Normalization-Part I

Hi! Here in this lecture we are going to discuss about the important concepts of DBMS, Normalization.

What is Normalization?

Yes, but what is this normalization all about? If I am simply putting it, normalization is a formal process for determining which fields belong in which tables in a relational database. Normalization follows a set of rules worked out at the time relational databases were born. A normalized relational database provides several benefits:

- Elimination of redundant data storage.
- Close modeling of real world entities, processes, and their relationships.
- Structuring of data so that the model is flexible.
- Normalization ensures that you get the benefits relational databases offer. Time spent learning about normalization will begin paying for itself immediately.

Why do they talk like that?

Some people are intimidated by the language of normalization. Here is a quote from a classic text on relational database design:

A relation is in third normal form (3NF) if and only if it is in 2NF and every nonkey attribute is no transitively dependent on the primary key.

Huh? Relational database theory, and the principles of normalization, was first constructed by people intimately acquainted with set theory and predicate calculus. They wrote about databases for like-minded people. Because of this, people sometimes think that normalization is "hard". Nothing could be more untrue. The principles of normalization are simple, commonsense ideas that are easy to apply.

Design versus Implementation

Now we will look in to the aspects regarding the tasks associated with designing and implementing a database.

Designing a database structure and implementing a database structure are different tasks. When you design a structure it should be described without reference to the specific database tool you will use to implement the system, or what concessions you plan to make for performance reasons. These steps come later. After you've designed the database structure abstractly, then you implement it in a particular environment--4D in our case. Too often people new to database design combine design and implementation in one step. 4D makes this tempting because the structure editor is so easy to use. Implementing a structure without designing it quickly leads to flawed structures that are difficult and costly to modify. Design first, implement second, and you'll finish faster and cheaper.

Normalized Design: Pros and Cons

Oh, now we've implied that there are various advantages to producing a properly normalized design before you implement your system. Let's look at a detailed list of the pros and cons:

Pros of Normalizing	**Cons of Normalizing**
More efficient database structure. Better understanding of your data. More flexible database structure. Easier to maintain database structure. Few (if any) costly surprises down	You can't start building the database before you know what the user needs.

the road. Validates your common sense and intuition. Avoids redundant fields. Ensures that distinct tables exist when necessary.	

We think that the pros outweigh the cons.

Terminology

There are a couple terms that are central to a discussion of normalization: "key" and "dependency". These are probably familiar concepts to anyone who has built relational database systems, though they may not be using these words. We define and discuss them here as necessary background for the discussion of normal forms that follows.

The Normal Forms

Hi! Now we are going to discuss the definitions of Normal Forms.

1st Normal Form (1NF)

Def: A table (relation) is in 1NF if

1. There are no duplicated rows in the table.
2. Each cell is single-valued (i.e., there are no repeating groups or arrays).
3. Entries in a column (attribute, field) are of the same kind.

Note: The order of the rows is immaterial; the order of the columns is immaterial. The requirement that there be no duplicated rows in the table means that the table has a key (although the key might be made up of more than one column—even, possibly, of all the columns).

So we come to the conclusion,

A relation is in 1NF if and only if all underlying domains contain atomic values only.

The first normal form deals only with the basic structure of the relation and does not resolve the problems of redundant information or the anomalies discussed earlier. All relations discussed in these notes are in 1NF.

For example consider the following example relation:

student(sno, sname, dob)
Add some other attributes so it has anomalies and is not in 2NF

The attribute *dob* is the date of birth and the primary key of the relation is *sno* with the functional dependencies *sno -> sname* and $sno \rightarrow dob$. The relation is in 1NF as long as *dob* is considered an atomic value and not consisting of three components *(day, month, year)*. The above relation of course suffers from all the anomalies that we have discussed earlier and needs to be normalized. (add example with date of birth)

A relation is in first normal form if and only if, in every legal value of that relation every tuple contains one value for each attribute

The above definition merely states that the relations are always in first normal form which is always correct. However the relation that is only in first normal form has a structure those undesirable for a number of reasons.

First normal form (1NF) sets the very basic rules for an organized database:

- Eliminate duplicative columns from the same table.
- Create separate tables for each group of related data and identify each row with a unique column or set of columns (the primary key).

2nd Normal Form (2NF)

Def: A table is in 2NF if it is in 1NF and if all non-key attributes are dependent on the entire key.

The second normal form attempts to deal with the problems that are identified with the relation above that is in 1NF. The aim of second normal form is to ensure that all information in one relation is only about one thing.

A relation is in 2NF if it is in 1NF and every non-key attribute is fully dependent on each candidate key of the relation.

Note: Since a partial dependency occurs when a non-key attribute is dependent on only a part of the (composite) key, the definition of 2NF is sometimes phrased as, "A table is in 2NF if it is in 1NF and if it has no partial dependencies."

Recall the general requirements of 2NF:

- Remove subsets of data that apply to multiple rows of a table and place them in separate rows.
- Create relationships between these new tables and their predecessors through the use of foreign keys.

These rules can be summarized in a simple statement: 2NF attempts to reduce the amount of redundant data in a table by extracting it, placing it in new table(s) and creating relationships between those tables.

Let's look at an example. Imagine an online store that maintains customer information in a database. Their Customers table might look something like this:

CustNum	**FirstName**	**LastName**	**Address**	**City**	**State**	**ZIP**
1	John	Doe	12 Main Street	Sea Cliff	NY	11579
2	Alan	Johnson	82 Evergreen	Sea Cliff	NY	11579

			Tr			
3	Beth	Thompson	1912 NE 1st St	Miami	FL	33157
4	Jacob	Smith	142 Irish Way	South Bend	IN	46637
5	Sue	Ryan	412 NE 1st St	Miami	FL	33157

A brief look at this table reveals a small amount of redundant data. We're storing the "Sea Cliff, NY 11579" and "Miami, FL 33157" entries twice each. Now, that might not seem like too much added storage in our simple example, but imagine the wasted space if we had thousands of rows in our table. Additionally, if the ZIP code for Sea Cliff were to change, we'd need to make that change in many places throughout the database.

In a 2NF-compliant database structure, this redundant information is extracted and stored in a separate table. Our new table (let's call it ZIPs) might look like this:

ZIP	**City**	**State**
11579	Sea Cliff	NY
33157	Miami	FL
46637	South Bend	IN

If we want to be super-efficient, we can even fill this table in advance -- the post office provides a directory of all valid ZIP codes and their city/state relationships. Surely, you've encountered a situation where this type of database was utilized. Someone taking an order might have asked you for your ZIP code first and then knew the city and state you were calling from. This type of arrangement reduces operator error and increases efficiency.

Now that we've removed the duplicative data from the Customers table, we've satisfied the first rule of second normal form. We still need to use a foreign key to tie the two tables together. We'll use the ZIP code (the primary key from the ZIPs table) to create that relationship. Here's our new Customers table:

CustNum	**FirstName**	**LastName**	**Address**	**ZIP**
1	John	Doe	12 Main Street	11579
2	Alan	Johnson	82 Evergreen Tr	11579
3	Beth	Thompson	1912 NE 1st St	33157
4	Jacob	Smith	142 Irish Way	46637
5	Sue	Ryan	412 NE 1st St	33157

We've now minimized the amount of redundant information stored within the database and our structure is in second normal form, great isn't it?

Let's take one more example to confirm the thoughts

The concept of 2NF requires that all attributes that are not part of a candidate key be *fully* dependent on each candidate key. If we consider the relation

student (sno, sname, cno, cname)

and the functional dependencies

sno → cname
cno -> cname

and assume that *(sno, cno)* is the only candidate key (and therefore the primary key), the relation is not in 2NF since *sname* and *cname* are not fully dependent on the key. The above relation suffers from the same anomalies and repetition of information as discussed above since *sname* and *cname* will be repeated. To resolve these difficulties we could remove those attributes from the relation that are not fully dependent on the candidate keys of the relations. Therefore we decompose the relation into the following projections of the original relation:

S1 (sno, sname)
S2 (cno, cname)
SC (sno, cno)

Use an example that leaves one relation in 2NF but not in 3NF. We may recover the original relation by taking the natural join of the three relations. If however we assume that *sname* and *cname* are unique and therefore we have the following candidate keys

(sno, cno)
(sno, cname)
(sname, cno)
(sname, cname)

The above relation is now in 2NF since the relation has no non-key attributes. The relation still has the same problems as before but it then does satisfy the requirements of 2NF. Higher level normalization is needed to resolve such problems with relations that are in 2NF and further normalization will result in decomposition of such relations

3rd Normal Form (3NF)

Def: A table is in 3NF if it is in 2NF and if it has no transitive dependencies.

The basic requirements of 3NF are as follows

- Meet the requirements of 1NF and 2NF
- Remove columns that are not fully dependent upon the primary key.

Although transforming a relation that is not in 2NF into a number of relations that are in 2NF removes many of the anomalies that appear in the relation that was not in 2NF, not all anomalies are removed and further normalization is sometime needed to ensure further removal of anomalies. These anomalies arise because a 2NF relation may have attributes that are not directly related to the thing that is being described by the candidate keys of the relation. Let us first define the 3NF.

A relation R is in third normal form if it is in 2NF and every non-key attribute of R is non-transitively dependent on each candidate key of R.

To understand the third normal form, we need to define *transitive dependence* which is based on one of Armstrong's axioms. Let *A*, *B* and *C* be three attributes of a relation *R* such that $A \rightarrow B$and $B \rightarrow C$. From these FDs, we may derive $A \rightarrow C$. As noted earlier, this dependence $A \rightarrow C$is *transitive*.

The 3NF differs from the 2NF in that all non-key attributes in 3NF are required to be *directly* dependent on each candidate key of the relation. The 3NF therefore insists, in the words of Kent (1983) that all facts in the relation are about the key (or the thing that the key identifies), the whole key and nothing but the key. If some attributes are dependent on the keys transitively then that is an indication that those attributes provide information not about the key but about a kno-key attribute. So the information is not directly about the key, although it obviously is related to the key.

Consider the following relation

subject (cno, cname, instructor, office)

Assume that *cname* is not unique and therefore *cno* is the only candidate key. The following functional dependencies exist

$$cno \rightarrow cname$$
$$cno \rightarrow instructor$$
$$instructor \rightarrow office$$

We can derive $cno \rightarrow office$from the above functional dependencies and therefore the above relation is in 2NF. The relation is however not in 3NF since office is not directly dependent on *cno*. This transitive dependence is an indication that the relation has information about more than one thing (viz. course and instructor) and should therefore be decomposed. The primary difficulty with the above relation is that an instructor might be responsible for several subjects and therefore his office address may need to be repeated many times. This leads to all the problems that we identified at the beginning of this chapter. To overcome these difficulties we need to decompose the above relation in the following two relations:

s (cno, cname, instructor)
ins (instructor, office)

s is now in 3NF and so is *ins*.

An alternate decomposition of the relation *subject* is possible:

s(cno, cname)
inst(instructor, office)
si(cno, instructor)

The decomposition into three relations is not necessary since the original relation is based on the assumption of one instructor for each course.

The 3NF is usually quite adequate for most relational database designs. There are however some situations, for example the relation *student(sno, sname, cno, cname)* discussed in 2NF above, where 3NF may not eliminate all the redundancies and inconsistencies. The problem with the relation *student(sno, sname, cno, cname)* is because of the redundant information in the candidate keys. These are resolved by further normalization using the BCNF.

Imagine that we have a table of widget orders:

Order Number	**Customer Number**	**Unit Price**	**Quantity**	**Total**
1	241	$10	2	$20
2	842	$9	20	$180

3	919	$19	1	$19
4	919	$12	10	$120

Remember, our first requirement is that the table must satisfy the requirements of 1NF and 2NF. Are there any duplicative columns? No. Do we have a primary key? Yes, the order number. Therefore, we satisfy the requirements of 1NF. Are there any subsets of data that apply to multiple rows? No, so we also satisfy the requirements of 2NF.

Now, are all of the columns fully dependent upon the primary key? The customer number varies with the order number and it doesn't appear to depend upon any of the other fields. What about the unit price? This field could be dependent upon the customer number in a situation where we charged each customer a set price. However, looking at the data above, it appears we sometimes charge the same customer different prices. Therefore, the unit price is fully dependent upon the order number. The quantity of items also varies from order to order, so we're OK there.

What about the total? It looks like we might be in trouble here. The total can be derived by multiplying the unit price by the quantity; therefore it's not fully dependent upon the primary key. We must remove it from the table to comply with the third normal form:

Order Number	Customer Number	Unit Price	Quantity
1	241	$10	2
2	842	$9	20
3	919	$19	1
4	919	$12	10

Now our table is in 3NF.

Revision Questions

1. What is normalization
2. Explain the pros and cons of normalization
3. Explain First, Second and Third normal forms