DBS211

Week 10 - Normalization 2 (2NF and 3NF)

Table of Contents

Reading Materials

- Textbook, Chapter 12
- Wikipedia - Database Normalization
- Wikipedia - 2NF
- Wikipedia - 3NF

Welcome to Week 10

Continuing where we left off last week, we continue to path through normalization and look in much more depth, the concept of attribute dependencies. Understanding the dependencies between attributes and the primary key is essential to great database design. This week, we will explore the area of both partial and transitive dependencies, how to find them and how to eliminate them resulting in a final 3NF database design.

After this week the student will be able to:

- identify the 2 variants of partial dependencies
- eliminate partial dependencies and normalize a database design to 2NF
- identify transitive dependencies and how to resolve them
- eliminate transitive dependencies and normalize a database design to 3NF

2NF - Partial Dependencies

Second normal form always starts with a database already in 1NF and then identifies partial dependencies. Partial dependencies are relationships where the attribute in focus is not dependent on the entire primary key or is dependent on more than the primary key. To satisfy 2NF, each attribute in an entity must be wholly dependent on the entire primary key. For this particular form, we need to review every single non-pk attribute for partial dependencies.

The 2 Variations of Partial Dependencies

An attribute partially dependent on a composite primary key

Let's review our soccer league again to display an example of this. We will use the business rule where teams have more than one player and each player can play on more than one team.

```
UNF
TEAMS       [teamID, teamName, ShirtColour, (PlayerID, PlayerName, PlayerDOB, ShirtNumber)]

1NF
TEAMS       [teamID, teamName, ShirtColour]
TEAM_PLAYERS[FK teamID, PlayerID, PlayerName, PlayerDOB, ShirtNumber]
```

IN the 1NF solution above we will investigate all non-PK attributes to detect partial dependencies

- ✓**teamName** - is team specific and does not depend on any field other than teamID
- ✓**ShirtColour** - is team specific and depends only on teamID. All players on one team wear the same colour shirt.
- **PlayerName** - is player specific and depends only on playerID. However the TEAM_PLAYERS entity has a composite key with teamID. PlayerName has no relation with teamID and therefore has a partial dependency with the entity's PK and must be moved.
- **PlayerDOB** - same situation as PlayerName
- ✓**ShirtNumber** - The number on the back of the players shirt will depend on the player, as no two players on the same team wear the same number, however, the player may wear a different number when playing on a different team. Therefore, the ShirtNumber depends on both the teamID and the PlayerID. This matches the composite primary key and is therefore in the right place.

from above, we can see that PlayerName and PlayerDOB both have partial dependencies with the PK of the TEAM_PLAYERS entity and therefore can not be in that entity. Since they depend only on the playerID, we need a new entity that has a single field PK of playerID. Let us therefore create a PLAYER entity and place those fields in it.

```
TEAMS       [teamID, teamName, ShirtColour]
TEAM_PLAYERS[FK teamID, FK PlayerID, ShirtNumber]
PLAYERS     [PlayerID, PlayerName, PlayerDOB]
```

In this case, the playerID is also left behind in the TEAM_PLAYERS entity in order to maintain the relationship, and will be a FK to the new entity, which now possess playerID as a PK.

Lastly, in order to satisfy 2NF, the solution must also be UNF and 1NF. Checking each attribute in all entities, we can see these are satisfied, and therefore this is a good final 2NF solution.

An attribute dependent on more than one field, but only a single field PK exists

In this example we will again review the soccer league database to demonstrate this.

```
TEAMS       [teamID, teamName, ShirtColour]
TEAM_PLAYERS[FK teamID, FK PlayerID]
PLAYERS     [PlayerID, PlayerName, PlayerDOB, ShirtNumber]
```

in the UNF solution above, we included shirtNumber in the repeating group parenthesis as it absolutely would go with the player. So it might seem natural to continue to keep the shirtNumber with player.

Now we can revisit the business rule associated with players and teams. If the player can only play on one team, then the shirt number would belong to the player and the team would not necessarily determine the number directly. But, in the case where a player may play on more than one team, then the player number can be different for each team and therefore the shirt number depends on both the player id and the team id. Because it is in the Players entity, which only contains the playerID, it has a partial dependency with the PK. Therefore it must move to an entity where both the teamID and the playerID are together a composite primary key (the TEAM_PLAYERS entity).

```
TEAMS       [teamID, teamName, ShirtColour]
TEAM_PLAYERS[FK teamID, FK PlayerID, ShirtNumber]
PLAYERS     [PlayerID, PlayerName, PlayerDOB]
```

The Use of Surrogate Keys and Normalization

Now is probably a good time to introduce the reason why we would not introduce surrogate keys, such as auto generated ID type fields, to replace composite keys yet. Some of you ave heard that composite keys are not usually left in database designs and they are often replaced with an autonumber ID field. Although this is very true and a good practice, it should not be done just yet.

When trying to determine things like partial and transitive dependencies, it must be made very clear that these dependencies must be determined with respect to the intended, oro original key fields. The surrogate keys are a replacement key for what was the original fields and therefore, those original fields must be used for the dependency analysis. If a surrogate key is introduced too early, then determining partial and transitive dependencies is greatly complicated.

For Example: if we were to replace the composite key in TEAM_PLAYERS with an autonumber field it might look like this.

```
TEAMS       [teamID, teamName, ShirtColour]
TEAM_PLAYERS[rosterID, FK teamID, FK PlayerID, ShirtNumber]
PLAYERS     [PlayerID, PlayerName, PlayerDOB]
```

Now looking at the ShirtNumber attribute, it is not nearly as clear that it is not partially dependent on the primary key. You have to remember that the surrogate key, rosterID, is a replacement for the original composite key between playerID and teamID and those are the fields in which the dependencies must be determined.

> Don't introduce the surrogate keys to replace composite keys until after normalization has been completed.

3NF - Transitive Dependencies

Transitive dependencies are dependencies where an attribute depends on another non-pk attribute in the same entity. This one is a little tricky to see, but absolutely needs to be taken care of.

Let us look at the following example to visualize transitive dependencies.

一个属性依赖于其他非 `primary key` 属性

```
CUSTOMERS       [CustomerID, CustFName, CustLName, CustPhone, SalesRepID, SaleRepName]
```

Transitive dependencies can often be discovered by asking the question, if I change one non-key field, does it force me to change another non-key field. If the answer is yes, you have a transitive dependency.

Looking at each field in the above CUSTOMERS entity, and asking the question above, it becomes obvious very quickly that by changing the SalesRepID, the SalesRepName would also have to change. Neither of these attributes are PKs or CKs, therefore SalesRepName has a transitive relationship with CustomerID, the PK for the entity. SalesRepName depends on SaleRepID, which in turn depends on CustomerID. This is Transitive.

To solve this, We move SalesRep details to another entity and only leave behind the ID as a FK to the newly created PK of the new entity.

移出属性的同时需要在原地留下 `FK` 标记

```
CUSTOMERS       [CustomerID, CustFName, CustLName, CustPhone, FK SalesRepID]
SALESREPS       [SalesRepID, SalesRepName]
```

Another More Complex Transitive Dependency Example

A good example of transitive dependencies are often in the automotive industry. A vehicle typically is referred to by its' make, model and version (example: Honda Civic LX) and each vehicle is uniquely identified by its' Vehicle Identification Number (VIN). The DBDL for the VEHICLE entity might look something like this.

```
VEHICLE         [VIN, make, model, version, colour, NumDoors, Transmission, ..... etc.]
```

and a data table may look something like:

VIN	Make	Model	Version	Colour	NumDoors	Transmission	...
FDSAGFSD8G68DG76SD	Honda	Civic	EX	Black	4	4-speed Automatic	...
789FDSA6796G7D8AS68	Honda	Civic	LX Hatchback	Forest Green	2	5-speed Manual	...
YUIS79S8DFS789789GSS	Ford	Mustang	GT	Bumblebee Yellow	2	5-speed Manual	...

The above example is full of transitive dependencies that can be tricky to navigate, to let's look at one at a time working right to left.

- **Transmission -** Any car can have a different transmission as this is an option when you buy a car. You can buy a Honda Civic LX with either an automatic or a manual transmission and therefore transmission is really independent of model of make and depends solely on the individual car (i.e. the VIN). Therefore, this is not a transitive dependency.
- ✓ **NumDoors -** This is a strange one as it could go either way. But I would definitely say that changing the version from LX to LX Hatchback would absolutely change the number of doors. Therefore, I would say the number of doors has a transitive dependency, through version to VIN, the PK.
- **Colour -** Any car can be painted any colour and therefore it is vehicle dependent, the VIN. Therefore, not transitive.
- ✓ **Version -** The version of the car is another tricky one, as manufacturers often reuse version codes in different models. For instance there are both Honda Civic EX as well as Honda Accord EX, but there is no Honda Civic Gt. GT is a version of Ford products. Therefore, changing the Make or Model will absolutely force the Version to change. Therefore, Version has a transitive dependency with VIN.
- ✓ **Model -** If I change the Make of the car, the model will absolutely have to change as well. For example, if in the first row above, I changed the Make from Honda to Toyota, the model and version would also have to change, meaning model depends on Make first

and the VIN and therefore has a transitive dependency with VIN, the PK.

- ✓ **Make -** the make also has a transitive dependency with VIN, although not as strongly. If you change Model, it is likely that the Make also has to change, but not every time. For instance, if Civic is change to Accord, the Make would remain as Honda, but if Civic was changed to Mustang, then the Make would have to change from Honda to Ford. This is still a transitive dependency as it is not entirely directly related to the PK field (VIN).

Wow, so how do we fix this?

so first thing we do is determine one of the fields and move them. Let us start with NumDoors. NumDoors depends on the Version, which depends on Model, which depends on Make. All somehow indirectly relate to VIN, but through transitive relationships. If we are to work this puzzle out, we first look at the field most directly related to VIN which can be unique enough. In this case that is Version, let us start there and move all related fields with it.

```
VEHICLE         [VIN, FK versionID, colour, Transmission, ..... etc.]
VERSIONS        [versionID, versionName, Model, Make, NumDoors]
We introduced a versionID field here is Version by itself is not a good candidate key.  (Remember Civic LX and Accord LX).
```

Now we still have transitive dependencies with make and model. The Number of Doors depends directly on the version of the car, so that is no longer transitive with its' PK.

For Make and Model we can follow the dependency line. The version depends on the Model and then the Make, so let's separate both those through the model.

```
VEHICLE         [VIN, FK versionID, colour, Transmission, ..... etc.]
VERSIONS        [versionID, versionName, FK Model, NumDoors]
MODEL           [Model, Make]
```

if we introduce ID fields for make and model here, we would see this.

```
VEHICLE         [VIN, FK versionID, colour, Transmission, ..... etc.]
VERSIONS        [versionID, versionName, FK ModelID, NumDoors]
MODEL           [ModelID, ModelName, MakeID, MakeName]
```

Now MakeName is dependent on MakeID and not ModelID the PK in that entity, so it must move.

```
VEHICLE         [VIN, FK versionID, colour, Transmission, ..... etc.]
VERSIONS        [versionID, versionName, FK ModelID, NumDoors]
MODEL           [ModelID, ModelName, FK MakeID]
MAKE            [MakeID, MakeName]
```

Now we have eliminated all transitive dependencies, verified there are no partial dependencies, all fields are atomic, there are no repeating groups, and all entities have a good candidate primary key. Therefore we have the design in 3NF.

Exceptions to Normalization

There are a few areas where normalization goes to far and will cause issues, such as data storage growth or a lack of available information. One of the key examples of this is to do with physical mailing addresses.

```
CUSTOMERS   [cID, name, phone, address1, street, city, province, country, postal_code]
```

In this case it is not so obvious, but there are several transitive dependencies here.

- Postal Codes are specific to countries and geographical areas
- province is dependent on country
- city is dependent on province
- street is dependent on city
- street house number is dependent on street

This gets extremely complicated and has some very specific requirements. First of all let us look at the normalized version of an address.

```
CUSTOMERS   [cID, name, phone, FK postal_code, house_number]
POSTAL_CODE [postal_code, FK streetID]
STREET      [streetID, streetName, FK City]
CITY        [city, FK provCode]
PROVINCE    [provCode, provName, FK CountryCode]
COUNTRY     [CountryCode, CountryName]
```

First off this seems absolutely ridiculous and it is. But in true 3NF form, this is what an address would look like. But let us look at more consequences using an example of an online shopping site that sells t-shirts:

> remember that normalization results in FK-PK relationships which require referential integrity, meaning data can not be added as a child record unless the parent record already exists. example: We could not add a province unless the country already existed, we could not add a street, unless the city already existed.

Either the store would have to restrict where their customers are located or they would end up with a very large database before they even sold a single t-shirt. Even if the store was restricted to Canada only, the database would need to contain every province, city, street, and postal code in the entire country to enable customers to enter their information. Additionally, new construction all the time means this database is growing and would need to be maintained with great effort.

This level of data storage and maintenance may be alright for very large companies, such as FedEx, but not for most small and medium businesses in the country. Therefore, we often by-pass the normalization process in order to save having to have to information pre-entered, the data storage requirements. Note that this will absolutely lead to errors in the data as humans enter the information, but the cost is worth it. Maybe the software can implement some checks using external APIs to verify addresses before saving them in the database.

The point here is that, sometimes completing normalization to the extreme is sometimes more costly than it is worth. So in some cases, we bypass normalization for simplicity.

> If you choose to bypass normalization as part of an assignment, project, lab or test, it would be best to confirm with your instructor before submitting your work.