Normal forms and normalization

An example of normalization using normal forms

We assume we have an enterprise that buys products from different supplying companies, and we would like to keep track of our data by means of a database.

We would like to keep track of what kind of **products** (e.g. cars) we buy, and from which **supplier** (e.g. Volvo) we buy them. We can buy each product from several suppliers.

Further, we want to know the **price** of the products. The price of a product is naturally dependent on which supplier we bought it from.

We would also like to store the address of each supplier, i.e. the **city** where the supplier is located. Here, we assume that each supplier is only located at one place.

Perhaps we suddenly need to order a large number of cars from Volvo. It can then be good to know how many people that live in the city where Volvo is located. Thus, we can know if Volvo can employ a sufficient amount of people to produce the cars. Hence we also store the **population** for each city.

A first try to design the database

We will try to store the data in a table called *PURCHASE*.

Table 1:

PRODUCT	SUPPLIER	PRICE	CITY	POPULATION
Cars	Volvo	100000	Torslanda	80000
Cars	SAAB	150000	Södertälje	50000
Trucks	SAAB	400000	Södertälje	50000
Aspirin	Astra	10	Södertälje	50000

The PRODUCT and SUPPLIER attributes provide the primary key.

This is not an especially good solution of designing the database schema.

Unnecessary redundancy. Certain information is stored in several places such as the data that SAAB is located in Södertälje. This takes up unnecessary space and it is also easy to introduce inconsistencies in the database if one forgets to perform updates for all occurrences.

Certain things can not be stored. This can become a problem for both updates and removals. Assume that we found a supplier that we would like to store in the database before we have bought something from the company. What should we write in the product attribute? On the other hand if we stop buying cars from Volvo and removes that row from the table we loose all information about the Volvo company.

The table has an unclear semantics, that means that it may be hard to understand the meaning of the table. Each table should preferably describe only one type of things and each row should contain data about one such thing. So what does one row means in our table? Does the first row say that we buy cars from Volvo? If that is true, what have the address of Volvo and the population of Torslanda to do with that fact?

The solution to these problems is to decompose the table. But how?

How do you divide your tables?

The basic rule is that **each table should describe one type of things, each row in the table should contain about one such thing, and the data we stored for each thing should exist in only one row**. For example, we can have a table that describes suppliers where each row contains data about one supplier. Thus, one type of things per table, one thing per row, and one row per things.

This can often be sufficient to know. If one follows this basic rule, ones databases will get a good design and one avoids problems with redundancy, things that will not be possible to store, and tables that is hard to understand.

However, sometimes it is difficult to actually know what kind of "things" it is that one would like to store and which data that is related to them. Then we can take use of the theory of normalization. It helps us to see exactly how different columns within a table are related and shows us how to divide the table to avoid our problems. Therefore we will start looking at the different **normal forms** that the theory of normalization describes. Normal forms are conditions that tables should fulfill. The simplest form is the first normal form and by adding more conditions one can define the second normal form, third normal form and further on.

The First Normal Form - 1NF

The first normal form only says that the table should only include **atomic** values, i.e. one value per box. For example, we can not in Table 1 above put in both Volvo and SAAB in the same box even if we buy cars from both suppliers. We must use to different rows for storing that. In most RDBMSs it is not allowed to assign more than one value to each box that results in that all tables are in first normal form.

Functional Dependency, FD

If we look in the example table, Table 1, we realize that on each row where it says **Södertälje** in the CITY column it will also say **50000** in the POPULATION column. This fact is called that the column POPULATION is **functionally dependent** of the CITY column.

More formally we can say that if the value of one (or several) attribute A unambiguously determines the value of another attribute B, the B is functionally dependent of A. This an be written as **A -> B** and we call A for the **determinant** since it determines B.

In the example table, Table 1, we have the following functional dependencies:

SUPPLIER -> CITY
CITY -> POPULATION
PRODUCT, SUPPLIER -> PRICE
SUPPLIER -> POPULATION

In the following, we will sometimes use the abbreviation FD for "functional dependency".

Full Functional Dependency, FFD

But wait! Are there not additional functional dependencies in Table 1?

We have seen that on every row where it says **Södertälje** in the CITY column it will also say **50000** in the POPULATION column. That meant that the column POPULATION is functionally dependent of the CITY column.

But then we can also say that where it says **Södertälje** in the CITY column and **Car** in the PRODUCT column, it will also say **50000** in the POPULATION column. Hence, the POPULATION column is functionally dependent of the combined CITY and PRODUCT columns.

However, this is a bit ridiculous so therefore one has defined a concept called **full functional dependency** which is a *minimal* functional dependency where one *can not remove any attributes from the determinant* and still keep the functional dependency. Another way of expressing this is to say that **the determinant is minimal**.

In the following we speak of full functional dependencies and when we speak about a determinant we mean one (or several) columns that another column is fully functional dependent of. We also sometimes abbreviate full functional dependency as FFD.

How do one know which dependencies that exists

Which fully functional dependencies exist in this table?,

Table 2:

A	B	C	D
1	4	10	100
2	5	20	50
3	6	20	200
1	4	10	200
2	6	20	0
3	6	20	300
1	4	10	null
2	6	20	50
3	6	20	50

The answer is that we do not know! Which dependencies that exist is not dependent on the data that for the moment happens to be in the table, but rather in the logics behind the table.

On the other hand one can see which FFD's that *can* exist by examining if there is a contradiction between any of the suggested FFD's and the data currently existing in the table. Actually, more properly one should consider the value domain for each attribute.

In our example we have:

A -> C

B -> C

Further, there can not exist an FFD A -> B since that for the same value (i.e. 2) there exist two different values on B (i.e. 5 and 6).

The Second Normal Form - 2NF

The second normal form says that a table, despite being in 1NF, is not allowed to contain any full functional dependencies on components of the primary key. If one graphically represents the FFD's between attributes, there should not be any arrows from components of the primary key, only from the complete primary key.

A first definition of non-key attribute: *a non-key attribute is an attribute that is not included in the primary key.*

A first definition of 2NF: *To fulfill 2NF a table should fulfill 1NF and in addition every non-key attribute should be FFB of the complete primary key.*

In our example in Table 1 we had a primary key combined by the two attribute PRODUCT and SUPPLIER, while there were two FFD's from the attribute SUPPLIER. These were:

SUPPLIER -> CITY
SUPPLIER -> POPULATION

These FFD's violate the 2NF and we therefore decompose Table 1 into two new tables.

The first new table, Table 3, is called *PURCHASE* where the PRODUCT and SUPPLIER attributes are still the primary key.

Table 3:

PRODUCT	SUPPLIER	PRICE
Cars	Volvo	100000
Cars	SAAB	150000
Trucks	SAAB	400000
Aspirin	Astra	10

The second table, Table 4, is called *SUPPLIER* where the SUPPLIER attribute is the primary key.

Table 4:

SUPPLIER	CITY	POPULATION
Volvo	Torslanda	80000
SAAB	Södertälje	50000
Astra	Södertälje	50000

Both these tables fulfills 2NF. Now the information that SAAB is located in Södertälje only exist in one place. We can also add a new supplier without the need to buy any products from it.

A Better Definition of 2NF

The previous definition of the 2NF is valid in a table with only one candidate key that also become the primary key. This primary key can of course be composed of several attributes but there are no other (minimal) attribute combination that is guarantied to be unique for each row.

However, in general there can exist several candidate keys in a table. If we would like to solve the problem that 2NF addresses, one must take all candidate keys into account in that case and not only the primary key.

We illustrate this in an example. Consider that we introduce a unique NUMBER for each PURCHASE relationship and stores that number as a column in the original PURCHASE table (Table 1). This number then become another candidate key in addition to the former combination of PRODUCT and SUPPLIER. This is illustrated in

Table 5:

NUMBER	PRODUCT	SUPPLIER	PRICE	CITY	POPULATION
1	Cars	Volvo	100000	Torslanda	80000
2	Cars	SAAB	150000	Södertälje	50000
3	Trucks	SAAB	400000	Södertälje	50000
4	Aspirin	Astra	10	Södertälje	50000

If we chose the combination of PRODUCT and SUPPLIER as primary key, exactly as before, everything is okay. The table does not fulfill 2NF and must be decomposed. On the other hand, if we chose the NUMBER as primary key, the table will immediately fulfill 2NF. Due to the fact that all non-key attributes are dependent of the complete primary key (anything else would be suspicious since the primary key is a simple attribute). Nevertheless, all problems with redundancy are still there, even if the table is in 2NF.

This calls for a better formulation of the 2NF that also works for the case with several candidate keys.

A better definition of non-key attribute: *a non-key attribute is an attribute that is not included in any candidate key.*

A better definition of 2NF: *To fulfill 2NF a table should fulfill 1NF and in addition every non-key attribute should be FFD of every candidate key.*

The Third Normal Form - 3NF

When Table 1 was decomposed in accordance with 2NF some problems with the design disappeared. But we are not finished yet! There are still several places that says that there live 50000 people in Södertälje and we can not insert a CITY where there are no suppliers. To be able to solve these problems we must apply **the third normal form**.

The third normal form says that a table, except from being in 2NF, should not include any **transitive** dependencies to non-key attributes. Thus, it is **not allowed to be any arrows between attributes outside the primary key**, only from the primary key to attributes outside. This means that if there is a combined primary key one is allowed to have arrows that points to one of the attributes in the key.

Definition of 3NF: *To fulfill 3NF a table should fulfill 2NF and in addition no non-key attribute should be FFD of any other non-key attribute.*

Again, since the *SUPPLIER* table does not fulfill 3NF we decompose that table into two new tables.

The first new table, Table 6, is called *SUPPLIER* with the SUPPLIER attribute as the primary key.

Table 6:

SUPPLIER	CITY
Volvo	Torslanda
SAAB	Södertälje
Astra	Södertälje

The second table, Table 7, is called *CITY* and has the CITY attribute as the primary key.

Table 7:

CITY	POPULATION
Torslanda	80000
Södertälje	50000

Both these tables fulfills 3NF. We have removed redundant population data and can also add a new city without providing information about any supplier.

How does one know how to perform the decomposition?

What would have happened if we instead had divided the *SUPPLIER* table in the following way.

One new table new table, Table 8, called *SUPPLIER* with the SUPPLIER attribute as primary key.

Table 8:

SUPPLIER	CITY
Volvo	Torslanda
SAAB	Södertälje
Astra	Södertälje

Another table, Table 9, called *CITY* that also has the SUPPLIER attribute as the primary key. In this table one can retrieve how big the population is in the town where a specific supplier is located.

Table 9:

SUPPLIER	POPULATION
Volvo	80000
SAAB	50000
Astra	50000

Both these tables also fulfill 3NF! There are no transitive dependencies. However, it seems like this was a rather stupid division. All problems we tried to avoid by applying 3NF still exists.

Rule of thumb: **Avoid stupid decompositions**. Think about the meaning of the tables.

Boyce-Codds Normal Form - BCNF

The 3NF did allow FFD's into the primary key, i.e. it was allowed to have arrows pointing to attributes in the key. The Boyce-Codds normal form forbids this type of FFD's. Hence, BCNF is a stricter condition than 3NF and prohibits certain problems that can occur in 3NF.

The tables we have now have designed that fulfill 3NF are also actually fulfilling BCNF. This is normally true. If a database is designed to fulfill 3NF, it usually also fulfills BCNF.

Definition of BCNF: *To fulfill BCNF a table should fulfill 1NF and every determinant should be a candidate key.*

Stated alternatively: draw a schema where every FFD's as arrows. Now every arrow should have their origin in candidate keys. If there is an arrow that starts from anything else than a candidate key, the table is not in BCNF.

Here is an example of a table, Table 10, that fulfills 3NF but not BCNF. The table is used to store the length of Swedish streets. Streets are unique in every town but not in the whole of Sweden. Thus, there can not be two Storgatan in Gnesta but there can be one in Gnesta and one in Linköping. Each street (we assume) is located completely within one zipcode area. Further there can be several zip code areas in each city.

Table 10:

STREET	ZIPCODE	CITY	LENGTH
Rydsvägen	58248	Linköping	19 km
Mårdtorpsgatan	58248	Linköping	0.7 km
Storgatan	58223	Linköping	1.5 km
Storgatan	64631	Gnesta	0.014 km

There are two candidate keys that both contain two attributes. The first is STREET and ZIPCODE and the second is STREET and CITY.

The table include the following full functional dependencies:

STREET, ZIPCODE -> LENGTH
STREET, CITY -> LENGTH
STREET, CITY -> ZIPCODE
ZIPCODE -> CITY

Hence, there is one attribute, ZIPCODE, that determines a key attribute, CITY. This is fine in 3NF. But we see that there is redundancy in the table (the data that the zip code 58248 is found in Linköping is repeated twice). Furthermore, it is not possible to insert information about a zip code area without adding information of at least one street.

Thus the table is not in BCNF and we will decompose it into two new relations, one for streets and one for zip code areas.

Table 11:

STREET	ZIPCODE	LENGTH
Rydsvägen	58248	19 km
Mårdtorpsgatan	58248	0.7 km
Storgatan	58223	1.5 km
Storgatan	64631	0.014 km

Table 12:

ZIPCODE	CITY
58248	Linköping
58223	Linköping
64631	Gnesta

Now these two tables fulfill BCNF and everything is fine.

Additional Normal Forms

Yes, there are additional normal forms that evaluates other types of dependencies within database schemas.

Always Normalize or Not?

Sometimes it is inconvenient to normalize. For instance in an address directory it might be good to have both zip code and city in the same table as the street address despite its violation of the 3NF. The semantics of the design will probably become clearer in this case. Another reason for not to normalize could be efficiency reasons but these could perhaps also be avoided by adding suitable views for instance.

Rule of thumb: Use the theory with sense - it is not always a good idea to normalize. On the other hand, if you decide on a lower degree of normalization, there ought to be good reasons for that as well. You should be aware of what problems that can arise and you should document your degree of normalization together you reasons for choosing that level and the potential problems that can be foreseen due to these decisions.

Normalization in other data models

Normalization is not only applicable to the relational data model, but for other data models as well such as object-oriented models and records.