RELATIONAL DATABASE PRINCIPLES

DATABASE DEFINITION

In a loose sense, almost anything that stores data can be called a database. So a spreadsheet, or a simple text file, or the cards in a Rolodex, or even a handwritten list, can be called a database. However, we are concerned here with computer software that manages the addition, modification, deletion and retrieval of data from specially formatted computer files. Such software is often called a **R**elational **D**ata**b**ase **M**anagement **S**ystem (RDBMS) and that will be the subject of the remainder of this tutorial.

RELATIONAL DATABASE HISTORY

The structure of a relational database is quite specific. **Data** is contained in one or more t**ables** (the purists call them *relations*). A table consists of **rows** (also known as *tuples* or *records*), each of which contains **columns** (also called *attributes* or *fields*). Within any one table, all rows must contain the same columns; that is, every table has a uniform structure; if *any* row contains a Zipcode column, *every* row will contain a Zipcode column.

In the late 1960's, a mathematician and computer scientist at IBM, Dr. E. F. (Ted) Codd, developed the principles of what we now call ***relational databases***, based on mathematical set theory. He constructed a consistent logical framework and a sort of calculus that would insure the integrity of stored data. In other words, he effectively said, ***If you structure data in accordance with these rules, and if you operate on the data within these constraints, your data will be guaranteed to remain consistent and certain operations will always work***. That's not to say that violating these rules *necessarily* prevents you from getting correct results in a particular instance, it's more like saying, ***all bets are off***.

DESIGNING A DATABASE

It is tempting for many people who have used spreadsheet software like Microsoft Excel to think of a relational database as just a more powerful version of a spreadsheet, or at least their mindset is to think in terms of "cells" and formulas. This kind of thinking will prevent these people from getting the results they want, because the underlying concept of relational databases is utterly different.

The database management approach is to organize *raw data* in accordance with very strict rules, and to use concepts like *queries* and *views* to retrieve, manipulate and combine the data as desired. *Calculations based on the raw data* are performed *at the time they are needed* for display or printing, **not** stored in the database, as is done in spreadsheets. Much of this retrieval and manipulation is accomplished with a standard *language* called **SQL** (which originally stood for **S**tructured **Q**uery **L**anguage, but is now considered just the name of the language). Nearly all RDBMSs use SQL, but there are small variations in syntax from one to another.

Dr. Codd's *relational database model* made it possible to handle extremely large amounts of data very efficiently and reliably. Without it, we would not have the airline reservation systems, search engines, online businesses, or any of the other database applications that we now take for granted.

So let's begin to get specific. What kinds of rules should be applied, and how should a database project be started?

Some of the fundamental rules are:

1. ***Every row in a table must be unique***. This is a requirement because if the relational calculus is to be reliable, we must be able to identify a specific row, for example, to be deleted. We cannot rely on the sequence of the rows; indeed, relational database theory specifies that *there is no defined order of data in a table*. This usually requires that each table has a **primary key** field or combination of fields that can never be duplicated within that table. The database engine itself normally guarantees that this state will be maintained; for example, if you try to enter a row of data with a primary key value that duplicates an existing value, the database engine will present an error message, and will not add the new data.

2. ***Data should be "atomic".*** That means that data should be stored in its most basic, indivisible form; for example, avoid storing an address as a single data element, with *street number*, *street name*, *apartment number*, *city*, *state* and *zip code*. Each of these are separate pieces of data and should be stored in separate columns.

3. ***Data should be single valued***. That means that you should not have a single field that contains "children's names," for example. If you need to join children's names to their parents' records, do it with relationships between tables, not by entering multiple names into one field in the parents' record.

4. ***Avoid data redundancy***. Ideally, a piece of data should be stored in only one place, so that its value can never be ambiguous. When a value is needed, refer to the one place where it is stored.

5. ***Avoid storing*** "***dependent da*****ta**." Only *raw* data should be stored in tables. If a value is dependent on (can be calculated from) other data, the calculation should be performed by the database application at the time the results are needed.

Experienced database developers sometimes "bend" some of these rules for a particular application, usually for reasons related to performance. But when they do so, it should be with a full understanding of what consequences may result from their departure from the theoretical rules expressed by E. F. Codd and Chris Date. *Beginners who break these rules do so at their own peril*.

A common mistake made by beginners is to start with visualizing a data entry screen, or a report format. While those are important parts of an application that will need to be designed later, it is counter productive to *begin* from those ideas. Instead, it is the data, itself, that should drive the design.

Every database application can be considered to be a ***model*** of some segment of the real world. No matter what your project may be, the most efficient design process is to begin by clearly identifying what is to be included in your data model. This should be done ***in writing*** to sharpen your model and permit your returning to the model description later, if you need to review or modify it.

This data model should clearly identify the **entities** that will be part of your data model and their **relationships** to each other. It could be, for example, that you want to model the assignment of students to classes, or the status of sales orders that are received by your business. By pausing to really define

the extent and scope of your database, you will find it easier to carry out the next step: identifying the ***entities*** in your model. An **entity** is a class of "things", such as *persons, objects, events, documents, transactions,* etc. Generally, every *entity* that you identify will become a *table* in your **schema**, which is what we call the description of your database structure—its tables and columns and relationships.

So, long before you even boot up your computer, you should be writing down on a piece of paper such thoughts as:

Model: schedules of classes for students, with grades.

Entities: Students
Classes
Teachers ?

Relationships: a student can attend many classes,
a class may have many students,
a class must have one and only one teacher.

Often you will not know all the details at first, so it is a good idea to include items with question marks, as I did Teachers in this example. That will remind you to review such questions, as your project is being refined.

Having done this, you can then begin to establish the *columns* required for each table. Here's where the disciplined approach really matters. For each table, representing an entity, you should specify all the ***attributes*** or characteristics of that entity that you intend to store in the database. Initially, the only consideration should be that it is clearly an attribute, or feature, of that entity. Later, we will take a second pass as we *normalize* each table, which we will explain further in this tutorial.

ENTITIES AND ATTRIBUTES

To illustrate this process, consider the entity *Students*. Each row in the table will represent a student. What attributes does any one student have? Obvious ones will include First Name, Middle Name, Last Name, Home Address, City, State, Zip, Home Phone, Mobile Phone, Email Address, maybe Date of Birth, Student Number, etc. You might have just written down Name, Address, etc., but the very first consideration when we begin to normalize the table will be whether every column is "*atomic*". That means that a column should consist of *data that cannot be divided into smaller parts*; so we never want to store the street address and the city, state and zipcode in one column or field! Likewise with the name. An inexperienced person might think that *grades* might be attributes of a *student*, but they are not, for several reasons. First of all, they don't define a student, like the student's name or address. They are merely values *associated with* a student who has completed certain *classes*. It makes no sense to say that the student can be described, in part, by "B, A, C+, B-, and A-". Grades are meaningful only in connection with **a** student and **a** class. We will see later how this data is stored and related to the student and to the class.

Next in our example, consider the entity *Classes*. Each row in the table will represent a class. What attributes does any one class have? Certainly that will include the *Catalog Number*, *Teacher* and *Room*

Number. You might think that the Course Name and Course Description, Year and Semester, and maybe even the list of students would be included, but you'd be wrong. Assuming that this database is going to be useful for more than one class conducted in one semester, this analysis should lead you to recognize that *you need another entity*; a Class is just an instance of a *Course*, which might be offered at several different time slots with different teachers, and might be offered semester after semester. So now you should see that ***Courses** is another entity*, one that we failed to anticipate when we first wrote down the list of entities. You should go back and add it now! *Course Name* and *Description* are attributes of a Course, not of a Class. Thus we need another table. We have already recognized that *Students* is an entity of its own. We will soon see how we link these entities together.

NORMALIZATION

Normalization is the process of refining the structure of columns within a table, in accordance with specific rules devised by E. F. Codd and published in the seminal textbook, *An Introduction To Database Systems* by Chris Date (now in its 8th edition!). The purpose of normalization is to reduce known data structure issues by applying rules that will methodically eliminate common problems. The rules are expressed as definitions of several successive ***forms*** in which data tables may be defined. These are known as ***First Normal Form* (1NF), *Second Normal Form* (2NF), *Third Normal Form* (3NF)** and ***Boyce-Codd Normal Form* (BCNF)**. There are actually two more, **4NF** and **5NF**, but these are not often used. There is controversy within the developers community as to how far along the normalization forms it is necessary to go, but probably most developers would recommend going at least to **2NF**, and many would go further.

So what do these normal forms require? To satisfy the condition of being in **1NF**, a table must contain *scalar values* only and must *not* contain any duplicate rows. Scalar values means essentially the same thing as "atomic" values; that is, a particular column value in a particular row of a table can have only one piece of data, not a series of values like, say, an array. At first, such a requirement may seem restrictive, but the Codd Relational Model is built on such rules. There are common techniques for handling the practical situations where one might want, for example, to store an array of values, but if the normal form rules are violated, the mathematics on which the database engine is built cannot be guaranteed to produce consistent results.

So, let's take on **2NF**: a table is in **2NF** if and only if it is already in **1NF** *and* every nonkey attribute is irreducibly dependent on the primary key. Yes, that's a little daunting, but all it means is that all the columns are scalar, or "atomic," *and* all columns except key fields are really dependent *on* the primary key, or to say it another way, each column provides a fact about the primary key, like in a customer table, with a customer ID number as the primary key, every other column that is not a foreign key provides some description of that customer, such as the address or credit rating. No column should provide data, for example, about the customer's *activity*, such as the customer's latest order number, since that information depends on the customer's ID *and* on other facts, such as when orders were placed.

Then, to be in **3NF**, a table must be in **2NF**, *and* all columns depend *directly* on the primary key. Tables violate the **3NF** if one column depends at least partly on another column, which in turn depends on the primary key (a transitive dependency).

That's already more than you'll probably need to know about normalization, at least for a long time! The most understandable tutorial on normalization that I have found is at this URL:

http://www.phlonx.com/resources/nf3/

The point I'd like to make is that there's a lot of heavy duty logic behind the way you should assign columns to tables. It's ***not*** a hand-waving, fuzzy kind of "*this is the way I'd do it*" or "*this looks right to me*" kind of thing. Or certainly it ***should not be***!

PRIMARY KEYS

Before going further, let us consider the primary key that every relational table must have. If an entity has a natural attribute that is *unique* to every instance of that entity, it may be used as the primary key field, but there are many pitfalls to be aware of. If the entity is a person, for example, a primary key that has been often used is the person's **S**ocial **S**ecurity **N**umber.

There can be problems with this, though. Some people don't have SSN's (foreign visitors, babies), and (believe it or not!) some people have more than one SSN! There have also been a small number of duplicate SSN's assigned mistakenly by the Social Security Administration! Furthermore, you *may not know* a person's SSN at the time you want to enter their record in the database, and you simply cannot enter a new record with a blank primary key.

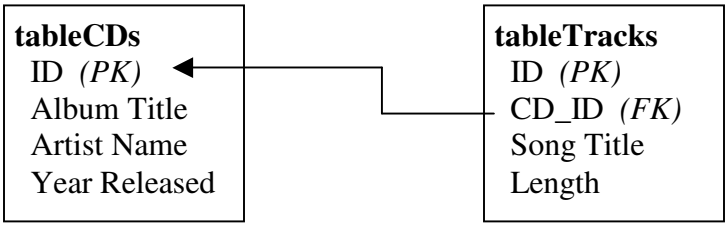
A primary key may consist of a *combination of more than one attribute*, which is known as a *composite primary key*. So you might use the combination of the last 4 digits of a bank account number, the full check number, and the date of the check, as the primary key for a table containing check transactions, which would be highly unlikely to produce duplicate rows.

There are formal ways to select the primary key for a table, involving a concept called *candidate keys*, but for this introductory tutorial, it's unnecessary to belabor the point further.

Sometimes (often) it will be simpler to assign an *independent field* as the primary key, which bears no relation to any natural attribute of the entity. An advantage of doing this is that nearly every database engine provides a data type that will automatically assign and manage numbers, to guarantee that every new row will have a unique identifier. By simply defining such a field and declaring it to be ***Autonumber*** or ***Auto-increment*** type, the database will manage the primary key for you. In this case, the database user should ordinarily never be concerned with what the value is, and usually never even has visibility of what the values are. The values are used by the database engine itself and by the application programs that manipulate the data. Indeed, allowing database users to change the value of a primary key is a sure road to database corruption.

FOREIGN KEYS

In order to relate data in a row of one table with data that may be stored in one or more rows of another table, one table must have a *primary key field* and the other table must have a *foreign key field* of the same data type. A simple example would be music CD's and the tracks of music on them. Since a CD usually has many tracks (songs) recorded on it, this requires a *one-to-many* relationship between the two tables, *tableCDs* and *tableTracks*. The usual way to depict such a relationship is an ***Entity-Relationship Diagram***, or **ERD**, like this:



Each table has its own primary key field (they do not need to have the same field name), which insures that every row in that table can be uniquely identified. The table on the "many" side of the relationship has a *foreign key field* that links each track to the CD it's on. Most database engines provide the ability to *enforce referential integrity*, as Microsoft Access calls it, by automatically "***cascading*** ***delete*** and ***update*** transactions to keep the two tables consistent. That means that if you try to delete a row in *tableCDs* which has "child" records in *tableTracks* (that is, there are rows in *tableTracks* that have the foreign key value of the *tableCDs* row that is going to be deleted), the database engine will ask the user if they wish to also delete these dependent rows in *tableTracks*. Unless the user agrees, the record will not be deleted from *tableCDs*. Similarly, if the user attempts to update (edit) the ID field in *tableCDs*, the database engine will ask the user if they wish to have all the dependent rows in *tableTracks* updated to correspond to the new value. Otherwise, the data in the two tables would be inconsistent, with "orphan" records whose foreign key values don't match a row in the other table.

This *primary key / foreign key* relationship is such a fundamental part of relational databases that I will present the same concept in another way, by showing sample data records in the two tables I previously described:

tableCDs:

ID	Album Title	Artist Name	Year Released
1625	Courage	Randy Gavin	2004
1626	Ain't It Hot	The Wheels	2004
1627	Changes A-Comin'	Bunny Perkins	2005

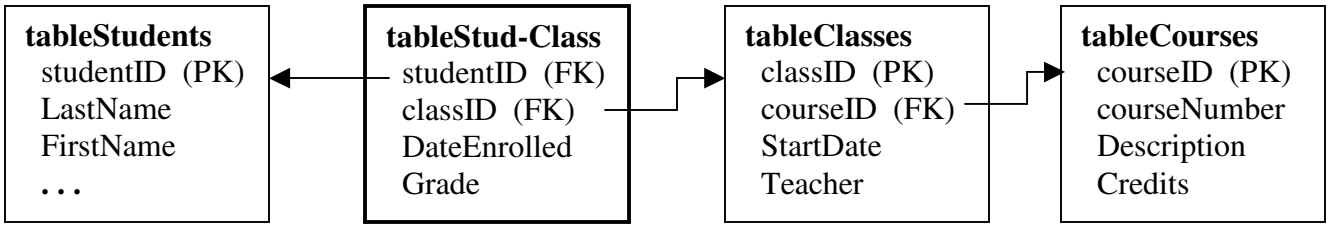
tableTracks:

ID	CD_ID	Song Title	Length
72013	1627	Why Am I Here?	4:15
72014	1627	No, You Can't	3:55
72015	1626	Solitude	6:20
72016	1627	You Got That Right!	4:45

You can see that each row in the second table has a value in the foreign key field (CD_ID) that matches a value in the primary key field (ID) of the first table, making it possible to determine what album each song is on.

The above example is called a **one-to-many** relationship. *One* CD can have *many* tracks. In tableCDs, a value for the (primary key) ID column can appear *only once;* that is, the ID column is *unique*. In tableTracks, there can be many duplicate values in the (foreign key) CD_ID column.

Another kind of relationship is a **many-to-many** relationship. Going back to our Students, Classes and Courses example, a *Student* may attend *many* classes **and** a *Class* will have *many* students. Such relationships are common, but to represent them in a relational database, ***you need a third table*** to link the primary keys of the other two tables. For Students and Classes, it would look something like this:



The table with 2 foreign keys in it is the *third table* and it represents the *occurrence* of **a** student enrolled in **a** course. This is where the date and grade are stored.

This is how relational databases can be so flexible. There is no limit to how many classes a student may take, nor how many students can enroll in a class.

RDBMS SCHEMA

We call the description of the *tables* and *fields* of a database and their *relationships*, a **schema**. It may be presented as a list of fields associated with each table, or as an *ER Diagram*, or in the case of some large and complex databases, a formal structure embedded in the database itself. For further information on RDBMS schemas, I recommend starting with Wikipedia's page at
http://en.wikipedia.org/wiki/Database_schema

A more detailed (and complex) tutorial can be found at
http://www.databaseanswers.org/tutorial4_db_schema/index.htm