# DBS211

# Week 9 - Normalization 1 (UNF and 1NF)

## Table of Contents

## Reading Materials

- Textbook, Chapter 12
- Wikipedia - Database Normalization
- Wikipedia - UNF
- Wikipedia - 1NF

## Welcome to Week 9 - Normalization

This week we dive into the process of designing a database from scratch when you have very little to work from. You might only have a few paper forms given to you to use as a base and you are expected to design a complete relational database. While utilizing the DBDL format and understanding field dependencies with the primary keys, you will be able to come up with a very satisfactory design for the database schema.

After this week, the student will understand what normalization is, apply DBDL to determine the un-normalized and first normal forms of the database schema.

## Database Design Fundamentals

Throughout the course to this point, we have focused on the physical properties of the database schema. We looked at several types of Keys, constraints, and the various forms that tables can take and we learned about calculated or derived fields and how fields relate to one another. Now it is time to take this a little further and learn about several different kinds of dependencies between each field and the key field for the table in which it resides. This understanding will help you ensure that fields are located in the right tables and that the tables required are all created.

### Where to start.....

We will start by going back and reviewing several topics that we covered before and make sure you grasp those concepts. All the below listed concepts are required for the database normalization process. So go back and quickly review the following chapters:

- Week 1 - Data Repetition and Redundancy
- Week 1 - Data Anomalies
- Week 2 - Various Types of Keys
- Week 2 - Various Types of Tables

Now that we have a refresher on those topics, we can move forward....

## DBDL

DBDL or DataBase Definition Language is a standardized way of describing entities in a relational database in a written format. It is very similar to the entity format from ERDs, except written in line as a paragraph of text would be written.

```
ENTITY_NAME [ KEY_FIELD, field2, FK field3, (field4, field5, field6), ... fieldn ]
```

The above is the basic syntax for a basic DBDL entity. A few points to note:

- The entity name is written in ALL-CAPS outside the square brackets []
- The Primary, or better referred to as Candidate, Key is underlined. If underlining is not possible, we write either PK or CK in front of it indicating Primary or Candidate Key.
- Fields that are related to fields in other entities are indicated using FK, for Foreign Key.
- The Round Brackets are used to distinguish groups of related data that may result in repeating groups (described in more detail below)

Once you start to obtain more entities, it may look something like this:

```
PLAYERS       [playerID, fname, lname, dob, preferred_position]
PLAYER_TEAM   [FK playerID, FK teamID, shirt_number, primary_position]
TEAMS         [teamID, team_name, home_location, manager_name]
```

Note in the above example, the PLAYER_TEAM entity has a Composite Candidate Primary Key and both fields of that key are also Foreign Keys (child records) to the other tables respectively. PLAYER_TEAM is an example of a junction or bridge entity.

## Introduction to Normalization

Now we are ready to work our conceptual design into a format that can be translated into tables for actual database implementation. Historically, this once had been a process of intuitive logic. For many students this intuitive approach is often used. Without a LOT of experience the intuitive process often leads to failure. As in the past the intuitive method often led to data redundancy, and designs where relationships could not exist.

Normalization entails organizing the columns (attributes) and tables (entities) of a database to ensure that their dependencies are properly enforced by database integrity constraints through relationships. It is accomplished by applying some formal rules either by a process of synthesis (creating a new database design) or decomposition (improving an existing database design).

There are 10 different forms of normalization today. However, most databases are considered well designed if they comply with the first 4 forms. The additional forms are typically only needed in advanced statistical analysis, machine learning and large data analysis processes. This course will only concentrate on the first four forms.

- **UNF - Unnormalized Form** - is the basic form simple utilized for obtaining and grouping attributes that are required. Repeating groups of data are typically indicated using parenthesis().
- **1NF - First Normal Form** - contains all the features of UNF plus the elimination of multi-value attributes or non-atomic fields.
- **2NF - Second Normal Form** - builds on 1NF by eliminating partial functional dependencies
- **3NF - Third Normal Form** - builds on 2NF by eliminating remaining transitive function dependencies

The following table shows the first 4 normal forms and displays the basic rules for each. Each of these are described further below and in next weeks content.

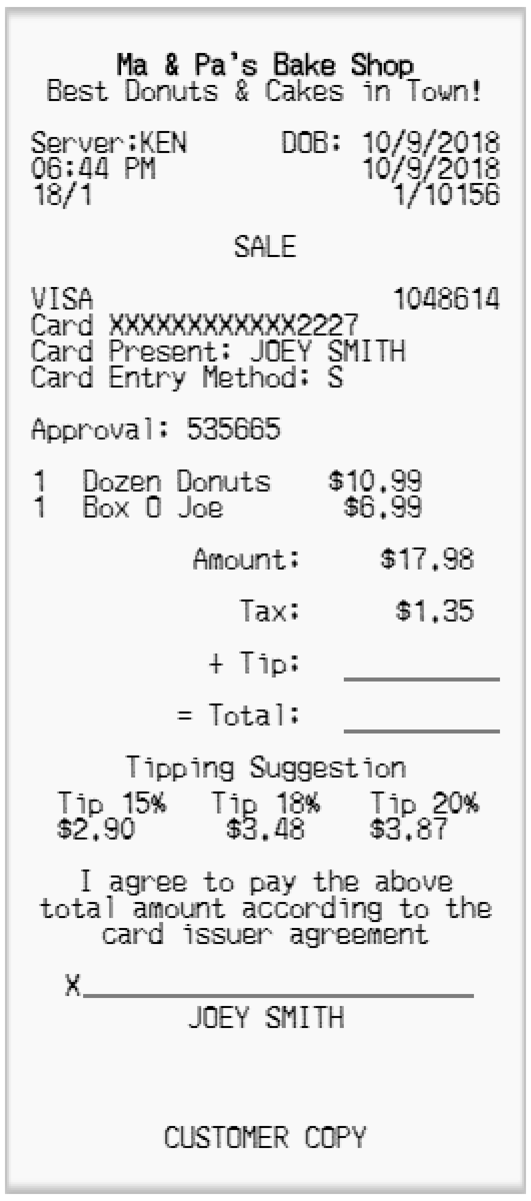
| Property | UNF | 1NF | 2NF | 3NF |
|---|---|---|---|---|
| Each Entity has a Candidate Primary Key | ✓ | ✓ | ✓ | ✓ |
| Repeating Groups Indicated by Parenthesis | ✓ | ✓ | ✓ | ✓ |
| Repeating Groups Eliminated | | ✓ | ✓ | ✓ |
| Atomic Fields Only | | ✓ | ✓ | ✓ |
| No Partial Dependencies | | | ✓ | ✓ |
| No Transitive Dependencies | | | | ✓ |

Let us now look at each normalized form in turn.

## Un-Normalized Form (UNF)

The un-normalized form is often derived from having a single form or a source of requirements in order to determine what attributes are included. But typically UNF will consist of a a simple list of attributes and then indicate which groups of attributes may have repeating data. Each UNF solution will clearly indicate the primary Entity of the subject matter and for each entity, indicate a candidate key that could be used to determine uniqueness. This is probably best learned through example.

### Example:



```
      Ma & Pa's Bake Shop
  Best Donuts & Cakes in Town!

Server:KEN        DOB: 10/9/2018
06:44 PM               10/9/2018
18/1                     1/10156

              SALE

VISA                     1048614
Card XXXXXXXXXXXX2227
Card Present: JOEY SMITH
Card Entry Method: S

Approval: 535665

1  Dozen Donuts    $10.99
1  Box O Joe        $6.99

         Amount:       $17.98

            Tax:        $1.35

          + Tip:     ________

        = Total:     ________

       Tipping Suggestion
 Tip 15%     Tip 18%    Tip 20%
 $2.90       $3.48      $3.87

   I agree to pay the above
 total amount according to the
    card issuer agreement

 X____________________________
          JOEY SMITH


         CUSTOMER COPY
```

The shown receipt is a simple typical receipt you would obtain from a small retail store. Let us use this receipt to learn about UNF at this point. By looking at this receipt in detail we can determine a significant amount of the attributes required and can use our knowledge to add additional ones needed.

The first step is to determine the **Primary Entity** used. There is no one particular entity that is the correct one, but some will make the process easier than others. Let us review potential entities from this receipt and choose a single one to start with.

- Bake Shop - since this appears to be an independent shop and not a franchise, there likely will only be one location
- Server - although the server's name is Ken, he is not typically the focus of a receipt
- Receipt - the receipt itself is an entity and maybe a good candidate to start with
- Products - this person purchased some donuts and a Box O Joe (whatever that is??). The products could also be a starting place, although might make it difficult to understand starting with this one.
- Customer - sometimes the focus can be placed on the customer, in membership or commercial circumstances, but in this case, customers are likely anonymous a significant amount of the time.

Let us choose the Receipt itself to be the main item here. The choosing of the main entity can be confusing, but realistically, it simply does not matter because as you go through the process of normalization, the other entities will appear anyways. This will become clear through example.

So since we chose the receipt for the main entity we can write the starting DBDL:

```
RECEIPT [receiptID, company_name, purchase_date]
```

Let's look at some other fields we can add from the information on the receipt.

- it is hard to tell, but the 10156 is the receipt ID, this is a good **candidate primary key**.

```
RECEIPT [receiptID, company_name, purchase_date]
```

- Ken was the server, so assumed there are more than one server and therefore there will be a list of servers at some point. There is also likely a PK for servers as we know name is not a good candidate key, let us use employee_ID. It looks like the 18 on the receipt may be Ken's employee number.

```
RECEIPT [receiptID, company_name, purchase_date, employee_id, server_name]
```

- the receipt shows the products being sold, the quantity and the unit price.

```
RECEIPT [receiptID, company_name, purchase_date, employee_id, server_name, product_name, quantity, unit_price]
```

- the Amount, or sub-total, the taxes, a tip if provided and the total charged. It must be made very clear that calculated fields are NEVER included in teh DBDL as they will never be in the final database design. They are derived fields which are not included in a relational database design. However, we need one piece of information in order to calculate all of these, and that is the rate of tax that is charged. We will discuss a little later, but the tax rate is time sensitive and therefore we need to record the tax rate on the date of the receipt, as it may change in the future.

```
RECEIPT [receiptID, company_name, purchase_date, employee_id, server_name, product_name, quantity, unit_price, taxRate]
```

- The payment method (Visa), card #, name on card and the method used to enter card (S-swipe, T-Tap, I-Insert) along with the approval code returned from the payment processing company.

```
RECEIPT [receiptID, company_name, purchase_date, employee_id, server_name, product_name, quantity,
            unit_price, taxRate, payment_method, cCardNum, cCardName, ApprovalCode]
```

- and the receipt shows a space for a signature if required.

```
RECEIPT [receiptID, company_name, purchase_date, employee_id, server_name, product_name, quantity,
            unit_price, taxRate, payment_method, cCardNum, cCardName, ApprovalCode, isSigned]
```

### Repeating Groups

Repeating groups are a group of related fields will have values repeated many times within the rows of the data. This is best described by viewing a sample table using the schema we have so far.

| ReceiptID | company_name | purchase_date | employee_id | server_name | product_name | quantity | unit_price | taxRa |
|---|---|---|---|---|---|---|---|---|
| 11056 | Ma & Pa's Bake Shop | 10/9/2018 | 18 | Ken | Dozen Donuts | 1 | 10.99 | 12% |
| 11056 | Ma & Pa's Bake Shop | 10/9/2018 | 18 | Ken | Box O Joe | 1 | 6.99 | 12% |
| 11057 | Ma & Pa's Bake Shop | 10/9/2018 | 18 | Ken | Choc. Cake | 1 | 15.99 | 12% |
| 11058 | Ma & Pa's Bake Shop | 10/9/2018 | 14 | Beth | Dozen Donuts | 2 | 10.99 | 12% |
| 11058 | Ma & Pa's Bake Shop | 10/9/2018 | 14 | Beth | 6 Bagels | 1 | 6.99 | 12% |

in the above table it is seen that several rows of data repeat themselves as we allow for multiple products to be included on each receipt. Storing repeated data goes against the fundamental principles of database design. Therefore we must eliminate these repeated groups of data.

In UNF we must indicate the presence of these repeated groups of data by using parenthesis in one of 2 ways. Use parenthesis around the groups of repeating data, or around the data that does not repeat. Either method will work, but it is easier to chose the groups that are smaller to start.

#### Completed UNF Solution

```
RECEIPT [receiptID, company_name, purchase_date, employee_id, server_name, ( taxRate, product_name, quantity, unit_price, )
         payment_method, cCardNum, cCardName, ApprovalCode, isSigned]
```

The DBDL shown, uses parenthesis to indicate the fields that do not repeat between rows. All other fields repeat for each Candidate Primary Key: The ReceiptID. There is no need to separate these fields at this point of UNF, but simply indicate them at this point.

## First Normal Form (1NF)

The First Normal Form, or 1NF, is defined as being a UNF solution in addition to the elimination of Non-Atomic Fields. An ***atomic field*** is a field that contains only one value per row. To demonstrate this, let us consider the above table, but follow the principle that the candidate primary key must be unique. Therefore, we will rewrite the table as:

| ReceiptID | company_name | purchase_date | employee_id | server_name | product_name | quantity | unit_price | taxRa |
|---|---|---|---|---|---|---|---|---|
| 11056 | Ma & Pa's Bake Shop | 10/9/2018 | 18 | Ken | Dozen Donuts | 1 | 10.99 | 12% |
| | | | | | Box O Joe | 1 | 6.99 | |
| 11057 | Ma & Pa's Bake Shop | 10/9/2018 | 18 | Ken | Choc. Cake | 1 | 15.99 | 12% |
| | | | | | Dozen Donuts | 2 | 10.99 | |

| ReceiptID | company_name | purchase_date | employee_id | server_name | product_name | quantity | unit_price | taxRa |
|---|---|---|---|---|---|---|---|---|
| 11058 | Ma & Pa's Bake Shop | 10/9/2018 | 14 | Beth | Dozen Donuts | 2 | 10.99 | 12% |
| | | | | | 6 Bagels | 1 | 6.99 | |

Seeing the table rewritten like this, it is clear that the product name, quantity and unit prices have more than one value per primary key. Therefore, these fields must be separated into new entities. This new entity should have a name that combines the source and the new concept. RECEIPT_PRODUCT would be an appropriate name.

```
RECEIPT         [receiptID, company_name, purchase_date, employee_id, server_name, taxRate, payment_method,
                 cCardNum, cCardName, ApprovalCode, isSigned]
RECEIPT_PRODUCT [product_name, quantity, unit_price]
```

However, with two separate entities now, it is obvious that they came from the same UNF solution and therefore must be related. This relationship MUST be maintained. Therefore we need to include a field in the new entity that links this back to the original entity. The appropriate field would be the primary key.

```
RECEIPT         [receiptID, company_name, purchase_date, employee_id, server_name, taxRate, payment_method,
                     cCardNum, cCardName, ApprovalCode, isSigned]
RECEIPT_PRODUCT [product_name, FK receiptID, quantity, unit_price]
```

Lastly at this point, we need to ensure that the new solution is both UNF and 1NF. The UNF property of each entity having an appropriate candidate primary key is required. In our last iteration, the product_name is not a great candidate key. We therefore should add a new key that we know will be needed in the future. To indicate unique products, we will need to add a productID field. Therefore, our final 1NF solution will become:

**Completed 1NF Solution**

```
RECEIPT         [receiptID, company_name, purchase_date, employee_id, server_name, taxRate, payment_method,
                    cCardNum, cCardName, ApprovalCode, isSigned]
RECEIPT_PRODUCT [productID, FK receiptID, product_name, quantity, unit_price]
```

## Time Sensitive Attributes

We will often run into some attributes that can change over time and can affect calculations if they change. The list_price of products is one of these type of attributes, as is tax rate and other surcharge amounts.

We can all image a situation where we have the above receipts and we would need to also have a PRODUCTS entity to store the list of products available and this entity would include the current list price for each product.

```
PRODUCTS    [productSKU, product_name, list_price, on_hand]
```

In this case it is important to note that the list_price would be the current price if someone bought that product today. But what if someone bought the product 7 days ago, and the product is now on sale for less money.The list_price in the products table would change. But if this list_price was used to calculate the resiept totals for the purchase 7 days ago, the totals would be different than they were 7 days ago, because the producers list price changed. Therefore, it is critical that an additional field be created to store the price it was on the date of the receipt, and this value is used for calculations.

```
RECEIPT         [receiptID, company_name, purchase_date, employee_id, server_name, taxRate, payment_method,
                    cCardNum, cCardName, ApprovalCode, isSigned]
RECEIPT_PRODUCT [productID, FK receiptID, product_name, quantity, unit_price]
PRODUCTS        [productSKU, product_name, list_price, on_hand]
```

in this example, the product price was split into 2 fields. The ***unit_price*** in the RECEIPT_PRODUCTS entity is the price the product was on the date of the sale and the ***list_price*** is the price of the product if it was purchased right now.

> Time sensitive attributes often need to be split into 2 different fields to maintain historic values.

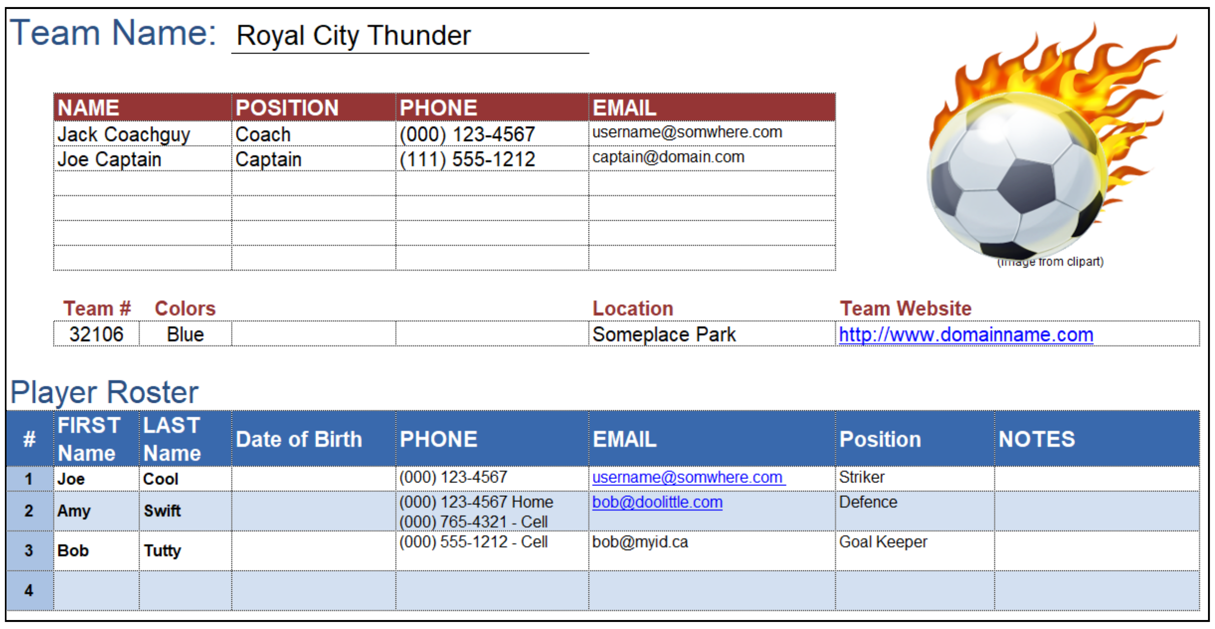
## Case Study

Let us do one more example that can demonstrate the process:

### Soccer League

A local recreational soccer league is starting up and needs a database to maintain the league. This database will include player data, team data in addition the league schedule, statistics and standings.

The following form will be used to determine the database design through normalization.



Team Name: Royal City Thunder

| NAME | POSITION | PHONE | EMAIL |
|---|---|---|---|
| Jack Coachguy | Coach | (000) 123-4567 | username@somwhere.com |
| Joe Captain | Captain | (111) 555-1212 | captain@domain.com |
| | | | |
| | | | |
| | | | |
| | | | |

| Team # | Colors | | | Location | Team Website |
|---|---|---|---|---|---|
| 32106 | Blue | | | Someplace Park | http://www.domainname.com |

Player Roster

| # | FIRST Name | LAST Name | Date of Birth | PHONE | EMAIL | Position | NOTES |
|---|---|---|---|---|---|---|---|
| 1 | Joe | Cool | | (000) 123-4567 | username@somwhere.com | Striker | |
| 2 | Amy | Swift | | (000) 123-4567 Home (000) 765-4321 - Cell | bob@doolittle.com | Defence | |
| 3 | Bob | Tutty | | (000) 555-1212 - Cell | bob@myid.ca | Goal Keeper | |
| 4 | | | | | | | |

## UNF

Like before we will start the UNF process by determining the main entity of the form. In this case we will use the team itself.

```
TEAMS  []
```

Next we determine an appropriate candidate key for the entity of choice. The team # would be unique and a good candidate key.

```
TEAMS  [Team#, teamName]
```

Now we go through the process of adding all the fields we can obtain from the form, ensuring not to include calculated or derived fields.

Coach and Captain Info:

```
TEAMS  [Team#, teamName, CoachName, CoachPhone, CoachEmail, CaptainName, CaptainPhone, CaptainEmail]
```

The team specific information:

```
TEAMS  [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
        CaptainName, CaptainPhone, CaptainEmail]
```

The Player Information:

```
TEAMS  [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
        CaptainName, CaptainPhone, CaptainEmail, pNum, PfName, PlName, pDOB, pPhone, pPhoneType, pEmail, pPosition, pNotes]
```

The last step is to indicate the repeated groups of data by either choosing the items that are repeating, or the items that differ and surrounding them with parenthesis. In this case, all the fields are repeating data for different players, therefore let us surround the player information in brackets.

### Final UNF Solution

```
TEAMS  [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
    CaptainName, CaptainPhone, CaptainEmail, ( pNum, PfName, PlName, pDOB, pPhone, pPhoneType, pEmail, pPosition, pNotes ) ]
```

## 1NF

The first step in the 1NF is to eliminate the repeating groups. This is done by creating a new entity, moving the fields there and creating a connecting Foreign Key.

First let's move the fields in parenthesis into a new entity called TEAM_PLAYERS.

```
TEAMS           [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
                  CaptainName, CaptainPhone, CaptainEmail]
TEAM_PLAYERS    [pNum, PfName, PlName, pDOB, pPhone, pPhoneType, pEmail, pPosition, pNotes]
```

In order to link the entities together, we need to bring a FK forward. Team# will be the field of choice.

```
TEAMS           [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
                      CaptainName, CaptainPhone, CaptainEmail]
TEAM_PLAYERS    [FK team#, pNum, PfName, PlName, pDOB, pPhone, pPhoneType, pEmail, pPosition, pNotes]
```

Next we need to determine a candidate primary key. This will differ based on a business rule. Can a player play on more than one team. If yes, then players can be repeated as can team numbers. If players can not play on more than one team, then the team id can be stored in the player table as a foreign key. In this case scenario, we do not know the answer to this question, so we will choose the more flexible answer to remove restrictions in the future. This is to assume they can play on more than one team.

Therefore, the candidate primary key will be a composite key made up of two fields in which the combination of values can not be repeated. This will be the combination of team and player.

```
TEAMS           [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
                        CaptainName, CaptainPhone, CaptainEmail]
TEAM_PLAYERS    [FK team#, pNum, PfName, PlName, pDOB, pPhone, pPhoneType, pEmail, pPosition, pNotes]
```

We need to ensure there are no repeating groups of data remaining in either entity. By careful observation we see that phone number creates a non-atomic field that can be considered a repeating group, when viewing it from a different perspective. We will choose to look at it from a non-atomic field perspective at this point.

Next we need to ensure that all fields are atomic: i.e. no one field has multiple values per primary key. Looking at the sample, it is possible for players to have more than one phone number. Therefore, the player phone number field must be separated.

```
TEAMS           [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
                        CaptainName, CaptainPhone, CaptainEmail]
TEAM_PLAYERS    [FK team#, pNum, PfName, PlName, pDOB, pEmail, pPosition, pNotes]
PLAYER_PHONES   [pNum, pPhone, pPhoneType]
```

We seem to have an issue with pNum as the relationship between TEAM_PLAYERS and PLAYER_PHONES is through the pNum fields. But it is not an independent PK in either case, therefore, we need to create an entity that has that as a single field PK.

外键要引用独立存在的主键

```
TEAMS           [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
                        CaptainName, CaptainPhone, CaptainEmail]
TEAM_PLAYERS    [FK team#, pNum, pPosition, pNotes]
PLAYERS         [pNum, PfName, PlName, pDOB, pEmail]
PLAYER_PHONES   [FK pNum, pPhone, pPhoneType]
```

The tricky part of the above, is knowing which fields go in which entity. For now we will take our best guess. I know that sounds vague, but generally move fields that relate directly to the entity (i.e. player name relates to the PLAYER, not the TEAM_PLAYERS and the player position depends on both the player and the team. Ultimately, if you don't get it quite right, that is okay at this point. When solving 2NF and 3NF, these mistakes here will work themselves out anyways. The key thing to ensure correct at this point is:

- no repeating groups
- all entities have an indicated PK or CK
- all fields are atomic
- that all tables are related through PK-FK relationships

## Final 1NF Solution

```
TEAMS           [Team#, teamName, TeamColor, HomeLocation, TeamURL, CoachName, CoachPhone, CoachEmail,
                        CaptainName, CaptainPhone, CaptainEmail]
TEAM_PLAYERS    [FK team#, pNum, pPosition, pNotes]
PLAYERS         [pNum, PfName, PlName, pDOB, pEmail]
PLAYER_PHONES   [FK pNum, pPhone, pPhoneType]
```