Database Normalization: A Step-By-Step-Guide With Examples

Database normalisation is a concept that can be hard to understand.

What Is Database Normalization?

Database normalisation, or just normalisation as it's commonly called, is a process used for data modelling or database creation, where you organise your data and tables so it can be added and updated efficiently.

It can be done on any relational database where data is stored in tables that are linked to each other. This means that normalization in a DBMS (Database Management System) can be done in Oracle, Microsoft SQL Server, MySQL, PostgreSQL and any other type of database.

To perform the normalization process, you start with a rough idea of the data you want to store, and apply certain rules to it in order to get it to a more efficient form.

Why Normalize a Database?

So why would anyone want to normalize their database?

Why do we want to go through this manual process of rearranging the data?

There are a few reasons we would want to go through this process:

- Make the database more **efficient**
- Prevent the same data from being stored in **more than one place** (called an "insert anomaly")
- Prevent updates being made to **some data but not others** (called an "update anomaly")

- Prevent data not being deleted when it is supposed to be, or from data being lost when it is not supposed to be (called a "delete anomaly")
- Ensure the data is **accurate**
- Reduce the **storage space** that a database takes up
- Ensure the **queries** on a database run as fast as possible

Normalization in a DBMS is done to achieve these points. Without normalization on a database, the data can be **slow, incorrect, and messy**.

Data Anomalies

Some of these points above relate to "anomalies".

An anomaly is where *there is an issue in the data that is not meant to be there*. This can happen if a database is not normalised.

Let's take a look at the different kinds of data anomalies that can occur and that can be prevented with a normalised database.

Our Example

We'll be using a **student database** as an example in this article, which records student, class, and teacher information.

Let's say our student database looks like this:

Student ID	Student Name	Fees Paid	Course Name	Class 1	Class 2	Class 3
1	John Smith	200	Economics	Economics 1	Biology 1	
2	Maria Griffin	500	Computer Science	Biology 1	Business Intro	Programming 2
3	Susan Johnson	400	Medicine	Biology 2		
4	Matt Long	850	Dentistry			

This table keeps track of a few pieces of information:

- The student names
- The fees a student has paid
- The classes a student is taking, if any

This is **not** a normalised table, and there are a few issues with this.

Insert Anomaly

An insert anomaly happens when we try to insert a record into this table without knowing all the data we need to know.

For example, if we wanted to add a new student but did not know their course name.

The new record would look like this:

Student ID	**Student Name**	**Fees Paid**	**Course Name**	**Class 1**	**Class 2**	**Class 3**
1	John Smith	200	Economics	Economics 1	Biology 1	
2	Maria Griffin	500	Computer Science	Biology 1	Business Intro	Programming 2
3	Susan Johnson	400	Medicine	Biology 2		
4	Matt Long	850	Dentistry			
5	**Jared Oldham**	**0**	**?**			

We would be adding incomplete data to our table, which can cause issues when trying to analyse this data.

Update Anomaly

An update anomaly happens when we want to update data, and we update some of the data but not other data.

For example, let's say the class Biology 1 was changed to "Intro to Biology". We would have to query all of the columns that could have this Class field and rename each one that was found.

Student ID	Student Name	Fees Paid	Course Name	Class 1	Class 2	Class 3
1	John Smith	200	Economics	Economics 1	**Intro to Biology**	
2	Maria Griffin	500	Computer Science	**Intro to Biology**	Business Intro	Programming 2
3	Susan Johnson	400	Medicine	Biology 2		
4	Matt Long	850	Dentistry			

There's a risk that we miss out on a value, which would cause issues.

Ideally, we would only update the value once, in one location.

Delete Anomaly

A delete anomaly occurs when we want to delete data from the table, but we end up deleting more than what we intended.

For example, let's say Susan Johnson quits and her record needs to be deleted from the system. We could delete her row:

Student ID	Student Name	Fees Paid	Course Name	Class 1	Class 2	Class 3
1	John Smith	200	Economics	Economics 1	Biology 1	
2	Maria Griffin	500	Computer Science	Biology 1	Business Intro	Programming 2

3	**Susan Johnson**	**400**	**Medicine**	**Biology 2**		
4	Matt Long	850	Dentistry			

But, if we delete this row, we lose the record of the Biology 2 class, because it's not stored anywhere else. The same can be said for the Medicine course.

We should be able to delete one type of data or one record without having impacts on other records we don't want to delete.

What Are The Normal Forms?

The process of normalization involves applying rules to a set of data. Each of these rules transforms the data to a certain structure, called a **normal form**.

There are **three main normal forms** that you should consider (Actually, there are six normal forms in total, but the first three are the most common).

Whenever the first rule is applied, the data is in "**first normal form**". Then, the second rule is applied and the data is in "**second normal form**". The third rule is then applied and the data is in "**third normal form**".

Fourth and fifth normal forms are then achieved from their specific rules.

What Is First Normal Form?

First normal form is the way that your data is represented after it has the first rule of normalization applied to it. Normalization in DBMS starts with the first rule being applied – you need to apply the first rule before applying any other rules.

Let's start with a sample database. In this case, we're going to use a student and teacher database at a school. We mentioned this earlier in the article when we spoke about anomalies, but here it is again.

Our Example Database

We have a set of data we want to capture in our database, and this is how it currently looks. It’s a single table called “student” with a lot of columns.

Student Name	**Fees Paid**	**Date of Birth**	**Address**	**Subject 1**	**Subject 2**	**Subject 3**	**Subject 4**	**Teacher Name**	**Teacher Address**	**Course Name**
John Smith	18-Jul-00	04-Aug-91	3 Main Street, North Boston 56125	Economics 1 (Business)	Biology 1 (Science)			James Peterson	44 March Way, Glebe 56100	Economics
Maria Griffin	14-May-01	10-Sep-92	16 Leeds Road, South Boston 56128	Biology 1 (Science)	Business Intro (Business)	Programming 2 (IT)		James Peterson	44 March Way, Glebe 56100	Computer Science
Susan Johnson	03-Feb-01	13-Jan-91	21 Arrow Street, South Boston 56128	Biology 2 (Science)				Sarah Francis		Medicine
Matt Long	29-Apr-02	25-Apr-92	14 Milk Lane,Â South Boston 56128					Shane Cobson	105 Mist Road, Faulkner 56410	Dentistry

Everything is in one table.

How can we normalise this?

We start with getting the data to First Normal Form.

To apply first normal form to a database, we look at each table, one by one, and ask ourselves the following questions of it:

1. **Does the combination of all columns make a unique row every single time?**
2. **What field can be used to uniquely identify the row?**

Let’s look at the first question.

Does the combination of all columns make a unique row every single time?

No. There could be the same combination of data, and it would represent a different row. There could be the same values for this row and it would be a separate row (even though it is rare).

The second question says:

What field can be used to uniquely identify the row?

Is this the student name? No, as there could be two students with the same name.

Address? No, this isn’t unique either.

Any other field?

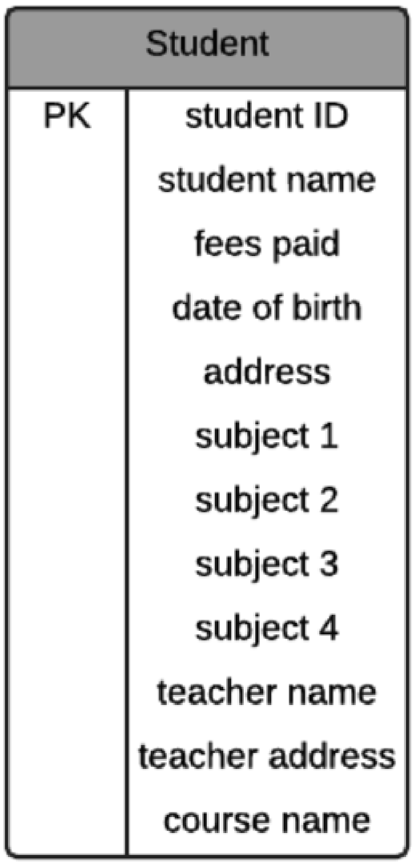
We don’t have a field that can uniquely identify the row.

If there is no unique field, we need to create a new field. This is called a primary key, and is a database term for a field that is unique to a single row.

When we create a new primary key, we can call it whatever we like, but it should be obvious and consistently named between tables. I prefer using the ID suffix, so I would call it **student ID.**

This is our new table:

Student (<u>student ID</u>, student name, fees paid, date of birth, address, subject 1, subject 2, subject 3, subject 4, teacher name, teacher address, course name)



Student	
PK	student ID
	student name
	fees paid
	date of birth
	address
	subject 1
	subject 2
	subject 3
	subject 4
	teacher name
	teacher address
	course name

The way I have written this is a common way of representing tables in text format. The table name is written, and all of the columns are shown in brackets, with the primary key underlined.

This data is now in first normal form.

This example is still in one table, but it's been made a little better by adding a unique value to it.

What Is Second Normal Form?

The rule of second normal form on a database can be described as:

1. Fulfil the requirements of first normal form
2. Each non-key attribute must be functionally dependent on the primary key

What does this even mean?

It means that the first normal form rules have been applied. It also means that **each field that is not the primary key is determined by that primary key**, so it is specific to that record. This is what "functional dependency" means.

Let's take a look at our table.

Student (<u>student ID</u>, student name, fees paid, date of birth, address, subject 1, subject 2, subject 3, subject 4, teacher name, teacher address, course name)

Are all of these columns dependent on and specific to the primary key?

The primary key is student ID, which represents the student. Let's look at each column:

- student name: Yes, this is dependent on the primary key. A different student ID means a different student name.
- fees paid: Yes, this is dependent on the primary key. Each fees paid value is for a single student.
- date of birth: Yes, it's specific to that student.
- address: Yes, it's specific to that student.
- subject 1: No, this column is not dependent on the student. More than one student can be enrolled in one subject.
- subject 2: As above, more than one subject is allowed.
- subject 3: No, same rule as subject 2.
- subject 4: No, same rule as subject 2
- teacher name: No, the teacher name is not dependent on the student.
- teacher address: No, the teacher address is not dependent on the student.
- course name: No, the course name is not dependent on the student.

We have a mix of Yes and No here. Some fields are dependent on the student ID, and others are not.

How can we resolve those we marked as No?

Let's take a look.

Subject

First, the subject 1 column. It is not dependent on the student, as more than one student can have a subject, and the subject isn't a part of the definition of a student.

So, we can move it to a new table:

Subject (subject name)

I've called it subject name because that's what the value represents. When we are writing queries on this table or looking at diagrams, it's clearer what subject name is instead of using subject.

Now, is this field unique? Not necessarily. Two subjects could have the same name and this would cause problems in our data.

So, what do we do? We add a primary key column, just like we did for student. I'll call this subject ID, to be consistent with the student ID.

Subject (<u>subject ID</u>, subject name)

This means we have a student table and a subject table. We can do this for all four of our subject columns in the student table, removing them from the student table so it looks like this:

Student (<u>student ID</u>, student name, fees paid, date of birth, address, teacher name, teacher address, course name)

But they are in separate tables. How do we link them together?

We'll cover that shortly. For now, let's keep going with our student table.

The next column we marked as No was the Teacher Name column. The teacher is separate to the student so should be captured separately. This means we should move it to its own table.

Teacher (teacher name)

We should also move the teacher address to this table, as it’s a property of the teacher. I’ll also rename teacher address to be just address.

Teacher (teacher name, address)

Just like with the subject table, the teacher name and address is not unique. Sure, in most cases it would be, but to avoid duplication we should add a primary key. Let’s call it teacher ID,

Teacher (teacher ID, teacher name, address)

Course

The last column we have to look at was the Course Name column. This indicates the course that the student is currently enrolled in.

While the course is related to the student (a student is enrolled in a course), the name of the course itself is not dependent on the student.

So, we should move it to a separate table. This is so any changes to courses can be made independently of students.

The course table would look like this:

Course (course name)

Let’s also add a primary key called course ID.

We now have our tables created from columns that were in the student table. Our database so far looks like this:

Student (student ID, student name, fees paid, date of birth, address)

Subject (subject ID, subject name)

Teacher (teacher ID, teacher name, address)

Course (course ID, course name)

Using the data from the original table, our data could look like this:

Student

student ID	student name	fees paid	date of birth	address
1	John Smith	18-Jul-00	04-Aug-91	3 Main Street, North Boston 56125
2	Maria Griffin	14-May-01	10-Sep-92	16 Leeds Road, South Boston 56128
3	Susan Johnson	03-Feb-01	13-Jan-91	21 Arrow Street, South Boston 56128
4	Matt Long	29-Apr-02	25-Apr-92	14 Milk Lane, South Boston 56128

Subject

subject ID	subject name
1	Economics 1 (Business)
2	Biology 1 (Science)
3	Business Intro (Business)
4	Programming 2 (IT)
5	Biology 2 (Science)

Teacher

teacher ID	teacher name	address
1	James Peterson	44 March Way, Glebe 56100
2	Sarah Francis	
3	Shane Cobson	105 Mist Road, Faulkner 56410

Course

course ID	course name
1	Computer Science
2	Dentistry
3	Economics
4	Medicine

How do we link these tables together? We still need to know which subjects a student is taking, which course they are in, and who their teachers are.

Foreign Keys in Tables

We have four separate tables, capturing different pieces of information. We need to capture that students are taking certain courses, have teachers, and subjects. But the data is in different tables.

How can we keep track of this?

We use a concept called a foreign key.

A foreign key is a column in one table that refers to the primary key in another table. Related: The Complete Guide to Database Keys.

It's used to link one record to another based on its unique identifier, without having to store the additional information about the linked record.

Here are our two tables so far:

Student (<u>student ID</u>, student name, fees paid, date of birth, address)

Subject (<u>subject ID</u>, subject name)

Teacher (<u>teacher ID</u>, teacher name, teacher address)

Course (<u>course ID</u>, course name)

To link the two tables using a foreign key, we need to put the primary key (the underlined column) from one table into the other table.

Let's start with a simple one: students taking courses. For our example scenario, a student can only be enrolled in one course at a time, and a course can have many students.

We need to either:

- Add the course ID from the course table into the student table
- Add the student ID from the student table into the course table

But which one is it?

In this situation, I ask myself a question to work out which way it goes:

Does a table1 have many table2s, or does a table2 have many table1s?

If it's the first, then table1 ID goes into table 2, and if it's the second then table2 ID goes into table1.

So, if we substitute table1 and table2 for course and student:

Does a course have many students, or does a student have many courses?

Based on our rules, the first statement is true: a course has many students.

This means that the course ID goes into the student table.

Student (<u>student ID</u>, course ID, student name, fees paid, date of birth, address)

Subject (<u>subject ID</u>, subject name)

Teacher (<u>teacher ID</u>, teacher name, teacher address)

Course (<u>course ID</u>, course name)

I've *italicised* it to indicate it is a foreign key – a value that links to a primary key in another table.

When we actually populate our tables, instead of having the course name in the student table, the course ID goes in the student table. The course name can then be linked using this ID.

student ID	course ID	student name	fees paid	date of birth	address
1	3	John Smith	200	4 Aug 1991	3 Main Street, North Boston 56125
2	1	Maria Griffin	500	10 Sep 1992	16 Leeds Road, South Boston 56128
3	4	Susan Johnson	400	13 Jan 1991	21 Arrow Street, South Boston 56128
4	2	Matt Long	850	25 Apr 1992	14 Milk Lane, South Boston 56128

This also means that the course name is stored in one place only, and can be added/removed/updated without impacting other tables.

Teacher

We've linked the student to the course. Now let's look at the teacher.

How are teachers related? Depending on the scenario, they could be related in one of a few ways:

- A student can have one teacher that teaches them all subjects
- A subject could have a teacher than teaches it
- A course could have a teacher that teaches all subjects in a course

In our scenario, a teacher is related to a course. We need to relate these two tables using a foreign key.

Does a teacher have many courses, or does a course have many teachers?

In our scenario, the first statement is true. So the teacher ID goes into the cou*Student (student ID, course ID, student name, fees paid, date of birth, address)*

Subject (subject ID, subject name)

Teacher (teacher ID, teacher name, teacher address)

Course (course ID, teacher ID, course name)

The table data would look like this:

Course

course ID	teacher ID	course name
1	1	Computer Science
2	3	Dentistry
3	1	Economics
4	2	Medicine

Teacher

teacher ID	teacher name	address
1	James Peterson	44 March Way, Glebe 56100
2	Sarah Francis	
3	Shane Cobson	105 Mist Road, Faulkner 56410

This allows us to change the teacher's information without impacting the courses or students.

Student and Subject

So we've linked the course, teacher, and student tables together so far.

What about the subject table?

Does a subject have many students, or does a student have many subjects?

The answer is both.

How is that possible?

A student can be enrolled in many subjects at a time, and a subject can have many students in it.

How can we represent that? We could try to put one table's ID in the other table:

student ID	course ID	subject ID	student name	fees paid	date of birth	address
1	3	1, 2	John Smith	200	4 Aug 1991	3 Main Street, North Boston 56125
2	1	2, 3, 2004	Maria Griffin	500	10 Sep 1992	16 Leeds Road, South Boston 56128
3	4	5	Susan Johnson	400	13 Jan 1991	21 Arrow Street, South Boston 56128
4	2		Matt Long	850	25 Apr 1992	14 Milk Lane, South Boston 56128

But if we do this, we're storing many pieces of information in one column, possibly separated by commas.

This makes it hard to maintain and is very prone to errors.

If we have this kind of relationship, one that goes both ways, it's called a **many to many relationship**. It means that many of one record is related to many of the other record.

Many to Many Relationships

A many to many relationship is common in databases. Some examples where it can happen are:

- Students and subjects
- Employees and companies (an employee can have many jobs at different companies, and a company has many employees)
- Actors and movies (an actor is in multiple movies, and a movie has multiple actors)

If we can't represent this relationship by putting a foreign key in each table, how can we represent it?

We use a joining table.

This is a table that is created purely for storing the relationships between the two tables.

It works like this. Here are our two tables:

Student (<u>student ID</u>, course ID, student name, fees paid, date of birth, address)

Subject (<u>subject ID</u>, subject name)

And here is our joining table:

Subject_Student (student ID, subject ID)

It has two columns. Student ID is a foreign key to the student table, and subject ID is a foreign key to the subject table.

Each record in the row would look like this:

student ID	subject ID
1	1
1	2
2	2

2	3
2	4
3	5

Each row represents a relationship between a student and a subject.

Student 1 is linked to subject 1.

Student 1 is linked to subject 2.

Student 2 is linked to subject 2.

And so on.

This has several advantages:

- It allows us to store many subjects for each student, and many students for each subject.
- It separates the data that describes the records (subject name, student name, address, etc.) from the relationship of the records (linking ID to ID).
- It allows us to add and remove relationships easily.
- It allows us to add more information about the relationship. We could add an enrolment date, for example, to this table, to capture when a student enrolled in a subject.

You might be wondering, how do we see the data if it's in multiple tables? How can we see the student name and the name of the subjects they are enrolled in?

Well, that's where the magic of SQL comes in. We use a SELECT query with JOINs to show the data we need. But that's outside the scope of this article – you can read the articles on my Oracle Database page to find out more about writing SQL.

One final thing I have seen added to these joining tables is a primary key of its own. An ID field that represents the record. This is an optional step – a primary key on a single new column works in a similar way to defining the primary key on the two ID columns. I'll leave it out in this example

So, our final table structure looks like this:

Student (student ID, course ID, student name, fees paid, date of birth, address)

Subject (subject ID, subject name)

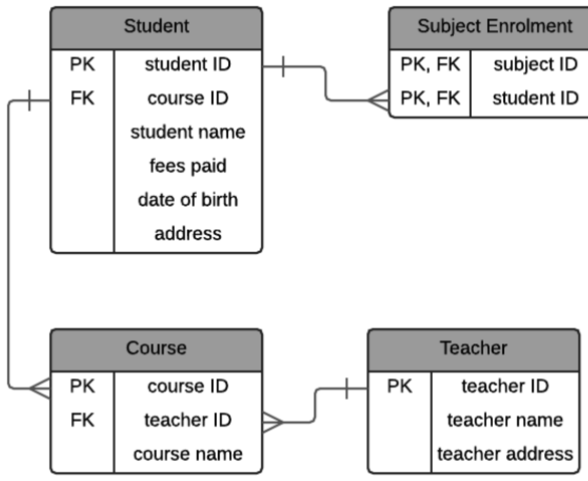
Subject Enrolment (student ID, subject ID)

Teacher (teacher ID, teacher name, teacher address)

Course (course ID, teacher ID, course name)

I've also underlined both columns in this table, as they represent the primary key. They can also represent a foreign key, which is why they are also italicised.

An ERD of these tables looks like this:



This database structure is in second normal form. We almost have a normalised database.

Now, let's take a look at third normal form.

What Is Third Normal Form?

Third normal form is the final stage of the most common normalization process. The rule for this is:

- Fulfils the requirements of second normal form
- Has no transitive functional dependency

What does this even mean? What is a transitive functional dependency?

It means that **every attribute that is not the primary key must depend on the primary key and the primary key only**.

For example:

- Column A determines column B
- Column B determines column C
- Therefore, column A determines C

This means that *column A determines column B which determines column C*. This is a transitive functional dependency, and it should be removed. Column C should be in a separate table.

We need to check if this is the case for any of our tables.

Student (student ID, course ID, student name, fees paid, date of birth, address)

Do any of the non-primary-key fields depend on something other than the primary key?

No, none of them do. However, if we look at the address, we can see something interesting:

Address
3 Main Street, North Boston 56125
16 Leeds Road, South Boston 56128

21 Arrow Street, South Boston 56128
14 Milk Lane, South Boston 56128

We can see that there is a relationship between the ZIP code and the city or suburb. This is common with addresses, and you may have noticed this if you have filled out any forms for addresses online recently.

How are they related? The ZIP code, or postal code, determines the city, state, and suburb.

In this case, 56128 is South Boston, and 56125 is North Boston. (I just made this up so this is probably inaccurate data).

This falls into the pattern we mentioned earlier: A determines B which determines C.

Student determines the address ZIP code which determines the suburb.

So, how can we improve this?

We can move the ZIP code to another table, along with everything it identifies, and link to it from the student table.

Our table could look like this:

Student (<u>student ID</u>, course ID, student name, fees paid, date of birth, street address, address code ID)

Address Code (<u>address code ID</u>, ZIP code, suburb, city, state)

I've created a new table called Address Code, and linked it to the student table. I created a new column for the address code ID, because the ZIP code may refer to more than one suburb. This way we can capture that fact, and it's in a separate table to make sure it's only stored in one place.

Let's take a look at the other tables:

Subject (subject ID, subject name)

Subject Enrolment (student ID, subject ID)

Both of these tables have no columns that aren't dependent on the primary key.

Teacher (teacher ID, teacher name, teacher address)

The teacher table also has the same issue as the student table when we look at the address. We can, and should use the same approach for storing address.

So our table would look like this:

Teacher (teacher ID, teacher name, street address, address code ID)

Address Code (address code ID, ZIP code, suburb, city, state)

It uses the same Address Code table as mentioned above. We aren't creating a new address code table.

Finally, the course table:

Course (course ID, teacher ID, course name)

This table is OK. The course name is dependent on the course ID.

So, what does our database look like now?

Student (student ID, course ID, student name, fees paid, date of birth, street address, address code ID)

Address Code (address code ID, ZIP code, suburb, city, state)

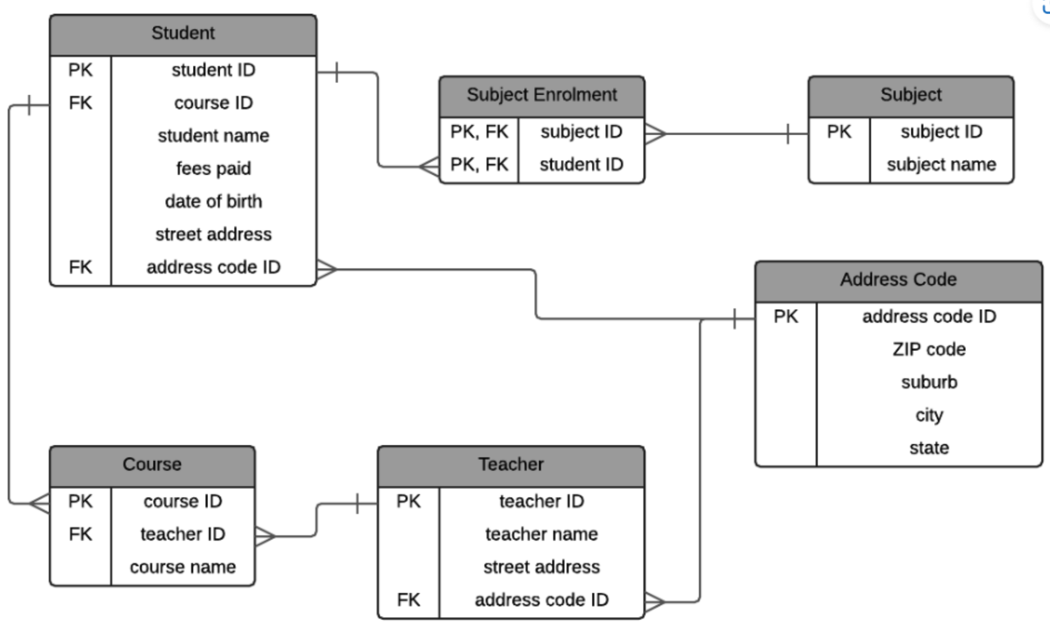
Subject (subject ID, subject name)

Subject Enrolment (student ID, subject ID)

Teacher (teacher ID, teacher name, street address, address code ID)

Course (<u>course ID</u>, teacher ID, course name)

So, that's how third normal form could look if we had this example.



Stopping at Third Normal Form

For most database normalisation exercises, stopping after achieving Third Normal Form is enough.

It satisfies a good relationship rules and will greatly improve your data structure from having no normalisation at all.

There are a couple of steps after third normal form that are optional. I'll explain them here so you can learn what they are.

Fourth Normal Form and Beyond

Fourth normal form is the next step after third normal form.

What does it mean?

It needs to satisfy two conditions:

- Meet the criteria of third normal form.
- There are no non-trivial multivalued dependencies other than a candidate key.

So, what does this mean?

A multivalued dependency is probably better explained with an example, which I'll show you shortly. It means that there are other attributes in the table that are not dependent on the primary key, and can be moved to another table.

Our database looks like this:

Student (<u>student ID</u>, course ID, student name, fees paid, date of birth, street address, address code ID)

Address Code (<u>address code ID</u>, ZIP code, suburb, city, state)

Subject (<u>subject ID</u>, subject name)

Subject Enrolment (<u>student ID, subject ID</u>)

Teacher (<u>teacher ID</u>, teacher name, street address, address code ID)

Course (<u>course ID</u>, teacher ID, course name)

This meets the third normal form rules.

However, let's take a look at the address fields: street address and address code.

- Both the student and teacher table have these
- What if a student moves addresses? Do we update the address in this field? If we do, then we lose the old address.
- If an address is updated, is it because they moved? Or is it because there was an error in the old address?
- What if two students have the same street address. Are they actually at the same address? What if we update one and not the other?
- What if a teacher and a student are at the same address?
- What if we want to capture a student or a teacher having multiple addresses (for example, postal and residential)?

There are a lot of “what if” questions here. There is a way we can resolve them and improve the quality of the data.

This is a multivalued dependency.

We can solve this by **moving the address to a separate table**.

The address can then be linked to the teacher and student tables.

Let’s start with the address table.

Address (address ID, street address, address code ID)

In this table, we have a primary key of address ID, and we have stored the street address here. The address code table stays the same.

We need to link this to the student and teacher tables. How do we do this?

Do we also want to capture the fact that a student or teacher can have multiple addresses? It may be a good idea to future proof the design. It’s something you would want to confirm in your organisation.

For this example, we will design it so there can be multiple addresses for a single student.

Our tables could look like this:

Student (student ID, course ID, student name, fees paid, date of birth)

Address (address ID, street address, address code ID)

Address Code (address code ID, ZIP code, suburb, city, state)

Student Address (address ID, student ID)

Subject (subject ID, subject name)

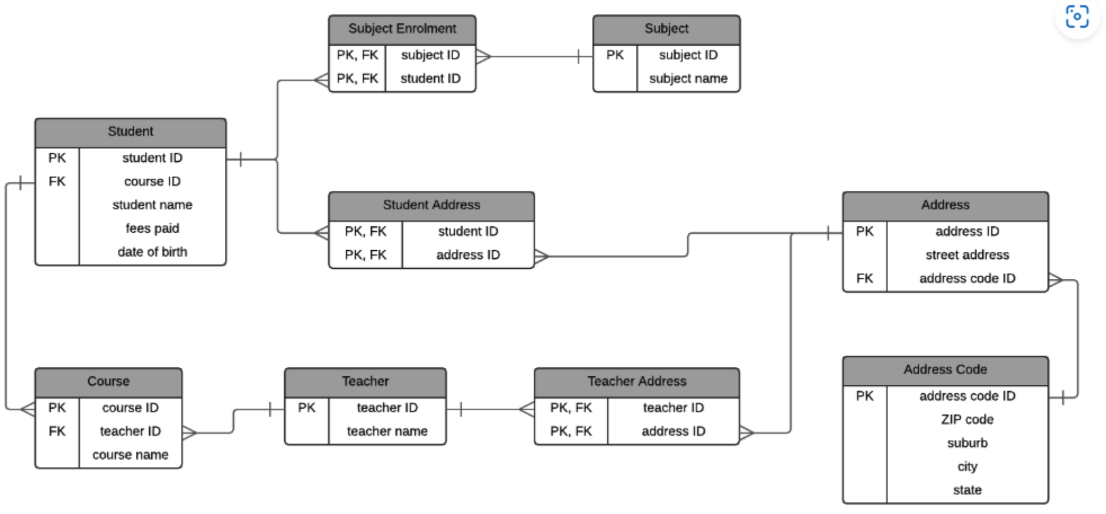
Subject Enrolment (student ID, subject ID)

Teacher (teacher ID, teacher name)

Teacher Address (teacher ID, address ID)

Course (course ID, teacher ID, course name)

An ERD would look like this:



A few changes have been made here:

- The address code ID has been removed from the Student table, because the relationships between student and address is now captured in the joining table called Student Address.
- The teacher's address is also captured in the joining table Teacher Address, and the address code ID has been removed from the Teacher table. I couldn't think of a better name for each of these tables.
- Address still links to address code ID

This design will let you do a few things:

- Store multiple addresses for each student or teacher
- Store additional information about an address relationship to a teach or student, such as an effective date (to track movements over time) or an address type (postal, residential)
- Determine which students or teachers live at the same address with certainty (it's linked to the same record).

So, that's how you can achieve fourth normal form on this database.

There are a few enhancements you can make to this design, but it depends on your business rules:

- Combine the student and teacher tables into a person table, as they are both effectively people, but teachers teach a class and students take a class. This table could then link to subjects and define that relationship as "teaches" or "is enrolled in", to cater for this.
- Relate a course to a subject, so you can see which subjects are in a course
- Split the address into separate fields for unit number, street number, address line 1, address line 2, and so on.
- Split the student name and teacher name into first and last names to help with displaying data and sorting.

These changes could improve the design, but I haven't detailed them in any of these steps as they aren't required for fourth normal form.

I hope this explanation has helped you understand what the normal forms are and what normalization in DBMS is.

[A Step-By-Step Guide to Normalization in DBMS With Examples (databasestar.com)]