# Appendix B Database Design

## B.1 Database Design and Definition

Throughout the SQL chapter we connected to and queried the IMDB database. This database was set up by IMDB and available for us to use. But what if you want to set up your own web application needing an entirely different database – how do you create a database from scratch for your own website? This is what we are going to explore in this appendix.

**database design**

The process of deciding the structure of a database.

Designing an accurate and consistent relational database is probably the most difficult and important part of using a relational database management system. If you design a database poorly your application can be slow and prone to replicated data and many errors. If your website is slow because of the database or falls over because your database contains bad data, your users will get easily frustrated and stop using your web site. Luckily the RDBMS and relational database design are based on sound theory and there are many books that cover the topic quite thoroughly. This is simply an introduction to key concepts. For a more thorough and complete coverage of database design, please see the reference list at the end of this appendix.

The first step toward creating a database from scratch is to decide what information you actually need to store in your database and how different pieces of information relate to one another. Once you have done that, you then think about how to logically structure that information into tables and how those tables should relate. In other words, what the tables are, what the columns are, and what the keys are. Next you think about how to physically create the structure on your RDBMS or what data types you'll be using. This entire process is called *database design*.

**data definition language (DDL)**

SQL statements used to create, delete, and edit the structure (i.e. tables, and columns) of a database.

After deciding what data to store and how to structure that data in your database, you use the SQL *data definition language* to create the database and create the tables in the database.

### B.1.1 Relational Database Design

Our two main goals of database design will be to:

1. accurately represent information, and
2. design a database structure that avoids repetitive data and thus data inconsistency.

An additional goal of database designers is to ensure that the database can run queries FAST! We will leave this as an advanced topic and refer you again to the references provided at the end of the appendix to learn how to tune and optimize your database.

Our final product of our design will be a *database schema*. A relational database schema is a description of the tables in a database and how they relate to one another. Specifically, a final database schema will contain:

**database schema**

The description of the tables in a database along with the relationships between them.

- the name of all tables in a database
- the columns belonging to each table
- the data type of each column, and whether the column allows `NULL` values
- the primary keys of each table

- any foreign keys referencing other tables

To get a database schema, there are three phases of database design:

1. Conceptual design – the process of determining what information should be stored in the database and what the relationships and dependencies are between the types of information to be stored.
2. Logical design – the process of mapping the information we have identified in the first step to tables, columns, and primary and foreign keys and then checking to see how we are doing in avoiding redundant data and refining our tables from there.
3. Physical design – the process of specifying how the database will be physically created in the RDBMS by choosing appropriate data types for our columns.

## The `world` Database

We ground our discussion of database design in the following scenario. We have been hired by an international non-profit organization, Ayuda, to design a database schema that stores current information about different countries in the world. For each country, Ayuda wants to store the name of the country in English, the standard, unique three-letter country code (e.g., 'USA' for United States of America), as well as what continent the country is in. In order to make decisions on which countries to dedicate aid, Ayuda wants to be able to know if the country is considered developed or undeveloped. Common indicators for how developed a country is are population, average income, and life expectancy. Ayuda is able to provide each country's gross national product and population and those two pieces of data together can be used to calculate average income. Ayuda is additionally interested in storing information about each country's government such as the type of government and the leader of the country.

In order to train its volunteers appropriately, Ayuda needs to know the languages spoken in each country. Ayuda also needs to know whether or not the language is official and what percentage of the population is speaking the language.

Lastly, Ayuda wants to store information about major cities of each country. Specifically, they want be able to store the name of the city, if it is the capital of the country, in what region of the country it resides, and its population.

## Conceptual Design

When starting to design a database, you usually have a scenario like the above. You need to think through what you need to store and why. If you are working with a client, you will likely go back and forth many times to refine what is necessary to store and what isn't. What you store will be dependent on the application you are building so the database design process is very subjective and context dependent – there may be many good database designs for the same database.

Once you have a solid idea of what you need to store and why, you try to identify the entities, attributes, and relationships. An entity is a person, place, event, or concept. An attribute is a characteristic of an entity. Entities will eventually map to tables and attributes will eventually map to columns of the tables. A relationship is how two entities are connected. Depending on the type of relationship, it will be represented by a column in a table or a table itself.

This process of identifying entities, attributes, and relationships is much like designing classes for an object-oriented system – the key difference being that the emphasis is on the relationships between entities instead of the behavior of classes. As in object-oriented programming, you can begin by identifying what the nouns are and whether they are an entity unto themselves or if they are actually a characteristic (i.e. an attribute) of some other entity.

In the description of what is needed for Ayuda's `world` database we can identify lots of nouns including country, continent, government, leader, population, capital, language, and city. It is fairly obvious that country is an entity with population, gross national product, and life expectancy all being characteristics of the country. It may be a little more difficult to decide whether or not continent should be considered its own entity or an attribute of country. To determine if a piece of data should be stored as an attribute instead of an entity consider if the data: (1) has any characteristics of its own, and (2) can be stored as a single piece of data (e.g., even though addresses usually don't have any attributes of their own, they are frequently considered entities since an address has a house number, street name, city, state, zip code, etc.). Since Ayuda doesn't need to store continent-specific information and since a continent can be stored simply by its name, we consider continent an attribute. Table B.1 presents one way to break down our scenario into entities and attributes.

| Entity (real world object) | Attributes (characteristics of the object) |
|---|---|
| countries | name, code, continent, GNP, population, life expectancy, government, leader |
| languages | name |
| cities | name, region, population |

**Table B.1 Entities and attributes in the `world` database**

Next we identify the relationships that connect the entities above. Here is a list of relationships we can identify between countries, languages and cities as given by our scenario:

- A language is spoken in one or more countries.
- A language is an official language of a country.
- A language is spoken by a percentage of the population of the country.
- A city is in one country.
- A capital of a country is a city.

## Logical Design

Our next step is to map entities, attributes, and relationships to tables and columns. As mentioned earlier, entities map to tables, attributes map to columns, and relationships map either to tables or columns depending on the type of relationship. Listing the tables and columns corresponding to our entities and attributes above, simply means changing the column headings of Table B.1 and renaming attributes such that they don't use spaces (column names cannot have spaces).

| Table | Columns |
|---|---|
| `countries` | `name, code, continent, gnp, population, life_expectancy, government, leader` |
| `languages` | `name` |
| `cities` | `name, region, population` |

**Table B.2 `world` database schema version 1**

The next step is to determine what the primary keys of each table will be. Remember that a primary key is an attribute that uniquely identifies each row in a table. The primary key for `countries` can simply be its county code as that is unique for each country. Similarly, the primary key for `Languages` can be its name as there are no two languages with the same name. The trickier table to pick a primary key for is `cities`. There could be two cities with the same name (e.g., Paris, France and Paris, Texas). We decide on a composite primary key consisting of name and region as there are probably no two distinct cities with the same name in the same region. Table B.3 lists the tables and columns including the proposed primary keys as being underlined column names.

| Table | Columns (primary keys underlined) |
|---|---|
| countries | name, code, continent, gnp, population, life_expectancy, government, leader |
| languages | name |
| cities | name, region, population |

**Table B.3 world database schema version 2 (with keys)**

Now to the trickiest part: mapping relationships. There are several kinds of relationships between entities. Relationships where an entity is only related to only one other entity are the easiest to map. For example, take the "A city is in a country" relationship. Since a city is associated with only one country, we can represent this relationship by simply adding a column to `cities` that references the country in which the city resides. Similarly, a capital is a kind of city and a country can only have one capital, so we can add a column to `countries` that references the City that is its capital.

Now to the relationships involving countries and languages. A country can speak many languages and a language can be spoken in many countries. Relationships like these where many of one entity can be associated with many of another entity are stored in a separate "mapping" table that associates the two entities. We create the `languages` table and in it store the attributes related to a language in a country (i.e. whether or not the language is officially recognized and the percentage of the population speaking the language). Table B.4 adds the relationships discussed.

| Table | Columns (primary keys underlined, foreign keys in bold) |
|---|---|
| countries | name, code, continent, gnp, population, life_expectancy, government, leader, **capital_name**, **capital_region** |
| languages | language, official, percentage |
| cities | name, region, population, **country_code** |

**Table B.4 world database schema version 3 (with relationships)**

We now have a proposed list of tables and columns that properly model the data that Ayuda needs to store, but there are two tweaks we can make that make it a better schema. The first is that any time there is a primary key that is made up of more than one column, every table that references that table must store all of those columns (e.g., `countries` has to store both the name and region of its capital). If this entity is referenced many places in the database it is more space efficient to just propose what is called a *surrogate key*. This is an extra column that is used to uniquely identify the entity but stores no new information about the entity. For example, we'll introduce a column in `cities` named `id` to identify a city.

Secondly, the information stored in `languages` is only needed if it is associated with a country, so all language names will need a `country_code` column to give them a proper relationship to the `countries` table. Table B.5 includes these tweaks.

| Table | Columns (primary keys underlined, foreign keys in bold) |
|---|---|
| countries | name, code, continent, gnp, population, |

| | |
|---|---|
| | life_expectancy, government, head_of_state, **capital** |
| cities | id, name, region, population, **country_code** |
| languages | **country_code**, language, official, percentage |

Table B.5 `world` database schema version 4 (tweaks)

## Normalization

A common mistake many people make (including seasoned web developers) when creating a database is putting too much information in one big table. For example, for the `world` database, why not combine `countries` and `languages` into a table like in Table B.6?

| name | code | conti-nent | gnp | popula-tion | life_expectancy | govern-ment | capi-tal | lan-guage | offi-cial | per-centage |
|---|---|---|---|---|---|---|---|---|---|---|
| Argen-tina | ARG | South America | 340238 | 37032000 | 75.1 | Federal Republic | 69 | Spanish | T | 96.8 |
| Argen-tina | ARG | South America | 340238 | 37032000 | 75.1 | Federal Republic | 69 | Italian | F | 1.7 |
| Argen-tina | ARG | South America | 340238 | 37032000 | 75.1 | Federal Republic | 69 | Indian | F | 0.3 |

Table B.6 `CountrySpeak` combined table (poor design)

This design is redundant. For every country that speaks many languages (and most do), you will be repeating the name, code, continent, GNP, population, life expectancy, government, head of state, and capital for the country for as many languages as the country speaks.

Repeating data is bad for two reasons. The first is quite obvious: space. The less you repeat the less space you take in the database. Well, these days large hard drives are pretty cheap so for small or mid-sized databases, maybe this isn't such a compelling argument.

A second reason repetition is bad that is slightly less obvious, but even more important: data consistency. When you have repeated data in a table it is very easy to have inconsistencies when you insert, update, and delete data. For example, you could easily mistype inserting 3703200 as the population for one of the rows above. The difference between 37032000 and 3703200 is huge. Now when an Ayuda user wants to know the population of Argentina, which value should they trust (both seem like viable populations for a country)? An update example: when Argentina elects a new president, an Ayuda user could easily forget to update all rows, again leaving the database in an inconsistent state. Lastly, a deletion example: Ayuda decides they no longer want to keep information about languages spoken in Argentina and so they delete all rows. Unintentionally, they have deleted all information about Argentina even though they meant only to delete information about languages spoken in Argentina. In a world where "Data is King", inconsistency is something to be avoided if at all possible.

How can we minimize data duplication and inconsistencies in our schema? The answer is a technique called normalization. There are various levels of normalization and the higher the level the higher the guarantee of consistency. We'll go through the first three levels of normalization; once you get to the third level you are guaranteed to be free of most update, insert, and deletion errors.

A table is at the first level of normalization, also called First Normal Form, if and only if there are no repeated rows (each row has some unique information) and there are no multi-valued columns. It is not uncommon for those who don't have much experience with relational databases to create a database that stores information like in Table B.7.

| name | code | continent | gnp | population | life_expectancy | government | head_of_state | capital | Languages |
|---|---|---|---|---|---|---|---|---|---|
| Argentina | ARG | South America | 340238 | 37032000 | 75.1 | Federal Republic | Christina Fernandez | 69 | Spanish (T, 96.8), Italian (F, 1.7), Indian Languages (F, 0.3) |

**Table B.7 Design for `CountrySpeak` that is not even in First Normal Form**

*Fields in tables are not meant to store lists.* They are meant to store information about a single, discrete piece of information. Here are a few of the downsides of storing lists in a multi-valued column:

- You might not have anticipated enough space if the list grows too large.
- The basic `INSERT`, `UPDATE`, and `DELETE` statements are not sufficient to manipulate multi-valued columns.
- Web programmers will have to do a lot of string parsing to get information that they need from the list.
- Table name, primary key, and column name do not map to a specific piece of data.

A table is at the second level of normalization, also called Second Normal Form, if and only if it is in First Normal form and the primary key determines all non-key column values. Table B.6 is in First Normal Form, but is not in Second Normal Form because the country code does not determine the value for language, official, or percentage. As discussed above, a table that is not in Second Normal Form is subject to errors on insert, update, and delete.

A table is at the third level of normalization, also called Third Normal Form, if it is in Second Normal Form and all columns are directly dependent on the primary key. A way to remember what Third Normal Form means was given by Bill Kent: every non-key attribute "must provide a fact about the key, the whole key, and nothing but the key so help me Codd" (Codd invented the theoretical basis for relational databases). All tables in the `world` database schema proposed in Table B.5 is in Third Normal Form as all non-key columns depend on the primary key.

An example of a table in Second Normal Form (but not in Third Normal Form) would be if we added the head of state's date of birth to the `countries` table. This is because the date of birth of the head of state relies on the person that is head of state, not on the country. The scenario where this could result in a data inconsistency is if the same person happened to be head of state in two countries at the same time (sounds ridiculous but could be viable if one country invades another), there is nothing to stop the head of state to have two different dates of birth in the two rows. In order to store this additional piece of information and stay in Third Normal Form, we would make a `HeadOfState` table in which we would store the name and date of birth and then `countries` would link to this table through a foreign key. Table B.8 summarizes the three levels of normalization.

| | |
|---|---|
| **First Normal Form** | No duplicate rows and no multi-valued columns (i.e. columns of lists) |
| **Second Normal Form** | In First Normal Form and primary key determines all non-key columns |
| **Third Normal Form** | In Second Normal Form and all columns are dependent on primary key |

**Table B.8 Three levels of normalization**

## Physical Design

At this stage of the database design process, you should have a good idea of what your tables, columns, and keys will be and that the structure of the database is safe from data duplication and inconsistencies. The last step in the design process is to figure out how the database will physically be configured for the hardware on which it runs. This includes choosing the data types of each table field and optimizing the database. Database tuning and optimization is an advanced topic and is out of the scope of this book, so in this section we will focus on choosing data types for fields.

The data types that are available to you to use are RDBMS-dependent, so we will focus on the data types that MySQL offers. Choosing the appropriate data type for each field in a database is important both in terms of correctness and speed. For example, if a field is always going to be a number, don't represent it as a string data type. MySQL offers a number of data types broken into three categories: numeric, date/time, and strings. Table B.9 lists a few of the most common types.

| MySQL Data Type | Description |
|---|---|
| **Numeric (http://dev.mysql.com/doc/refman/5.0/en/numeric-types.html)** | |
| `INTEGER` (or `INT`) | A 4-byte integer whose signed range is from -2147483648 to 2147483647. |
| `BIGINT` | An 8-byte integer whose signed range is from -9223372036854775808 to 9223372036854775807. |
| `FLOAT(M, D)` | A 4-byte floating-point number whose display length is defined by `M` (default 10) and number of decimals is defined by `D` (default 2). Decimal precision can go to 24 binary digits (roughly 7 decimal places) for a `FLOAT`. |
| **Date and Time (http://dev.mysql.com/doc/refman/5.0/en/date-and-time-types.html)** | |
| `DATE` | A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31 |
| `DATETIME` | A date and time combination in YYYY-MM-DD HH:MM:SS format, between 1000-01-01 00:00:00 and 9999-12-31 23:59:59. |
| **String Types (http://dev.mysql.com/doc/refman/5.0/en/string-types.html)** | |
| `CHAR(M)` | A fixed-length string between 1 and 255 characters in length (for example `CHAR(5)`), right-padded with spaces to the specified length when stored. |
| `VARCHAR(M)` | A variable-length string between 1 and 255 characters in length. |
| `BLOB` or `TEXT` | A field with a maximum length of 65535 characters. `BLOB`s are "Binary Large Objects" and are used to store large amounts of binary data, such as images or other types of files. Fields defined as `TEXT` also hold large amounts of data; the difference is that sorts and comparisons on stored data are case sensitive on `BLOB`s and are not case sensitive in `TEXT` fields. |
| `ENUM` | A string object with a value chosen from a list of allowed values enumerated explicitly in the column specification at table creation time. |

**Table B.9 Common MySQL data types**

Let's look at a few of the fields in the `countries` table of the `world` database to get an idea of how data types for fields should be chosen. A country's name is a string of various lengths so we choose `VARCHAR`. To determine the length of the field we ask Ayuda what is the longest name of a country name they want to store information on. Their answer is: South Georgia and the Sandwich Items. This name is 44 bytes long (each character is 1 byte). To be safe we make the data type of the country name `VARCHAR(52)`. The country code is also of string type and is guaranteed to be 3 letters long so we will make it of `CHAR(3)` type.

We know the value in the continent field should be restricted to a list of values for continents, but there are a couple of ways to classify continents. After speaking with Ayuda, they say they want the value of the continent field to be one of Asia, Europe, North America, Africa, Oceania, Antarctica, or South America. So the data type of `continent` will be `ENUM('Asia', 'Europe', 'North America', 'Africa', 'Oceania', 'Antarctica', 'South America')`.

Ayuda wants to store the GNP of a country in terms of millions of dollars. The maximum amount for a GNP can go into the trillions, so we can't use a regular `INTEGER`. One option is to use the `BIGINT` data type which will definitely store the range of values needed for GNP, but is 8 bytes long. Since Ayuda will be storing GNP at the accuracy of millions of dollars, the last six digits of any GNP stored will be zeroes. This is sort of useless information, so instead, we can use the default `FLOAT` type and use only 4 bytes to store the GNP.

## Final World Schema

The final schema for the world database is shown in Table B.10 and Table B.11.

| **countries** | |
|---|---|
| **Field** | **Type** |
| name | VARCHAR(52) |
| <u>code</u> | CHAR(3) |
| gnp | FLOAT |
| population | INTEGER |
| life_expectancy | FLOAT(3,1) |
| continent | ENUM('Asia','Europe','North America','Africa', 'Oceania','Antarctica','South America') |
| government | VARCHAR(45) |
| head_of_state | VARCHAR(60) |
| **capital** | INTEGER |

Table B.10 Final `countries` table for world database (primary keys underlined; foreign keys bolded)

| cities | |
|---|---|
| **Field** | **Type** |
| id | INTEGER |
| name | VARCHAR(35) |
| region | VARCHAR(20) |
| population | INTEGER |
| **country_code** | CHAR(3) |

| languages | |
|---|---|
| **Field** | **Type** |
| **country_code** | CHAR(3) |
| language | VARCHAR(30) |
| official | ENUM('T', 'F') |
| percentage | FLOAT(4,1) |

**Table B.11 Final `cities` and `languages` tables for `world` database (primary keys underlined; foreign keys bolded)**

## B.1.2 Data Definition Language

Phew! After all that work figuring out what the columns and tables should be for the `world` database, we are finally ready to actually define them using SQL's Data Definition Language. In MySQL to create database objects you use the `CREATE` keyword and to delete database objects you use the `DROP` keyword. The use of these is normally only permitted for users with high access levels on MySQL server; you might not have permission to use them on your school's server. The syntax to create a new database is shown in Example B.1.

```
CREATE DATABASE databaseName
```

**Example B.1 Syntax for `CREATE DATABASE` statement**

Just as with deleting table rows, you want to be careful when deleting a database because all its information is lost when you delete it. The syntax to delete a database is shown in Example B.2.

```
DROP DATABASE databaseName
```

**Example B.2 Syntax for `DROP DATABASE` statement**

Example B.3 creates a database called `test` and then deletes it in a MySQL client window. The same statements could be performed through a PHP script using the `CREATE` and `DROP` statements as arguments to `mysql_query` just as you would any other SQL statement.

```
mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.00 sec)

mysql> SHOW databases;
+--------------------+
| Database           |
+--------------------+
| imdb               |
| imdb_small         |
| test               |
+--------------------+
11 rows in set (0.00 sec)
```

```
mysql> DROP DATABASE test;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW databases;
+--------------------+
| Database           |
+--------------------+
| imdb               |
| imdb_small         |
+--------------------+
10 rows in set (0.00 sec)
```

**Example B.3 Using CREATE DATABASE and DROP DATABASE statements**

To create a table in a database, you should first select the database in which you want to create the table using the `USE` command in the client window or the `mysql_select_db` in a PHP script. The `CREATE TABLE` statement requires:

- The name of the table
- The names of the columns
- Definition columns for each column, including data type of the column and any optional column properties

The syntax to create a database table is shown in Example B.4.

```
CREATE TABLE tableName (
       column1Name column1Type [column1Properties],
       column2Name column2Type [column2Properties],
       ...,
       columnNName columnNType [columnNProperties],
       PRIMARY KEY (priKeyCol1, priKeyCol2, ..., priKeyColN),
       FOREIGN KEY (columnMName) REFERENCES otherTable (otherColumn))
```

**Example B.4 Syntax for CREATE TABLE Statement**

There are a number of column properties that you can use, but we'll only introduce you to a few. The first is `PRIMARY KEY`. If the primary key of a table only consists of one column you can use this column property to define a column as being the primary key. If you have a primary key that consists of more than one column you have to use the clause at the bottom. In Example B.5, we define the `countries` and `cities` tables using the `PRIMARY KEY` column property, but for the `languages` table we have to use the `PRIMARY KEY` clause.

Another frequently used column property is `NOT NULL` which tells the database that a user cannot insert a row into this table where the value for this column is `NULL`.

The `AUTO_INCREMENT` column property can be used with `INTEGER` and `FLOAT` types. `AUTO_INCREMENT` values should not be designated on an `INSERT` – instead the database automatically assigns the field a value starting at 1 and increments the field by 1 for each additional row added to the table. `AUTO_INCREMENT` is typically used on surrogate keys.

Example B.5 shows how to create the `world` database and its tables in a MySQL client.

```
mysql> CREATE DATABASE world;
Query OK, 1 row affected (0.00 sec)

mysql> USE world;
Database changed

mysql> CREATE TABLE countries (
    -> name VARCHAR(52) NOT NULL,
    -> code CHAR(3) PRIMARY KEY,
    -> gnp FLOAT,
    -> population INTEGER,
    -> life_expectancy FLOAT(3, 1),
    -> continent  ENUM('Asia','Europe','North
America','Africa','Oceania','Antarctica','South America') NOT NULL,
    -> government VARCHAR(45),
    -> head_of_state VARCHAR(60),
    -> capital INTEGER,
    -> FOREIGN KEY (capital) REFERENCES City(id));

mysql> CREATE TABLE cities (
    -> id INTEGER AUTO_INCREMENT PRIMARY KEY,
    -> name VARCHAR(35) NOT NULL,
    -> region VARCHAR(20),
    -> population INTEGER,
    -> country_code CHAR(3) NOT NULL,
    -> FOREIGN KEY (country_code) REFERENCES Country(code));

mysql> CREATE TABLE languages (
    -> country_code CHAR(3),
    -> language VARCHAR(30),
    -> official ENUM('T', 'F'),
    -> percentage FLOAT(4, 1),
    -> PRIMARY KEY (country_code, language),
    -> FOREIGN KEY (country_code) REFERENCES Country(code));
Query OK, 0 rows affected (0.00 sec)
```

**Example B.5 Creating the `world` database**

There are many more useful statements and components of the MySQL Data Definition Language that is out of the scope of this book. To learn more visit the MySQL Documentation for Data Definition Statements at http://dev.mysql.com/doc/refman/6.0/en/sql-syntax-data-definition.html

## References

- Wikipedia - Database design: http://en.wikipedia.org/wiki/Database_design
- Wikipedia - Data Definition Language: http://en.wikipedia.org/wiki/Drop_(SQL)
- Database Design for Mere Mortals:
    - http://www.amazon.com/Database-Design-Mere-Mortals-Hands/dp/0201752840/
- Database Modeling and Design: Logical Design:
    - http://www.amazon.com/Database-Modeling-Design-Kaufmann-Management/dp/0126853525/