Database Primer

Contents

[1 Introduction 1](#_Toc195096783)

[2 What is a Database? 1](#_Toc195096784)

[3 How are Database Tables Related? 2](#_Toc195096785)

[4 Referential Integrity 3](#_Toc195096786)

[5 Database Programming – General Introduction 4](#_Toc195096787)

[6 Notes about the Databases we will use 4](#_Toc195096788)

[Appendix 1 n/a 6](#_Toc195096789)

# Introduction

An important aspect of almost every type of software is *data persistence*. Four important aspects of database technology are how to:

1. Write code that uses a database – This is the main focus of this Topic. This is discussed in the following two documents: SQL Primer and Database Programming Primer, found in the code download.
2. Design a database – We will not actually design databases, but we do need to understand enough so that we can write code to use a database. This document is our introduction to the design/organization of a database.
3. Build a database – We might create a database, and/or I might give you one
4. Administer a database – We will not consider this in this course.

For homework assignments and tests, I will give you databases, explain their design and then you will write code that uses them.

# What is a Database?

A database is a set of (usually) related *tables* that store information about something of interest. For example, below we have the *Players Table* and the *Teams Table*.

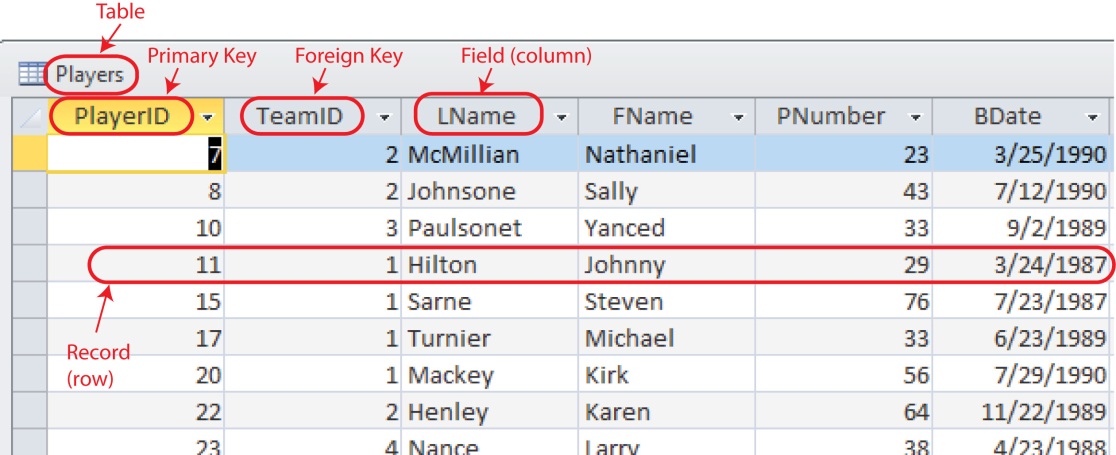


Figure 1 – Players Table

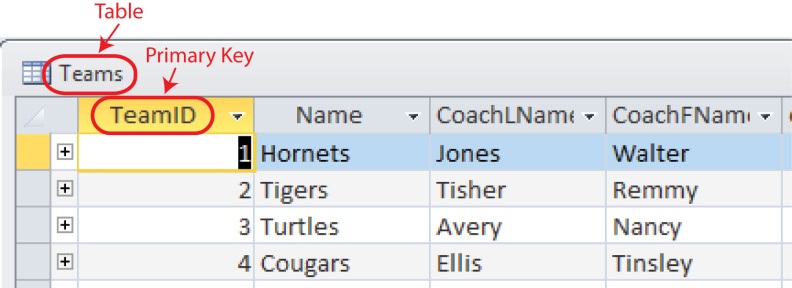


Figure 2 – Teams Table

As shown in Figure 1 above, a database *table* is composed of *rows* and *columns.*

* Each *column* (also called a *field*) has a name and data type.
* Each *row* (also called a *record*) contains the data values for one particular entity.
* All tables will have a *primary key* whose value is unique across all records in that table. The primary key is composed of one or more fields. In code, we typically use the primary key to retrieve, change, or delete a record. The primary key is supplied by the database engine when a record is added.
* Some tables will have a *foreign key*. These will be discussed shortly.

# How are Database Tables Related?

The example below shows how database tables can be *related*. Notice

1. *TeamID* is the primary key in the *Teams* table, thus each team has a unique ID.
2. *TeamID* is a *foreign key* in the *Players* table. A *foreign key* is a link to a field (column) in another table. The table and field it is linked to is specified when the database is created. Thus, we can use the *TeamID* of a *Player* to find the *Team* that they are on. This is called a *primary key – foreign key relationship.* A foreign key must have a value that exists in the associated table, or *NULL* to ensure data consistency.
3. Sometimes we call a table like *Teams* a *lookup table.*
4. We use this type of relationship so that we don’t repeat data. If we didn’t have a lookup table for the team, then we would need all *Team* fields in the *Players* table and as a result we would have repeated data. Changing a team name would be cumbersome.

In the example, we see that player 8 (Sally Johnsone) is on team 2 which is the Tigers, coached by Remmy Tisher. **This example shows using a *foreign key* to find the one associated row in the related table.**

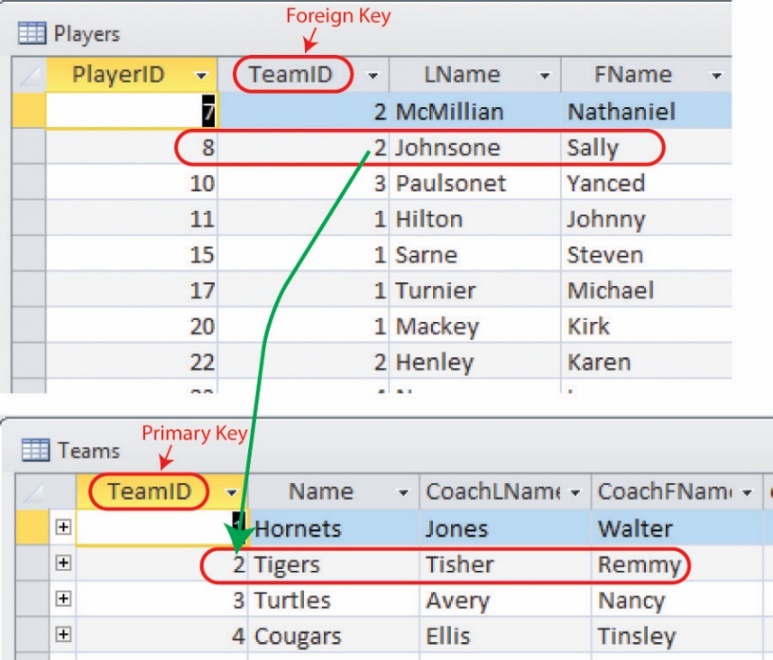


Figure 3 –Identifying the Team a Player is on.

Similarly, we can match the *TeamID* in the opposite direction: we can find the players on a particular team. **This example shows using a *primary key* in a table to find all the associated rows in the related table.**

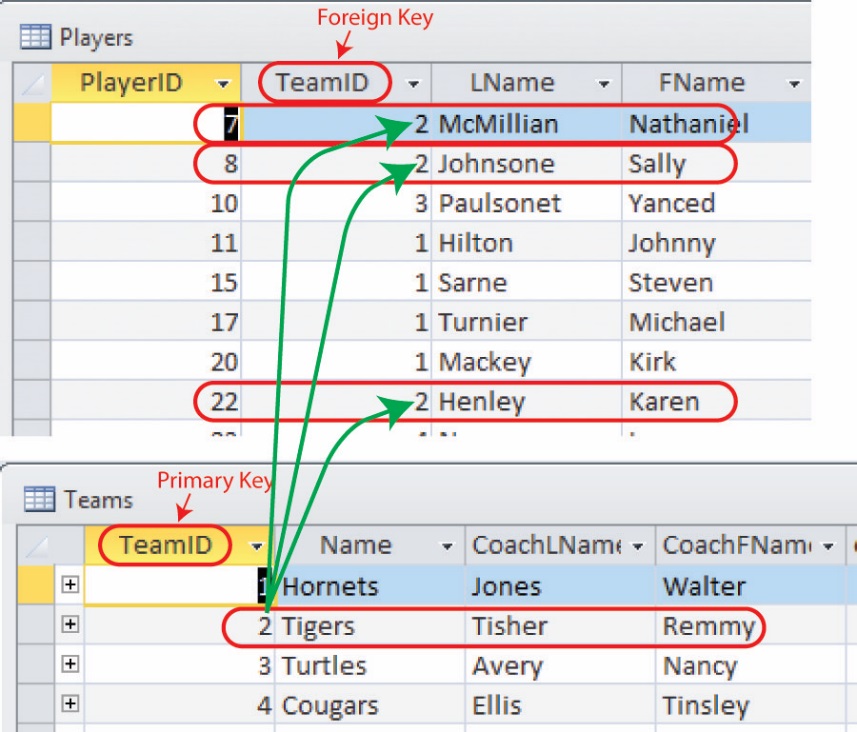


Figure 4 –Identifying the Players on a Team

An [Entity Relationship Diagram](https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model) is used to represent the relationships between tables. This is also called a [*database schema*](https://en.wikipedia.org/wiki/Database_schema). MS Access’s representation for the *Player* and *Team* tables is shown below. It is very similar to a class diagram. Thus, we can say that there a one-to-many relationship between *Teams* and *Players*: each team has many players and each player has one team. For assignments and tests you will usually be given a diagram similar to Figure 5 so that you can identify the primary key and any foreign keys.

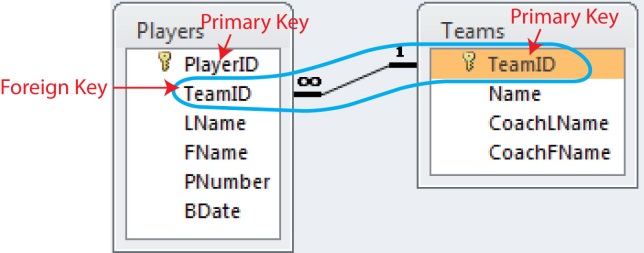


Figure 5 – Relationship between the *Players* and *Teams* Tables

# Referential Integrity

Many databases (including MS SQL Server and Access) *enforce referential integrity.* This means that any foreign key value must be valid. In other words, when a foreign key value is supplied for a record in one table, the value must map to an existing primary key value in another table. For example, in Figure 3, a Player cannot be assigned to a non-existent Team.

Implications of referential integrity:

* You can only delete a Team if there are no Players assigned to it. For example, in Figure 3, we cannot delete Team 2 (Tigers) because there are Players assigned to it (McMillian, Johnsone, *etc*). Many databases can be set to do a *cascading delete.* In this example, deleting a team would also delete all the players.
* You can always delete a Player because the primary key does not appear as a foreign key in any other table (we only have 2 though).

We illustrate this further in the last section.

# Database Programming – General Introduction

Many times, the primary key (and foreign keys) are not meaningful to the user as in the Players table and Teams tables above. In such a situation, we do not typically show these values to the user. However, as programmers we use them to retrieve related information and to make changes to data. In other cases, for instance using SSN as a primary key, the value is meaningful to the user and would be displayed.

Example – We want a drop-down list to display Player names and when one is selected by the user we want to display the other information about the player (jersey number, birth date, and perhaps team name and coach).

Solution – We define the drop-down list’s *Text* property as the player's name (perhaps a concatenation of *LName* and *FName*). In the drop-down list’s *Value* property we store the player's unique ID, the primary key (*PlayerID*). When the user selects a player, say "McMillian", our code will access the Value (*PlayerID*=2) and use it (in a SQL statement) to search in the Players table for this particular player, retrieve the associated data and display it.

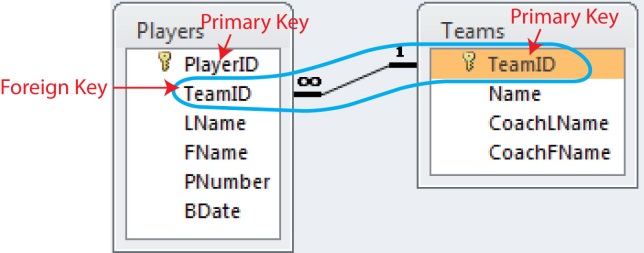
Example – We want a drop-down list to display Teams and when a team is selected by the user we want to display all the players on that team. In a general sense, this is referred to as *filtering*. We are filtering the players, based on the team that was selected.

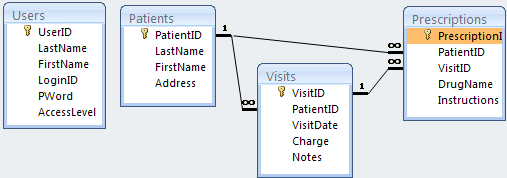
Solution – We define the drop-down list’s *Text* property as the team name and its *Value* property to contain the primary key (*TeamID*). When the user selects a team, our code will access the *Value* (*TeamID*) and use it (in a SQL statement) to search for all occurrences of this value in the foreign key field (also *TeamID*) in the Players table.

# Notes about the Databases we will use

This is a brief introduction to the 3 databases we will use. The first one, *Players* is used in examples in Lab and notes. The other two will be used in homework assignments.

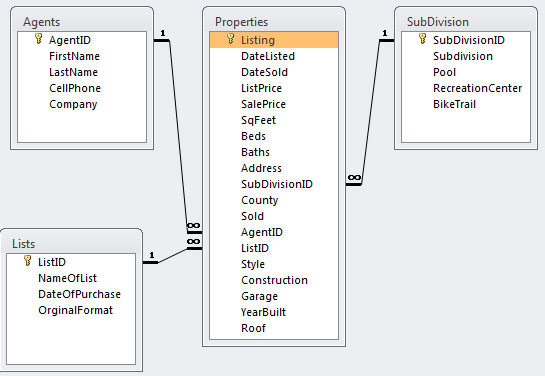
1. **Players database (*players.mdb*)** – We have discussed the Players and Teams tables in Figure 5 above (and below).



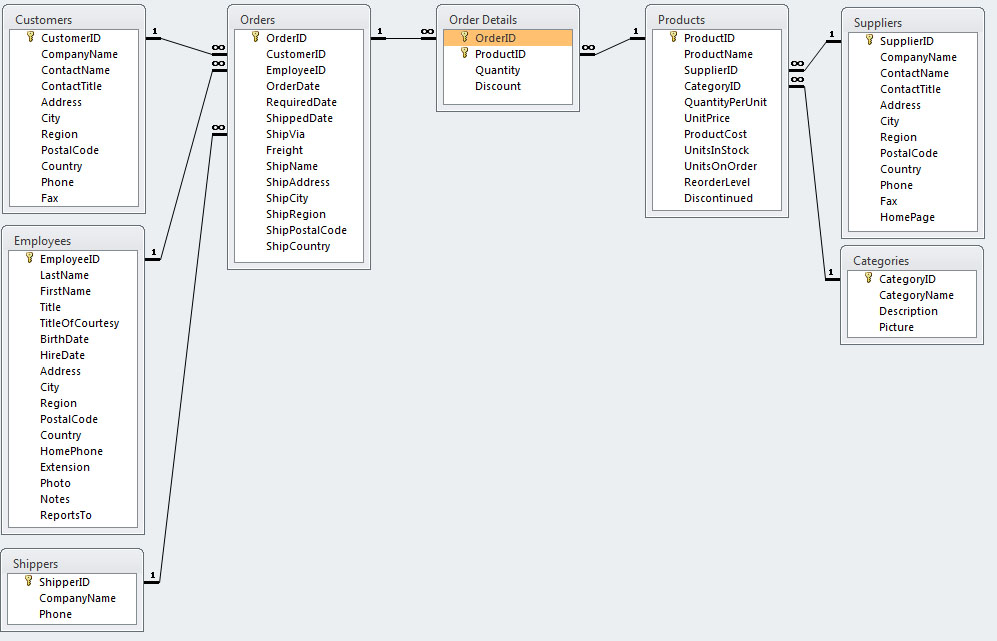
1. **Patients database (*patients.mdb*)** – Another set of tables in your database models a doctor's office.
2. The Users table is not associated (in a formal sense) with any of the other tables. This table represents the Doctors and Nurses who are allowed to use the system. We see fields for the user’s name as well as their *LoginID* and *Password* as well as *AccessLevel* (more on this below). We probably won’t use this table.
3. The *Patients* table shows that each patient can have many visits to the doctor and many prescriptions.
4. The *Visits* table shows that each visit can result in many prescriptions that are written for that visit and each visit corresponds to one patient
5. The *Prescriptions* table shows that each prescription corresponds to one patient and one particular visit to the doctor. Notice that the *PatientID* foreign key in the *Prescriptions* table is redundant; it is not necessary as we can get all a patient’s prescriptions, by getting all their visits and then using each visit to get the prescriptions associated with that visit. Thus, this foreign key is a convenience.

The *AccessLevel* field in the *Users* table is used to control access to information in the other tables, programmatically. For instance, a doctor may have *AccessLevel=1* and this may mean that he or she can add/edit/delete any information in the database. A nurse may have *AccessLevel=2* which may mean that they can add or edit Patients or Visits, but can only view Prescriptions.

1. **Property database (*property.mdb*)** – Some more related tables in your database:



1. **(Optional, we will not use this database, Spring 2020) Traders database (*traders.mdb*)** Notice that the “Order Details” table has a *composite primary key* which is composed of two fields (OrderID and ProductID). *Orders* and *Products* are in a many-to-many relationship, i.e. each order has many products, and each product can be in many orders. Many-to-many relationships are not implemented in a database. Instead, we introduce a table, *Order Details* so that we have two 1-many relationships.



Appendix

1. n/a