to accommodate it.

The key question to ask each time is: what kind of thing is this piece of data a property of - books, people, authorings, or some new kind of thing? If the answer is books, then we should add a new column to the books table in which to store the data. Similarly for people and authorings. If the answer is a new kind of thing, then we should create a new table for that kind of thing and add a column for the data, and link this new table to existing tables as required.

Let's see some examples.

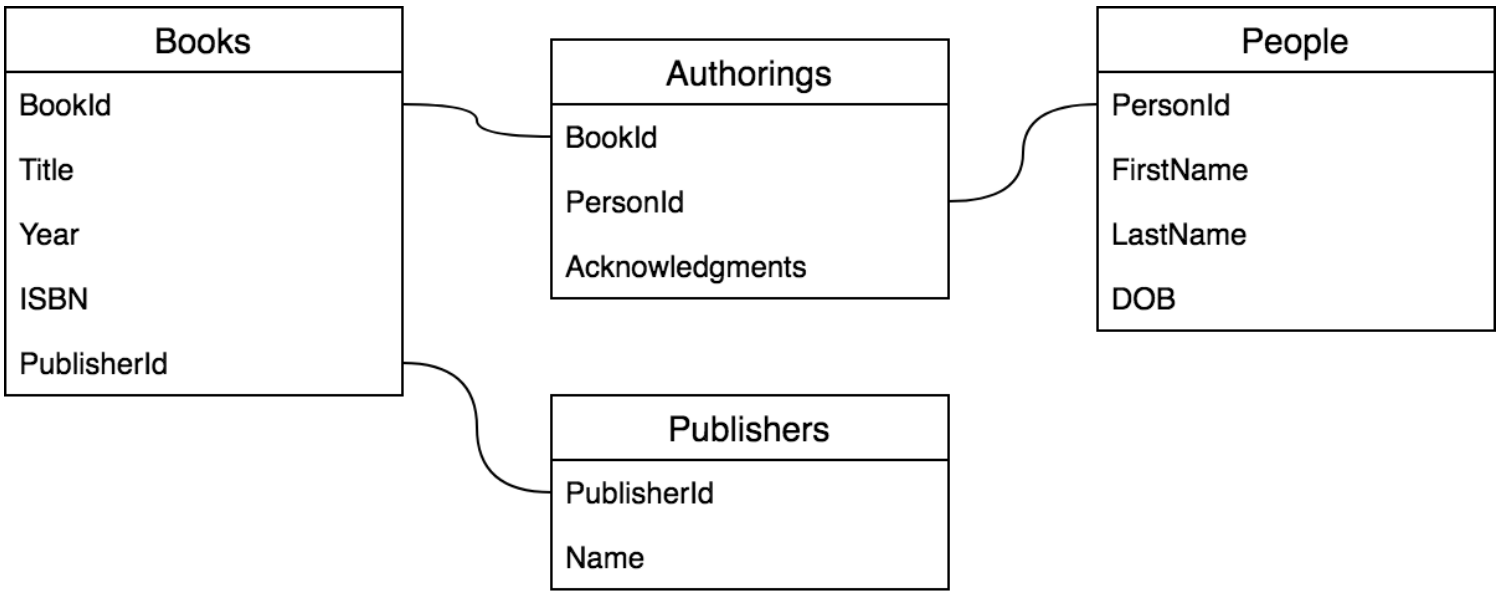
Suppose we want to start storing data about the ISBNs of books. Is this piece of data a property of books, people, authorings, or some new kind of thing? It's a property of books. So we should add a column for this to the books table - call it "ISBN".

Suppose we want to start storing data about the date of birth of book authors. Is this piece of data a property of books, people, authorings, or some new kind of thing? It's a property of people. So we should add a column for this to the people table - call it "DOB".

Suppose we want to start storing data about an author's acknowledgments. Is this piece of data a property of books, people, authorings, or some new kind of thing? This one is a bit trickier. You might be tempted to think this is a property of people. But an author's acknowledgments might vary from book to book, so it's actually a property of *authorings*. So we should add a column for this to the authorings table - call it "Acknowledgments".

Suppose we want to start storing the name of a book's publishing company. Is this piece of data a property of books, people, authorings, or some new kind of thing? Again, this one is a bit trickier. You might be tempted to think this is a property of books. But keep in mind that many books might have the same publishing company, and if we have a column in the books table for the name of the publishing company then we might end up with redundancy. It's better to think of this as a property of a new kind of thing, publishers. So we should create a table for publishers, called it "Publishers", which has a column called "PublisherId", containing an automatically generated unique number to serve as primary key, and a column called "Name", to store the name of the publisher, and in the books table we should have a column called "PublisherId" in which we store the id number of the company that published the book (thus making it a foreign key).

After making these four additions the schema of our database will look like this:


Books
BookId
Title
Year
ISBN
PublisherId
Authorings
BookId
PersonId
Acknowledgments
People
PersonId
FirstName
LastName
DOB
Publishers
PublisherId
Name

Normalisation

If you follow the design principles we have just described then you will end up with what's called a **normalised database**. A normalised database is one in which every table is normalised, and a table is normalised when:

- Each of its columns contains atomic values, and
- It has no redundancy

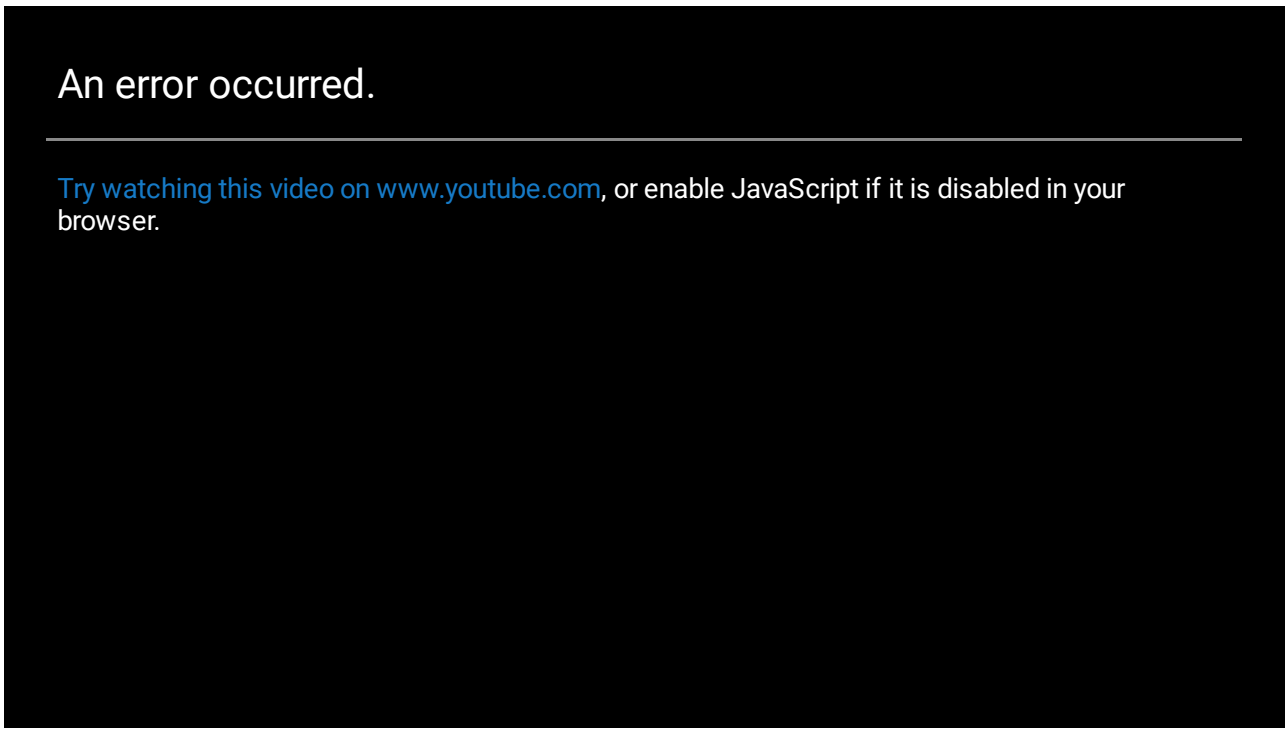
The actual definitions are more complicated than this. There are a series of so-called **normal forms** for a table:

- First normal form
- Second normal form
- Third normal form
- Boyce-Codd normal form
- Fourth normal form
- Fifth normal form

A table is in first normal form when each of its columns contains atomic values. It is in second and third normal form when it contains no redundancy (the actual definitions are quite technical, but this what they amount to). When it is in these three normal forms it is conventional to say that it is normalised, without worrying about the other normal forms, and we will follow that convention. So, for the purposes of this course, we can say that a table is normalised when:

- Each of its columns contains atomic values, and
- It has no redundancy

Normalisation (12:44)



The drawbacks of normalisation

As we have seen, there are advantages to having the tables in a database normalised (e.g. it minimises redundancy, which is a good thing because redundancy can cause problems when adding, updating, and deleting data.)

But there are disadvantages too.

For one, it can make it very difficult to write the SQL queries that are needed to extract data from the database (you will learn about these in detail next week, so don't worry if you don't fully understand this paragraph). Normalising the tables in a database can quickly lead to having many tables, and having them stand in complex relations to each other. When writing an SQL query to extract data you have to include the right tables, and connect them in the right way, and this can be difficult when there are many tables and the relations between them are complex. It's easy to make mistakes.

Even if you get the queries right, another disadvantage is that those queries can be slow to execute. Complicated queries can be difficult for the query engine of an RDBMS to execute efficiently, sometimes requiring the creation of temporary tables (which is slow). This second problem is exacerbated if your database contains a lot of data, as is often the case when organisations start dealing with big data.

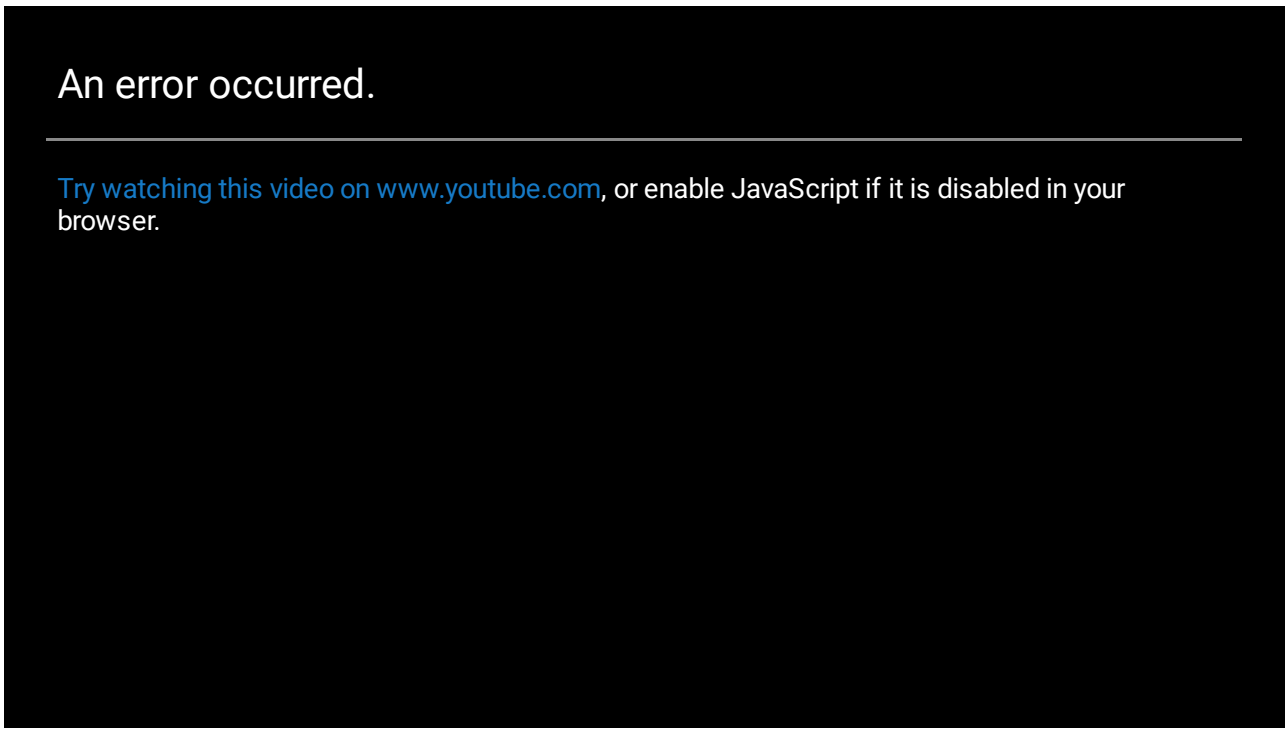
Because of these disadvantages, it can often make sense to **denormalise** a database, by allowing some degree of redundancy in the data. This can simplify and speed up the queries.

But what about the disadvantages of having redundancy? Redundancy is only really a problem when we are adding, modifying, and deleting data. If that doesn't happen very often then the advantages of redundancy can outweigh its disadvantages.

With this in mind, organisations that deal with large volumes of data often have at least *two* databases. One is used to manage the adding, updating and deleting of data, but not used for any large amounts of analysis. This is called an **operational database**. It is best kept normalised. The other keeps a copy of the data from the former database, but is only updated at certain times (e.g. once a day, at night). This one is used for analysing the data. Since the data is not added to, updated or deleted very often, it need not be so normalised. And since it is used for the analysis of the data, which can require a lot of querying, it is actually better for it not to be normalised. This is called a **data warehouse**.

Designing a data warehouse well is a bit different from designing an operational database well. You'll learn about that next.

Denormalisation (6:38)



Data warehouses

How does an organisation trade off the pros and cons of normalising a database?

A common strategy is to have two databases: a normalised one, which is used to add, update, and delete data as required (this is called an **operational database**), and a denormalised one, specially designed to keep a copy of the data in the operational database but in a way that makes it easy and fast to analyse to answer organisational questions (this is called a **data warehouse**).

The operational database is optimised for **Online Transactional Processing** (**OLTP**) - processing data in real-time. Data analysis can still be done but performance might be an issue.

The data warehouse is optimised for **Online Analytical Processing** (**OLAP**). Because the data warehouse is denormalised it contains redundant data. But that makes querying the data easier and faster.

Data in the operational database is copied to the data warehouse on a regular basis (e.g. every night). Data from many operational databases can be included, and so can data from other sources, such as spreadsheets.

This process is often called **Extract, Transform and Load** (**ETL**). Extraction is the process of retrieving data from the operational data sources. Transformation is the next stage, where data is transformed into a suitable form (often the data needs to be "cleaned" before it can be stored in the data warehouse). Loading is the last stage, where data is uploaded and stored in the data warehouse. At the end of the process, a data warehouse should have all of the data required for proper analysis.

Designing a data warehouse - Star schemas

Because a data warehouse in not fully normalised, their principles of design are slightly different from those of an operational database.

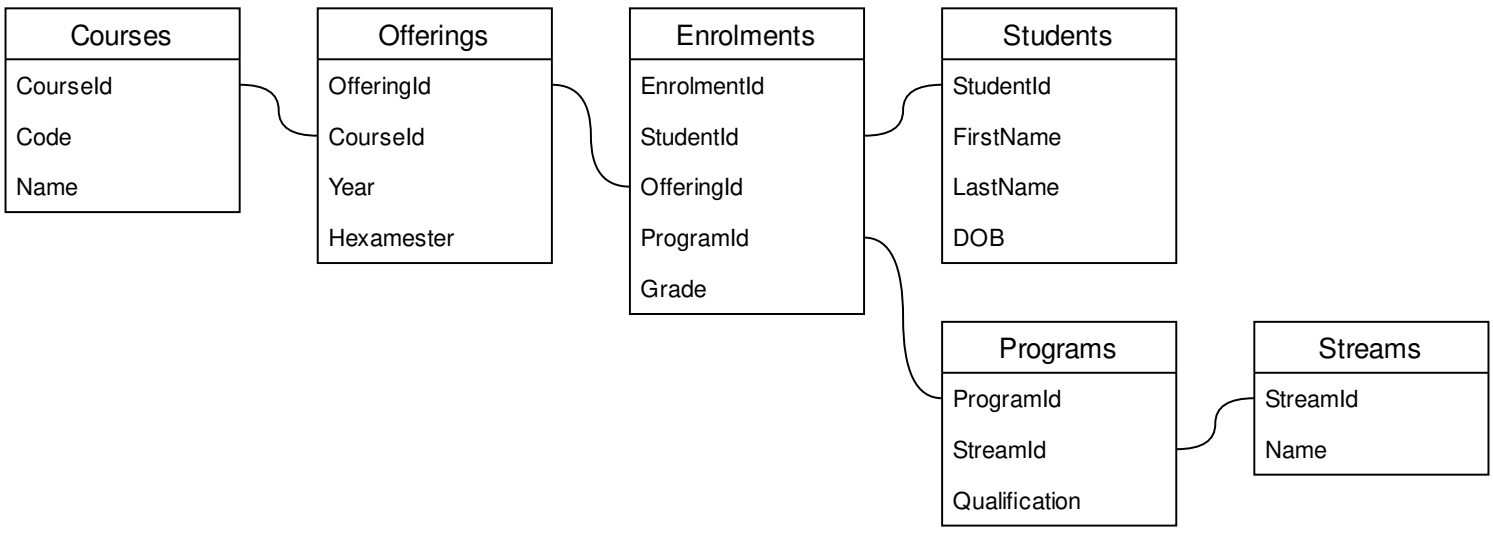
In general it's a good idea for a data warehouse to have what's called a **star schema**, so-called because it looks like a collection of stars, one or more. (You'll see below why it looks like a collection of stars).

In a star schema there are **fact tables** (which are at the centre of the stars) each of which is connected to some **dimension tables** (which are at the points of the stars).

Let's look at an example.

An operational database

Here is the schema of a operational database containing data about course enrolments:



The database stores data about student enrolments in course offerings. An enrolment consists of a student enrolling in a course offering as part of a program.

Enrolment data is stored in a table called "Enrolments". The table has columns called "EnrolmentId", which contains an auto-generated unique id number, "StudentId", which contains the id number of the student, "OfferingId", which contains the id number of the course offering, "ProgramId", which contains the id number of the program, and "Grade", which contains the grade the student gets in the course.

Student data is stored in a table called "Students". The table has columns called "StudentId", which contains an auto-generated unique id number, "FirstName", "LastName", and "DOB", which should be self-explanatory.

Course data is stored in a table called "Courses". The table has columns called "CourseId", which contains an auto-generated unique id number, "Code", which contains the code of the course (e.g. ZZEN9313), and "Name", which contains the name of the course (e.g. "Big Data Management").

Offering data is stored in a table called "Offerings". The table has columns called "OfferingId", which contains an auto-generated unique id number, "CourseId", which contains the id number of the course that is being offered, "Year", and "Hexamester", which should be self-explanatory.

Program data is stored in a table called "Programs". The table has columns called "ProgramId", which contains an auto-generated unique id number, "StreamId", which contains the id number of the stream that the program belongs to, and "Qualification", which contains the level of qualification of the program (e.g. "Graduate Certificate", "Graduate Diploma", "Masters", etc.).

Stream data is stored in a table called "Streams". The table has columns called "StreamId", which contains an auto-generated unique id number, and "Name", which contains the name of the stream (e.g. "Analytics", "Data Science", etc.).

Because it has been normalised the database has quite a few tables with quite a few links between them, which can make querying the data difficult and slow, especially when there is a large volume of it.

An example star

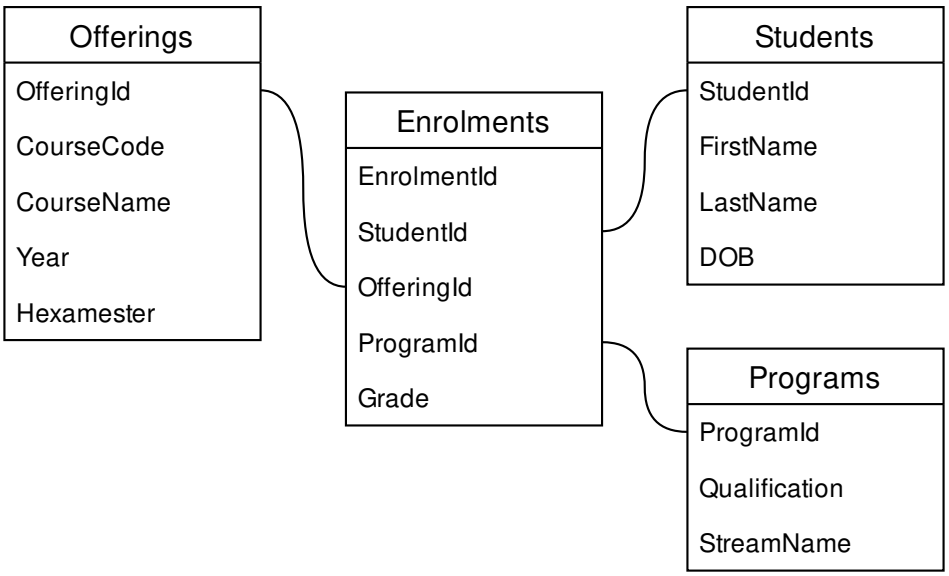
Let's now see how to design a star schema for this database. We'll look at how to design one of the stars in the star schema (later we'll look at a second one).

When you design a star in a star schema, you focus on a particular set of facts about which you are interested. The purpose of the star is to make it easier and faster to query your data to analyse those facts.

One set of facts that are stored in the operational database above, and about which we are interested, are **facts about enrolments**. So let's design a star that focuses on those facts.

Since we are interested in facts about enrolments we start with the Enrolments table - this is our fact table. Then we surround this fact table with tables that contain whatever further data we have about enrolments and are interested in - these are our dimension tables. We link each dimension table to the fact table using the appropriate column.

This is what we end up with:



Please note: Progarms.StreamID, Offerings.CourseId is probably missing from this diagram, and should be in the *Programs* dimension table.

Notice that it looks like a star, hence the name "star schema".

The fact table here, the Enrolments table, hasn't changed - it has the same columns with the same names as in the operational database.

Nor has one of the dimension tables - the Students table. The role of the Students table in this star is to contain all of the non-enrolment data about students in which we are interested. Since this is all contained in the Students table already, it can serve as the dimension table just as it is.

But the other two dimension tables have changed - Offerings and Programs.

Offerings. In our star we have merged the data from the Offerings table and the Courses table into a single table, keeping the name "Offerings". We can't keep them separate, and link them, because then would no longer have a star - we would have a second level of linking. The whole point of a star schema is to stop us from having a second level, thereby ensuring that our queries can be simple and fast.

This means that we now have redundancy in the table. For example, the same CourseCode will appear many times, as will the same CourseName. In the normalised operational database we pushed these columns out to another table, the Courses tables, where they appear just once. But in our denormalised star schema we have pulled them back in. We tolerate some data redundancy for the sake of structural simplicity.

Note that in the process of merging the tables we have renamed the columns "Code" and "Name" to "CourseCode" and "CourseName", just to make it clear what these columns contain.

Programs. Similarly, in our star we have merged the data from the Programs table and the Streams table into a single table, keeping the name "Programs". Again, this means that we now have redundancy in the table. For example, the same StreamName will appear many times. In the

normalised operational database we pushed this column out to another table, the Streams tables, where it appears just once. But in our denormalised data warehouse we have pulled it back in, tolerating redundancy for the sake of structural simplicity. Note that in the process of merging the tables we have renamed the column "Name" to "StreamName", again to make it clear what that column contains.

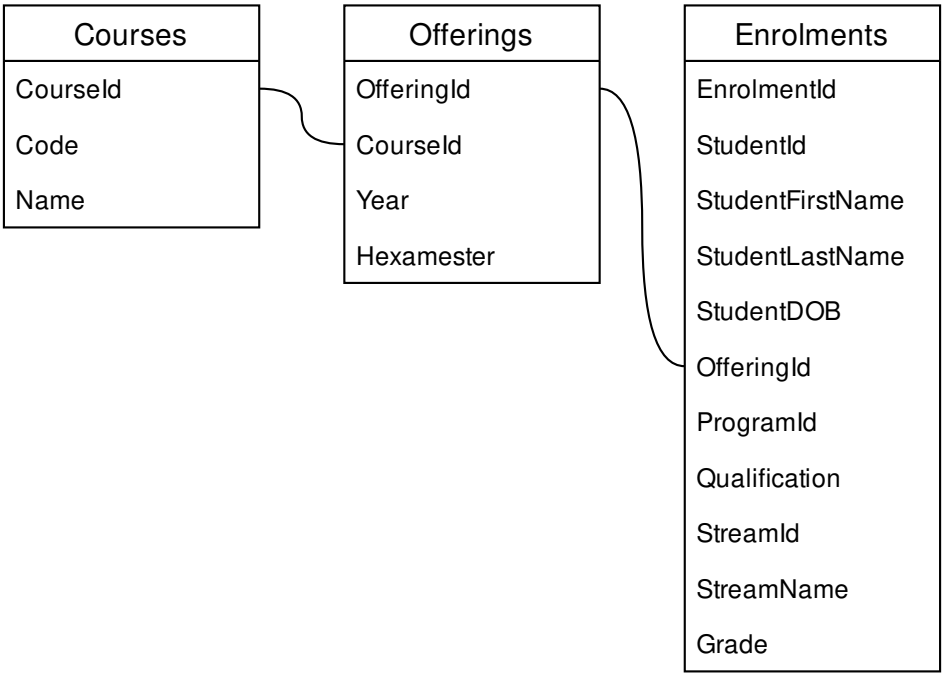
Another star

Now let's design another star in our star schema.

Another set of facts that are stored in the operational database, and about which we are interested, are **facts about offerings**. So let's design a star that focuses on these facts instead.

Since we are interested in facts about offerings we start with the Offerings table - this is our fact table. Then we surround this fact table with tables that contain whatever further data we have about offerings and are interested in - these are our dimension tables. We link each dimension table to the fact table using the appropriate column.

This is what we end up with:



Note: There is a strong case to say that StreamId would no longer be needed in the diagram above, since we've flattened it.

Since we only have two dimension tables in this case it looks less like a star, but it's still a star schema.

The fact table here, the Offerings table, hasn't changed - it has the same columns with the same names as in the operational database.

Nor has one of the dimension tables - the Courses table. The role of the Courses table in this star is to contain all of the non-offering data about courses in which we are interested. Since this is all contained in the Courses table already, it can serve as the dimension table just as it is.

But the other dimension table has changed - Enrolments. For this table we need to gather together all of the data about an offering's enrolments about which we are interested. In the operational database this is spread across four tables - Enrolments, Students, Programs, and Streams. In our star, we gather this all together and put into the one table, keeping the name "Enrolments". We can't keep this data in separate tables, and link them, because then we would no longer have a star. Again, the whole point of a star schema is to stop us from having that second level of linking (or higher), thereby ensuring that our queries can be simple and fast.

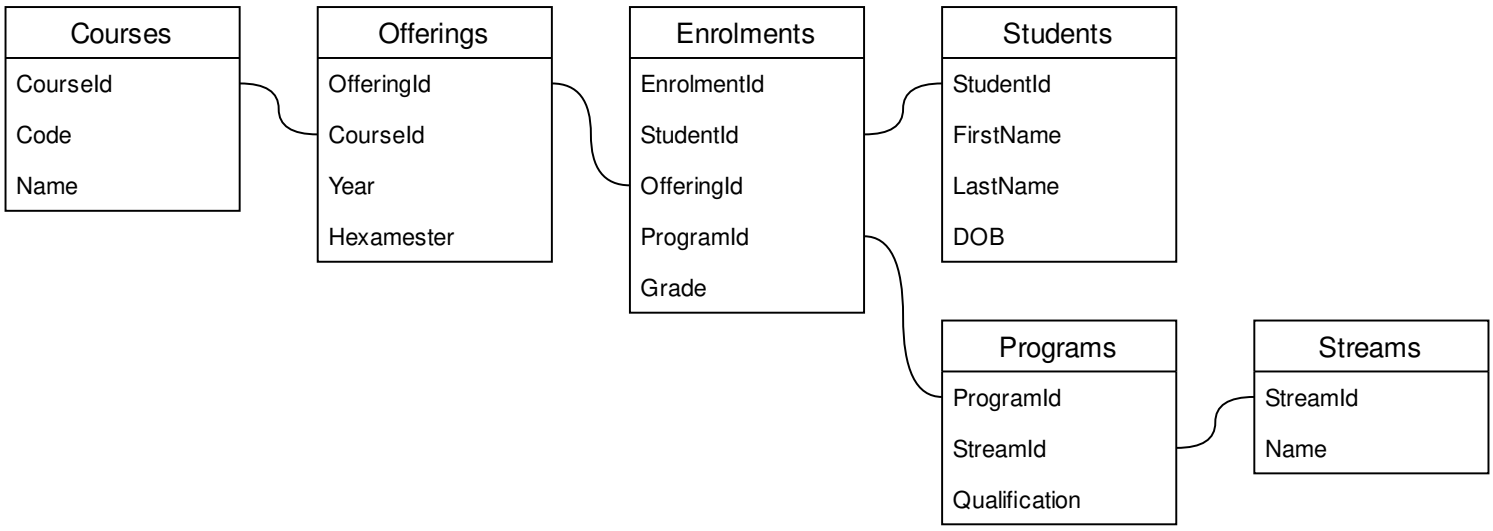
Again, this means that we now have redundancy in the Enrolments table, and quite a bit of it. For example, the same student first and last names will appear many times. In the normalised operational database we pushed these columns out to another table, the Students tables, where they appear just once. But in our denormalised star schema we have pulled them back in. We tolerate some data redundancy for the sake of structural simplicity.

Note that in the process of merging the tables we have renamed some columns: "StudentFirstName", "StudentLastName", "StudentDOB", and "StreamName", just to make it clear what these columns contain.

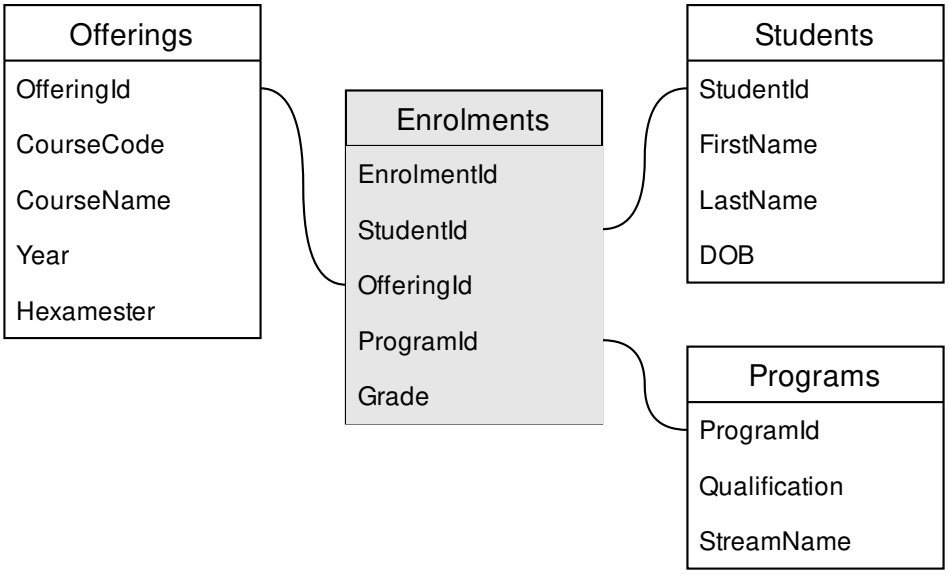
Comparison

To see how they compare, here are the schemas of the operational database and each of our two stars. In the case of the stars, the fact table has been shaded:

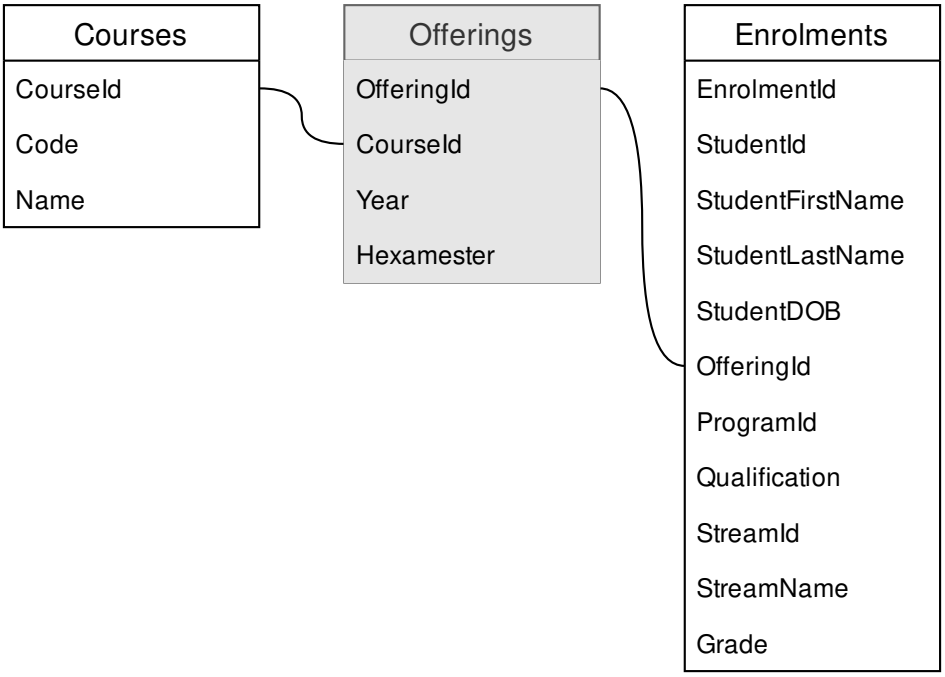
Operational database:



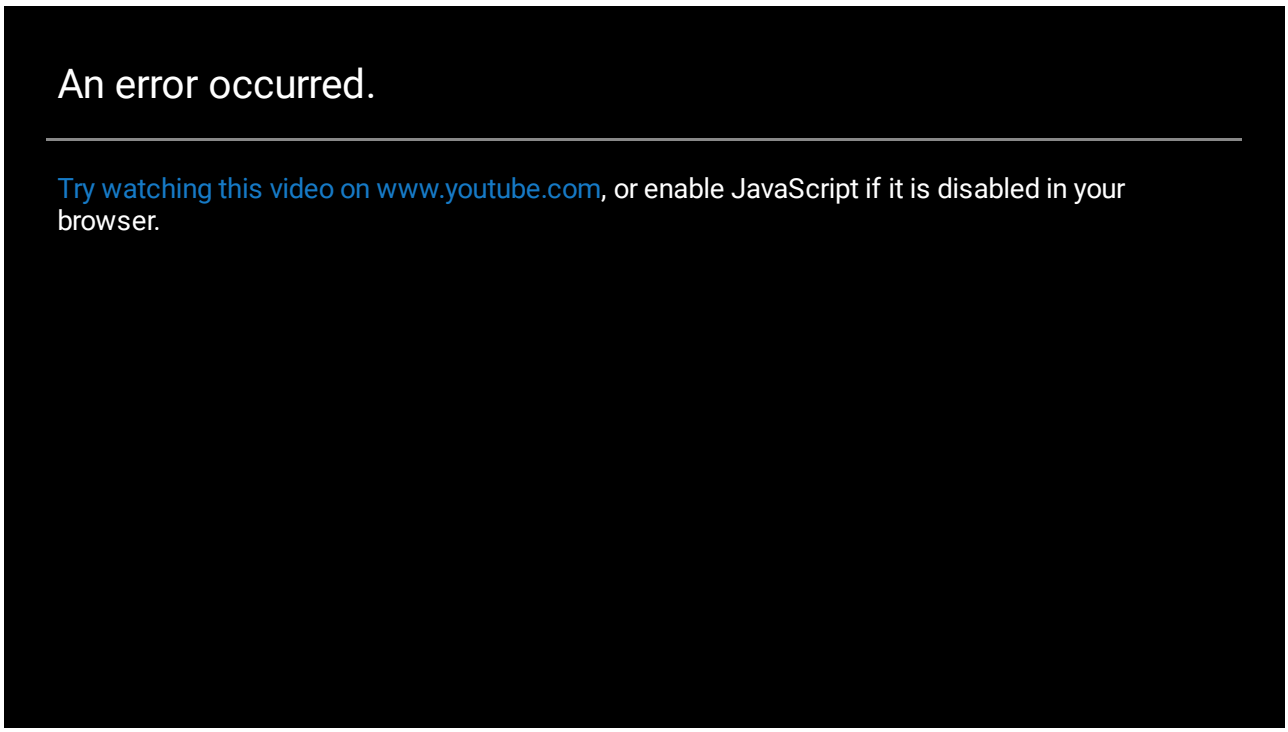
Star focusing on facts about enrolments:



Star focusing on facts about offerings:



Star schemas (6:49)



Further resources

If you'd like to learn more about the topics introduced this week you could look at **Elmasri and Navathe**, "Fundamentals of Database Systems". This book is on the course's reading list, which you can find in Moodle.

For details about normalisation, including the technical definitions of the normal forms, look at:

- Chapter 15, if you have Edition 6
- Chapter 14, if you have Edition 7

For details about data warehousing, look at:

- Chapter 28, if you have Edition 6
- Chapter 29, if you have Edition 7

The following video could also help you better understand Data warehouses.