Creating Tables...contd.

Start Here

Storing Data

Now that you have a basic understanding of databases, SQL, and MySQL, this topic begins the process of taking that knowledge deeper.  The focus in this chapter, as the title states, is real-world database design. Like the work we did last week, much of the effort this week requires paper and pen—and serious thinking about what your applications will need to do.

Data stored in a relational database is organized into **tables**.  A database table organizes data in a grid-like fashion, where each entry forms a row and each column identifies a specific value in the entry.  To illustrate this, here’s a table showing the number of medals won by each of the top five medal-winning countries that participated in the 2014 Winter Olympic Games.  Each row lists the country’s name, how many gold medals, silver medals, and bronze medals were won, and the total number of medals won.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Country | Gold | Silver | Bronze | Total |
| Russia | 13 | 11 | 9 | 33 |
| United States | 9 | 7 | 12 | 28 |
| Norway | 11 | 5 | 10 | 26 |
| Canada | 10 | 10 | 5 | 25 |
| Netherlands | 8 | 7 | 9 | 24 |

 A table like the one above is “physical” in that we can see it printed in a book or drawn on a whiteboard.  It’s limited only by the amount of physical space available.  On the other hand, a database table is an intangible structure stored somewhere on a hard drive or in computer memory.  We can only imagine it or make drawings to represent it.  A database table is interpreted by a computer process (such as MySQL), and the limitations of the interpreting process impose restrictions on the table.  The number of columns, the number of rows, and even what the individual values in a row can be, all depend upon what the computer system and database server can handle.  But despite these limitations, a database table is very flexible.  We can define relationships between tables, combine multiple tables together, sort rows and view specific entries, remove rows, and easily perform various calculations on the data.

We’ll look at the **CREATE TABLE** statement—which defines new database tables—and discuss some important details surrounding table creation: MySQL’s supported data types, naming restrictions, and storage engines.  We’ll also see how to add rows to a table with the INSERT statement, and finish by discussing transactions.

Creating Tables

Tables are created using the CREATE TABLE statement.  In its simplest form, the statement provides the name of the table we we want to create and a list of column names and their data types.  Not surprisingly, a CREATE TABLE statement can be very very complex depending on the requirements driving the design of the table.  We can specify one or more attributes as part of a column’s definition; such attributes can limit the range of values the column can store or specify a default value when one isn’t provided by the user. Defining any logical relationships that exist between the table and another, and which storage engine MySQL should use to manage the table, is also common.  You can see how detailed the statement can be if you look at the syntax and options for CREATE TABLE in the [MySQL documentation. (Links to an external site.)](http://dev.mysql.com/doc/refman/5.6/en/create-table.html)

Let’s look at a pair of relatively simple CREATE TABLE statements.  (I’ll highlight some common points that add complexity, but I won’t get too crazy, I promise.)  With the "company" database created in the previous page as your active database, issue the statements below.  MySQL should respond “Query OK” after each one.

CREATE TABLE employees (  
employee\_id INTEGER UNSIGNED NOT NULL AUTO\_INCREMENT,   
last\_name VARCHAR(30) NOT NULL,   
first\_name VARCHAR(30) NOT NULL,   
email VARCHAR(100) NOT NULL,   
hire\_date DATE NOT NULL,  
notes MEDIUMTEXT,  
  
PRIMARY KEY (employee\_id),  
INDEX (last\_name),  
UNIQUE (email) );

CREATE TABLE address (  
employee\_id INTEGER UNSIGNED NOT NULL, address VARCHAR(50) NOT NULL,  
city VARCHAR(30) NOT NULL,  
state CHAR(2) NOT NULL,  
postcode CHAR(5) NOT NULL,  
FOREIGN KEY (employee\_id)  
  
REFERENCES employee (employee\_id) );

The first statement creates a table named employee, designed to store basic information about a company’s employees—their name, email address, date of hire, and perhaps any notes the Human Resources director might provide.  The formatting is just to keep things readable for ourselves; it makes no difference to MySQL whether we write a statement entirely on one line or across several lines with indentation.  The spacing in a statement is also generally irrelevant.

**LOCAL BIAS:**

The address table has a North American bias.  An address in the United States or Mexico fits perfectly, and a Canadian address can store the two-letter province or territory abbreviation in the state column.  But an address in the Netherlands, for example, needs space for a 6-character postal code.  Feel free to adapt the definition to your own locale.

Names chosen for a table and its columns can be anything we like so long as they adhere to the following restrictions:

* The name uses basic Latin letters (A–Z, both uppercase and lowercase), the dollar sign ($), underscore (\_), or Unicode characters U+0080–U+FFFF.
* The null character 0x00, Unicode characters U+10000 and higher, and characters that are prohibited in file names like slash (/), backslash (\), and period are not allowed in a name.
* The name must be quoted if it contains characters outside of the above.  MySQL uses back ticks by default for this (`…`) although it can be configured to use single quotes ('…') as well.  I recommend sticking with the default.
* The name must be quoted if it’s a MySQL reserved keyword. A list of reserved words can be found in the [online documentation (Links to an external site.)](https://dev.mysql.com/doc/refman/5.7/en/keywords.html).

The employee\_id column is designated as the table’s primary key.  A **primary key** is a column in which all the values are distinct and can be used to uniquely identify each row in the table.  In more complex table definitions, we may define a primary key from multiple columns together but using a single INTEGER type column is the most common practice.  Only one primary key can be defined per table (hence the name primary key).

The employee\_id column also has the **AUTO\_INCREMENT** attribute.  Whenever we add a row that doesn’t provide a value for this column, MySQL will automatically use the next highest sequential integer as its value.  Suppose we have several rows in the employee table and the largest employee\_id value among them is 42.  If we add a new row without an employee\_id value, MySQL will use 43 for the missing value.  If we then add another row without the value, MySQL will use 44, and so on.  Only one column in the table can be designated an auto-increment column, and the column must also be a primary key.

Behind the scenes, MySQL maintains various data structures to track data and relationships.  The INDEX defined on last\_name lets MySQL know that we might use its value in our selection criteria later when we retrieve rows—for example, if we wanted to search for employees named Smith or Jones.  MySQL will create and manage a special index structure with the values in the column to make its search more efficient. Don’t go overboard adding indexes though.  It takes time for MySQL to maintain them so row retrieval may be faster, but adding/updating rows will be slower.

The term**constraint** describes a special condition imposed on a column or table that must always be adhered to.  Most of the column definitions have NOT NULL, a constraint that prohibits storing NULL values in the column.  NULL is a special value that represents the absence of a value.  Essentially, NOT NULL means the column must hold a value.  MySQL treats NULL differently from an empty value, such as an empty text string.

The UNIQUE constraint defined on the email column ensures all the email addresses stored in the table are different.  UNIQUE and PRIMARY KEY are similar, but there are important differences between them. Because the values in a primary key column must be able to unambiguously identify each row, its uniqueness is inherent.  We don’t explicitly specify UNIQUE with PRIMARY KEY.  And while only one primary key can be defined per table, we can provide any number of UNIQUE constraints.  A UNIQUE column may also contain NULL values, something PRIMARY KEY doesn’t allow.

The **FOREIGN KEY** constraint in the address table’s CREATE TABLE statement references the employee table, thus defining a relationship between the two tables.  This relationship means that a row in the address table is logically related to whatever row in the employee table that has the same value in its employee\_id column.  Take, for instance, a row in the address table with an employee\_id value of 42.  That row may be associated with the row in the employee table whose employee\_id value is also 42.  In other words, an address with employee\_id 42 is linked to employee 42’s employee record.  A FOREIGN KEY column doesn’t need to have the same name as its partner column in the other table, but the two must share the same data type and NULL constraint.

We can issue DESCRIBE or SHOW CREATE TABLE statements to verify a table was created or view the definition of an existing table. The DESCRIBE statement returns the list of the table’s column names and their data types, and SHOW CREATE TABLE returns a statement that can be used later to re-create the table.

DESCRIBE employee;

SHOW CREATE TABLE employee;

**Pick A Convention:** A convention I’ve adopted is to type MySQL keywords in uppercase and my own identifiers in lowercase.  MySQL doesn’t treat keywords and column names in a case-sensitive manner, but table names might be case-sensitive depending on the file system storing your tables’ files.  It’s best to pick a convention—whatever it may be—and stick with it.

So far, we’ve discussed the column attributes and table constraints that appear in the example, but we haven't discussed the **data types**. The next part may be a little dry, but it covers some important information.  Each type requires a different amount of storage on disk and in memory, so we always want to specify the minimum viable type for a column.  The amount of wasted space from assigning a data type that’s larger than necessary might be negligible at first because there’s only a handful of rows, but it can add up quickly as more and more data is added to the table.

# Data Types and Storage Requirements

# Start Here

### Data Types and Storage Requirements

MySQL supports many different data types, most of which we’ll discuss in the following paragraphs.  The term data type refers to the classification of data based on its possible values, the set of operations we can perform on it, and its storage requirements.  Values of the INTEGER type can only consist of integers like 0, 42, and 1337.  This is different from the DECIMAL type which consists of decimal numbers like 1.61, 3.14, and 100.0.  We can perform operations like addition, subtraction, multiplication, and division on INTEGER and DECIMAL values, but these cannot be per-formed on text-based types like CHAR and TEXT.

**Numeric Types**

MySQL offers the INTEGER (also abbreviated as INT), TINYINT, SMALLINT, MEDIUMINT, and BIGINT data types for storing integer data.  These types differ in the number of bytes they occupy to represent a value. This in turn limits the range of integers each type can hold.  For example, TINYINT uses 1 byte, so its range is -128 to 127—the range of numbers than can be expressed in binary with 8 bits. INTEGER uses 4 bytes, so its range is larger: -2,147,483,648 to 2,147,483,647.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Type | Storage Used  (Bytes) | Min. Signed | Max. Signed | Min. Unsigned | Max. Unsigned |
| TINYINT | 1 | -128,127 | 127 | 0 | 255 |
| SMALLINT | 2 | -32,768 | 32,767 | 0 | 65,535 |
| MEDIUMINT | 3 | -8,388,608 | 8,388,607 | 0 | 6,777,215 |
| INTEGER | 4 | -2,147,483,648 | 2,147,483,647 | 0 | 4,294,967,295 |
| BIGINT | 8 | -9,223,372, 036,854,775, 808 | 9,223,372,  036,854,775  807 | 0 | 18,446,744,  073,709,551,  615 |

We can also specify the UNSIGNED attribute with integer-based types.  The type consumes the same amount of space but negative values are disallowed in exchange for raising the upper bound.  For example, the range of TINYINT UNSIGNED becomes 0 to 255.  Both TINYINT and TINYINT UNSIGNED represent a range of 256 integers, but their starting points are -128 and 0 respectively.

The following table shows the storage requirements and range for each of MySQL’s integer types, both signed and unsigned:

DECIMAL, FLOAT, and DOUBLE are types that support real numbers.  We also must provide the precision (the number of total digits) and scale (the number of digits that follow the decimal point) when we use one of these types.  DECIMAL(5,2) has a range of -999.99 to 999.99—that is, five digits in total with two of them following the decimal point.  We can specify UNSIGNED for these types as well, but doing so only disallows negative values.  This is because the upper limit is defined by the precision and scale we provide.

The DECIMAL type is a fixed-point data type which means it preserves the exact precision of its value in calculations.  This is useful for representing values like monetary amounts.  The maximum precision we can specify for DECIMAL is 65, and the maximum scale is 30.  On the other hand, FLOAT and DOUBLE are both floating-point types.  Calculations with these types are approximate because some rounding may occur due to how the values are represented internally in the computer.  The difference between FLOAT and DOUBLE is the amount of space they occupy, which in turn affects their accuracy.  FLOAT is 4-byte single-precision which is generally accurate up to 7 decimal places.  DOUBLE is 8-byte double-precision which is generally accurate up to 15 decimal places.

The BIT data type stores a bit-sequence.  This is useful for storing bit-field values like flags and bit masks. BIT has a capacity of 1 to 64 bits.  BIT(1) can only hold 0 or 1; BIT(2) can hold the binary values 00, 01, 10, and 11; BIT(3) can hold the binary values 000, 001, 010, 011, 100, 101, 110, and 111, and so on. MySQL uses the notation b'value' to specify the value as string of binary digits, like b'101010'.

**String Types**

MySQL devotes several data types to storing textual data: CHAR, VARCHAR, BINARY, VARBINARY, TEXT, TINYTEXT, MEDIUMTEXT, LONGTEXT, BLOB, TINYBLOB, MEDIUMBLOB, and LONGBLOB.  The sized types like TINYTEXT and MEDIUMTEXT behave exactly like TEXT although each is constrained by a different maximum amount of text it can hold.  The same is true for BLOB and its sized counterparts, TINYBLOB, MEDIUMBLOB, and LONGBLOB.

We must provide a length when we specify a CHAR or VARCHAR type. CHAR(255), for instance, stores text strings 255 characters long, and VARCHAR(255) stores strings up to 255 characters in length.  Notice that I said “255 characters” and “up to 255 characters.”  CHAR is intended to store fixed-length strings, values that will always have the same number of characters across all rows in the table.  The amount of space remains constant.  VARCHAR stores variable-length strings, values that can have different lengths across the rows.  The amount of space each value occupies is determined by the length of the string.

I’ll highlight the difference between CHAR and VARCHAR using the string “Hello World”.  The string is 11 characters long, and it will occupy 11 bytes (plus an extra byte or two that MySQL needs to add for its own bookkeeping) if we store it in a VARCHAR(255) column.  But with CHAR(255), the storage space is constant across all rows in the table.  MySQL pads the string with 244 spaces.  The padding is removed when we retrieve the string and the original 11-character “Hello World” string is returned, but all CHAR(255) strings occupy 255 bytes when they’re stored.

**Maximum Lengths:** CHAR and VARCHAR have different maximum lengths.  CHAR is allowed up to 255 characters and VARCHAR is allowed up to 65,535 characters.  You probably won’t want to use VARCHAR(65535) though.  MySQL limits the size of a row to 25,535 bytes.  Almost all of the row’s columns contribute to the size (TEXT and BLOB are excluded), so you wouldn’t have space left for the other columns. You find detailed information about the limits on row and column sizes in the [online documentation (Links to an external site.)](https://dev.mysql.com/doc/refman/5.6/en/column-count-limit.html).

BINARY and VARBINARY behave similarly to CHAR and VARCHAR except they’re used for binary strings. MySQL treats binary strings as a series of bytes, not characters, and doesn’t take collation or character set into consideration when working with them.  No special semantics are applied; any sorting or comparison operations performed are based on the ordinal value of each byte.  MySQL uses the NULL byte 0x00 to pad/strip BINARY values.

TEXT and BLOB are variable-length data types for storing larger amounts of text.  Neither performs padding/stripping, which makes them ideal for preserving the exact nature of the data.  TEXT values are treated as character strings and BLOB values are treated as binary strings.  The TEXT and BLOB columns (and their sized variants) are also excluded when MySQL calculates the length of a row, so consider using one of them when you need to store more than 65,535 bytes of data.

The non-binary string types CHAR, VARCHAR, and TEXT can be given the CHARACTER SET attribute to specify the data’s encoding.  A character set determines how the underlying bits and bytes are interpreted as human-readable characters.  Common sets include ASCII (ascii), ISO 8859-1 (latin1), and UTF-8 (utf8). The default character set when none is specified is latin1.

CREATE TABLE charset\_example (  
id INTEGER UNSIGNED NOT NULL AUTO\_INCREMENT,  
ascii\_string VARCHAR(255) CHARACTER SET ascii NOT NULL,   
latin1\_string VARCHAR(255) CHARACTER SET latin1 NOT NULL,   
utf8\_string VARCHAR(255) CHARACTER SET utf8 NOT NULL,  
  
PRIMARY KEY (id)  
);

The ENUM and SET data types restrict a value to those allowed by a defined list.  A column defined as ENUM('Alpha', 'Beta', 'Gamma') can only contain one of the strings listed in the definition, either “Alpha”, “Beta”, or “Gamma”.  SET holds strings with one or more comma-separated values from its list. SET('Alpha', 'Beta', 'Gamma') can hold values like “Alpha”, “Alpha,Beta,Gamma”, “Beta,Gamma”, and so on.

**Use ENUM with Caution:** I have nothing against ENUM when it’s use suits my data, but this seemingly innocent data type is not without controversy.   Chris Komlenic’s [blog post (Links to an external site.)](http://komlenic.com/244) “8 Reasons Why MySQL’s ENUM Data Type Is Evil” is a good read on the subject.

The following table shows the storage requirements and the maximum length allowed for MySQL’s string types:

|  |  |  |
| --- | --- | --- |
| Data Type | Storage Used (Bytes) | Maximum |
| CHAR(m) | m × maximum-size character in the character set | 255 chars |
| VARCHAR(m) | up to 2 bytes + m × maximum-size character in the character set | 65,535 chars |
| TEXT | size of string in bytes + 2 | 65,535 chars |
| TINYTEXT | size of string in bytes + 1 | 255 chars |
| MEDIUMTEXT | size of string in bytes + 3 | 16,777,215 chars |
| LONGTEXT | size of string in bytes + 4 | 4,294,967,295 chars |
| BINARY(m) | m | 255 bytes |
| VARBINARY(m) | up to 2 bytes + m | 65,535 bytes |
| BLOB | size of string in bytes + 2 | 65,535 bytes |
| TINYBLOB | size of string in bytes + 1 | 255 bytes |
| MEDIUMBLOB | size of string in bytes + 3 | 16,777,215 bytes |
| LONGBLOB | size of string in bytes + 4 | 4,294,967,295 bytes |
| ENUM | up to 2 bytes | 65,535 values |
| SET | up to 8 bytes | 64 members |

**Temporal Types**

The data types DATETIME, TIMESTAMP, DATE, TIME, and YEAR are for working with date and time values.  Both DATETIME and TIMESTAMP hold values containing date and time parts using the format 'YYYY-MM-DD HH:mm:ss' (YYYY is a four-digit year, MM is a two-digit month, DD a two-digit day, HH a two-digit hour, mm two-digit minutes, and ss two-digit seconds).  For example, '2015-03-15 13:15:00' is 1:15 p.m. on the third of March, 2015.  The DATE, TIME, and YEAR types all store their single respective part values.  Besides 'HH:mm:ss' to represent a time of day—such as 13:15:00—TIME values may also be given like 'HHH:mm:ss' to represent an elapsed amount of time—such as 293:23:10, meaning 293 hours, 23 minutes, and 10 seconds.

MySQL can automatically initialize and update DATETIME and TIMESTAMP values with the current date and time whenever a row is added or updated. If the INSERT statement adds a new row to the table but doesn’t have a value for the TIMESTAMP column, MySQL will use the current date and time as the value. MySQL also updates a TIMESTAMP column’s value when any of the values in its row are updated.  While this behavior is automatic for TIMESTAMP, we can also apply it to DATETIME columns by specifying the attributes DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP in the column definition.

**Variations in Temporal Types:** There are a few subtleties to the behavior of temporal data types across different releases of MySQL, especially with TIMESTAMP and YEAR.  Review the documentation on[timestamp initialization (Links to an external site.)](http://dev.mysql.com/doc/refman/5.6/en/timestamp-initialization.html) and [two-digit years (Links to an external site.)](http://dev.mysql.com/doc/refman/5.6/en/two-digit-years.html).

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Storage Used (Bytes) | Minimum | Maximum |
| DATETIME | 8 | 1000-01-01 00:00:00 | 9999-12-31 23:59:59 |
| TIMESTAMP | 4 | 1970-01-01 00:00:01 UTC | 2038-01-19 03:14:07 UTC |
| DATE | 3 | 1000-01-01 | 9999-12-31 |
| TIME | 3 | -838:59:59 | 838:59:59 |
| YEAR | 1 | 1901 | 2155 |

The following table shows the range and storage requirements for each of MySQL’s temporal types:

Design and Normalization Techniques

Database Design

Now that you have a basic understanding of databases, SQL, and MySQL, this topic begins the process of taking that knowledge deeper.  The focus in this chapter, as the title states, is real-world database design. Like the work we did last week, much of the effort this week requires paper and pen—and serious thinking about what your applications will need to do.

We begin this week with thorough coverage of database normalization, a vital approach to the design process.  After that, we focus on design-related concepts specific to MySQL: working with indexes, table types, language support, times, and foreign key constraints.

You’ll explore steps involved in proper database design and how to make the most of MySQL.  You’ll also plan a couple of multi-table databases.  Next week you’ll learn more advanced SQL and MySQL, and put these concepts to use.

Normalization

Whenever you are working with a relational database management system such as MySQL, the first step in creating and using a database is to establish the database’s structure (also called the database schema). Database design, also known as data modeling, is crucial for successful long-term management of information.  Using a process called normalization, you carefully eliminate redundancies and other problems that would undermine the integrity of your database.

The techniques you will learn this week will help ensure the viability, usefulness, and reliability of your databases.  The primary example we will discuss is that of a forum where users can post messages, but the principles of normalization apply to any database you might create.

Normalization was developed by an IBM researcher named E. F. Codd in the early 1970s (he also invented the relational database).  A relational database is merely a collection of data, organized in a particular manner, and Dr. Codd created a series of rules called normal forms that help define that organization.  Below sections discusses the first three of the normal forms, which are sufficient for most database designs.

Before you begin normalizing your database, you must define the role of the application being developed. Whether it means that you thoroughly discuss the subject with a client or figure it out for yourself, understanding how the information will be accessed dictates the modeling.  Thus, this process will require paper and pen rather than the MySQL software itself (although database design is applicable to any relational database, not just MySQL).

In this example, I want to create a message board where users can post messages and other users can reply. I imagine that users will need to register, and then log in with an email address/password combination to post messages.  I also expect that there could be multiple forums for different subjects.  I have listed a sample row of data in the following table.  The database itself will be called forum.

***Table-1: Display Sample Forum Data***

|  |  |
| --- | --- |
| Item | Example |
| username | troutster |
| password | mypass |
| actual name | Michael Clarke |
| user email | email@example.com |
| forum | MySQL |
| message subject | Question about normalization |
| message body | I have a question about… |
| message date | November 2, 2017 12:20 AM |

***Important Tips:***

* One of the best ways to determine what information should be stored in a database is to think about what questions will be asked of the database and what data would be included in the answers.
* Always err on the side of storing more information than you might need.  It’s easy to ignore unnecessary data but impossible to later manufacture data that was never stored in the first place.
* Normalization can be hard to learn if you fixate on the little things.  Each of the normal forms is defined in a very cryptic way; even when put into layman’s terms, they can still be confounding.  My best advice is to focus on the big picture as you follow along.  Once you’ve gone through normalization and seen the end result, the overall process should be clear enough.

Keys

Keys are integral to normalized databases.  There are two types of keys: **primary** and **foreign**.  A primary key is a unique identifier that has to abide by certain rules.  They must:

* **Always have a value (they cannot be NULL)**
* Have a value that remains the same (never changes)
* Have a unique value for each record in a table

A good real-world example of a primary key is the U.S. Social Security number: everyone has a unique Social Security number, and that number never changes.  Just as the Social Security number is an artificial construct used to identify people, you’ll frequently find creating an arbitrary primary key for each table to be the best design practice.

The second type of key is a foreign key.  Foreign keys are the representation in Table B of the primary key from Table A.  If you have a cinema database with a movies table and a directors table, the primary key from directors would be linked as a foreign key in movies.  You’ll see better how this works as the normalization process continues.

The forum database is just a simple table as it stands (above table), but before beginning the normalization process, identify at least one primary key.  The foreign keys will come in later steps.

**To assign a primary key:**

1. Look for any fields that meet the three tests for a primary key.  
   In this example (Table-1), no column fits all the criteria for a primary key.  The username and email address will be unique for each forum user but will not be unique for each record in the database because the same user could post multiple messages.  The same subject could be used multiple times as well.  The message body will likely be unique for each message but could change (if edited), violating one of the rules of primary keys.
2. If no logical primary key exists, invent one (Table-2).

***Table-2: Display Sample Forum Data***

|  |  |
| --- | --- |
| Item | Example |
| message ID | 325 |
| username | troutster |
| password | mypass |
| actual name | Michael Clarke |
| user email | email@example.com |
| forum | MySQL |
| message subject | Question about normalization |
| message body | I have a question about… |
| message date | November 2, 2017 12:20 AM |

Frequently, you will need to create a primary key because no good solution presents itself.  In this example, a message ID is manufactured.  When you create a primary key that has no other meaning or purpose, it’s called a surrogate primary key.

***Important Tips:***

* As a rule of thumb, I name my primary keys using at least part of the table’s name (e.g., message) and the word id.  Some database developers like to add the abbreviation pk to the name as well.  Some developers just use id.
* MySQL allows for only one primary key per table, although you can base a primary key on multiple columns.  A multiple-column primary key means the combination of those columns must be unique and never change.
* Ideally, your primary key should always be an integer, which results in better MySQL performance.

Relationships

Database relationships refer to how the data in one table relates to the data in another.  There are three types of relationships between any two tables: **one-to-one**, **one-to-many**, or **many-to-many**.  Two tables in a database may also be unrelated.

A relationship is one-to-one if one and only one item in Table A applies to one and only one item in Table B. For example, each U.S. citizen has only one Social Security number, and each Social Security number applies to only one U.S. citizen; no citizen can have two Social Security numbers, and no Social Security number can refer to two citizens.

A relationship is one-to-many if one item in Table A can apply to multiple items in Table B.  The terms on and off will apply to many switches, but each switch can be in only one state or the other.  A one-to-many relationship is the most common one between tables in normalized databases.

Finally, a relationship is many-to-many if multiple items in Table A can apply to multiple items in Table B.  A book can be written by multiple authors, and authors can write multiple books.  Although many-to-many relationships are common in the real word, you should avoid many-to-many relationships in your design because they lead to data redundancy and integrity problems.  Instead of having many-to-many relationships, properly designed databases use intermediary tables that break down one many-to-many relationship into two one-to-many relationships.

A many-to-many relationship between two tables will be better represented as two one-to-many relationships those tables have with an intermediary table.

Relationships and keys work together in that a key in one table will normally relate to a key in another, as mentioned earlier.

Database modeling uses certain conventions to represent the structure of the database, which I’ll follow through a series of images throughout this week.  The symbols for the three types of relationships are shown below.  These symbols, or variations on them, are commonly used to represent relationships in database modeling schemes.

**Important Tips:**

* The process of database design results in an ERD (entity-relationship diagram) or ERM (entity-relationship model).  This graphical representation of a database uses shapes for tables and columns and the symbols from above figure to represent the relationships.
* The term “relational” in RDBMS actually stems from the tables, which are technically called relations.

First Normal Form (1NF)

As already stated, normalizing a database is the process of changing the database’s structure according to several rules, called forms.  Your database should adhere to each rule exactly, and the forms must be followed in order.

Every table in a database must have the following two qualities to be in First Normal Form (1NF):

* Each column must contain only one value (this is sometimes described as being atomic or indivisible).
* No table can have repeating groups of related data.

A table containing one field for a person’s entire address (street, city, state, zip code, country) would not be 1NF compliant, because it has multiple values in one column, violating the first property.  As for the second, a movies table that had columns such as actor1, actor2, actor3, and so on would fail to be 1NF compliant because of the repeating columns all listing the exact same kind of information.

To begin the normalization process, check the existing structure (Table-2) for 1NF compliance.  Any columns that are not atomic should be broken into multiple columns.  If a table has repeating similar columns, then those should be turned into their own, separate table.

**To make a database 1NF compliant:**

* Identify any field that contains multiple pieces of information.
  + Looking at Table-2, one field is not 1NF compliant: actual name.  The example record contained both the first name and the last name in this one column.
  + The message date field contains a day, a month, and a year, plus a time, but subdividing past that level of specificity isn’t warranted.  And, MySQL can handle dates and times quite nicely using the DATETIME type.
  + Other examples of problems would be if a table used just one column for multiple phone numbers (mobile, home, work) or stored a person’s multiple interests (cooking, dancing, skiing, etc.) in a single column.
* Break up any fields found in Step 1 into distinct fields (Table-3, see below).
  + To fix this problem for the current example, create separate first name and last name fields, each containing only one value.

|  |  |
| --- | --- |
| Item | Example |
| message ID | 325 |
| username | troutster |
| password | mypass |
| first name | Michael |
| last name | Clarke |
| user email | email@example.com |
| forum | MySQL |
| message subject | Question about normalization |
| message body | I have a question about… |
| message date | November 2, 2017 12:20 AM |

***Table-3: Forum Database, Atomic***

* Turn any repeating column groups into their own table.
  + The forum database doesn’t have this problem currently, so to demonstrate what would be a violation, consider Table-4 below.  The repeating columns—the multiple actor fields—introduce two problems.  First, there’s no getting around the fact that each movie will be limited to a certain number of actors when stored this way.  Even if you add columns actor 1 through actor 100, there will still be that limit (of a hundred).  Second, any record that doesn’t have the maximum number of actors will have NULL values in those extra columns.  You should generally avoid columns with NULL values in your database schema.  As another concern, the actor and director columns are not atomic.

***Table-4: Movies Table***

|  |  |
| --- | --- |
| Column | Value |
| movie ID | 976 |
| movie title | Casablanca |
| year released | 1943 |
| director | Michael Curtiz |
| actor 1 | Humphrey Bogart |
| actor 2 | Ingrid Bergman |
| actor 3 | Peter Lorre |

* + To fix the problems in the movies table, a second table would be created (Table-5).  This table uses one row for each actor in a movie, which solves the problems mentioned in the last paragraph.  The actor names are also broken up to be atomic.  Notice as well that a primary key column should be added to the new table.  The notion that each table has a primary key is implicit in the First Normal Form.

***Table-5: Movies-Actors Table***

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Movie | Actor First Name | Actor Last Name |
| 1 | Casablanca | Humphrey | Bogart |
| 2 | Casablanca | Ingrid | Bergman |
| 3 | Casablanca | Peter | Lorre |
| 4 | The Maltese Falcon | Humphrey | Bogart |
| 5 | The Maltese Falcon | Peter | Lorre |

* Double-check that all new columns and tables created in Steps 2 and 3 pass the 1NF test.

**Important Tips:**

* The simplest way to think about 1NF is that this rule analyzes a table horizontally: inspect all of the columns within a single row to guarantee specificity and avoid repetition of similar data.
* Various resources will describe the normal forms in somewhat different ways, likely with much more technical jargon.  What is most important is the spirit—and end result—of the normalization process, not the technical wording of the rules.

Second Normal Form (2NF)

For a database to be in Second Normal Form (2NF), the database must first already be in 1NF.  You must normalize in order.  Then, every column in the table that is not a foreign key must be dependent on the primary key.  You can normally identify a column that violates this rule when it has non-key values that are the same in multiple rows.  Such values should be stored in their own table and related back to the original table through a key.

Going back to the *cinema* example, a movies table (Table-4) would have the director Martin Scorsese listed 20+ times.  This violates the 2NF rule, as the column(s) that store the directors’ names would not be keys and would not be dependent on the primary key (the movie ID).  The fix is to create a separate *directors* table that stores the directors’ information and assigns each director a primary key.  To tie the director back to the movies, the director’s primary key would also be a foreign key in the *movies* table.

Looking at Table-5 (for actors in movies), both the movie name and the actor names are also in violation of the 2NF rule: they aren’t keys and they aren’t dependent on the table’s primary key.  In the end, the *cinema* database in this minimal form requires four tables.  Each director’s name, movie name, and actor’s name will be stored only once, and any non-key column in a table is dependent on that table’s primary key.  In fact, normalization could be summarized as the process of creating more and more tables until potential redundancies have been eliminated.

**Note:** To make the *cinema* database 2NF compliant (given the information being represented), four tables are necessary. The directors are represented in the *movies* table through the*director ID* key; the movies are represented in the *movies-actors* table through the *movie ID* key; and the actors are represented in the *movies-actors* table through the *actor ID* key.

To make a database 2NF compliant:

* Identify any non-key columns that aren’t dependent on the table’s primary key.
  + Looking at Table-3, the username, first name, last name, email, and forum values are all non-keys (message ID is the only key column currently), and none are dependent on the message ID. Conversely, the message subject, body, and date are also non-keys, but these do depend on the message ID.
* Create new tables accordingly, to make the forum database 2NF compliant, three tables are necessary.  The most logical modification for the forum database is to make three tables: users, forums, and messages.  In a visual representation of the database, create a box for each table, with the table name as a header and all its columns (also called its attributes) underneath.
* Assign or create new primary keys.
  + Each table needs its own primary key.  Using the techniques described earlier, ensure that each new table has a primary key.  Here I’ve added a user ID field to the users table and a forum ID field to forums.  These are both surrogate primary keys.  Because the username field in the users table and the name field in the forums table must be unique for each record and must always have a value, you could have them act as the primary keys for their tables.  However, this would mean that these values could never change (per the rules of primary keys) and the database would be a little slower, using text-based keys instead of numeric ones.
* Create the requisite foreign keys and indicate the relationships.  To relate the three tables, add two foreign keys to the messages table, each key representing one of the other two tables.
  + The final step in achieving 2NF compliance is to incorporate foreign keys to link associated tables. Remember that a primary key in one table will often be a foreign key in another.
  + With this example, the user ID from the users table links to the user ID column in the messages table.  Therefore, users has a one-to-many relationship with messages: each user can post multiple messages, but each message can be posted by only one user.
  + Also, the two forum ID columns are linked, creating a one-to-many relationship between messages and forums: each message can only be in one forum, but each forum can have multiple messages.
  + There is no direct relationship between the users and forums tables.

**Important Tips:**

* Another way to test for 2NF is to look at the relationships between tables.  The ideal is to create one-to-one or one-to-many situations.  Tables that have a many-to-many relationship may need to be restructured.
* Looking back at an earlier step (the movies-actors table is an intermediary table), which turns the many-to-many relationship between movies and actors into two one-to-many relationships.  You can often tell a table is acting as an intermediary when all its columns are keys.  In fact, in that table, the primary key could be the combination of the movie ID and the actor ID.
* A properly normalized database should never have duplicate rows in the same table: two or more rows in which the values in every non–primary key column match.
* To simplify how you conceive of the normalization process, remember that 1NF is a matter of inspecting a table horizontally, and 2NF is a vertical analysis: hunting for repeating values over multiple rows.

Third Normal Form (3NF)

A database is in Third Normal Form (3NF) if it is in 2NF and every non-key column is mutually independent. If you followed the normalization process properly to this point, you may not have 3NF issues.  You would know that you have a 3NF violation if changing the value in one column would require changing the value in another.  In the *forum* example thus far, there aren’t any 3NF problems, but I’ll explain a hypothetical situation where this rule would come into play.

Take, as an example, a database about books.  After applying the first two normal forms, you might end up with one table listing the books, another listing the authors, and a third acting as an intermediary table between books and authors, since there’s a many-to-many relationship there.  If the books table listed the publisher’s name and address, that table would be in violation of 3NF (this database as currently designed fails the 3NF test.).  The publisher’s address isn’t related to the book, but rather to the publisher itself.  In other words, that version of the *books* table has a column that’s dependent on a non-key column: the publisher’s name.

As I said, the *forum* example is fine as is, but I’ll outline the 3NF steps just the same, showing how to fix the books example just mentioned.

To make a database 3NF compliant:

* Identify any fields in any tables that are interdependent.
  + As just stated, what you need to look for are columns that depend more on each other than they do on the record as a whole. In the *forum* database, this isn’t an issue. Just looking at the *messages* table, each *subject* will be specific to a *message ID*, each body will be specific to that *message ID*, and so forth.  With a books example, the problematic fields are those in the *books* table that pertain to the publisher.
* Create new tables accordingly.
  + If you found any problematic columns in Step 1, like *address1, address2, city, state, and zip* in a *books* example, you would create a separate publishers table.  (Addresses would be more complex once you factor international publishers in.)
* Assign or create new primary keys.
  + Every table must have a primary key, so add publisher ID to the new tables.
* Create the requisite foreign keys that link any of the relationships.  Going with a minimal version of a hypothetical *books* database, one new table is created for storing the publisher’s information.  Finally, add a *publisher ID* to the books table.  This effectively links each book to its publisher.

**Important Tips:**  
Despite the existence of set rules for how to normalize a database, two different people could normalize the same example in slightly different ways. Database design does allow for personal preference and interpretations. The important thing is that a database has no clear and obvious NF violations. Any NF violation will likely lead to problems down the road.

Reviewing the Design

After walking through the normalization process, it’s best to review the design one more time.  You want to make sure that the database stores all the data you may ever need.  Often the creation of new tables, thanks to normalization, implies additional information to record.  For example, although the original focus of the *cinema* database was on the movies, now that there are separate *actors* and *directors* tables, additional facts about those people could be reflected in those tables.

With that in mind, although there are many additional columns that could be added to the *forum* database, particularly regarding the user, one more field should be added to the *messages* table.  Because one message might be a reply to another, some method of indicating that relationship is required.  One solution is to add a *parent\_id* column to *messages* (to reflect a message hierarchy, the *parent\_id* column is added to *messages*).  If a message is a reply, its *parent\_id* value will be the *message\_id* of the original message (so *message\_id* is acting as a foreign key to this same table). If a message has a *parent\_id* of 0, then it’s a new thread, not a reply).

After making any changes to the tables, you must run through the normal forms one more time to ensure that the database is still normalized.  Finally, choose the column types and names, per the concepts we learned from last week.  Note that every integer column is UNSIGNED (noted as UN), the three primary key columns are also designated as AUTO\_INCREMENT (noted as AI), and every column is set as NOT NULL (noted as NN).

# Overruling Normalization

### Overruling Normalization

As much as ensuring that a database is in 3NF will help guarantee reliability and viability, you won’t fully normalize every database with which you work.  Before undermining the proper methods, though, understand that doing so may have devastating long-term consequences.

The two primary reasons to overrule normalization are convenience and performance.  Fewer tables are easier to manipulate and comprehend than more tables.  Further, because of their more intricate nature, normalized databases will most likely be slower for updating, retrieving data from, and modifying. Normalization, in short, is a trade-off between data integrity/scalability and simplicity/speed.  On the other hand, there are ways to improve your database’s performance but few to remedy corrupted data that can result from poor design.

The previous page included an example where normalization is ignored: a message’s post date and time is stored in one field.  As mentioned, because MySQL is so good with dates, there are no dangers to this approach.  Another situation where you would overrule normalization is a table that stored a person’s preference for a certain setting, such as “receive notifications.”  If stored as just Y/N or Yes/No (instead of linking to an answers table), there would be many repeating values.  But that is fine in this case, since those labels are stable values, not likely to change over time (i.e., it’s unlikely that a third option will be invented, or that “Yes” will be renamed, forcing a mass update of half the records in the table).

Practice and experience will teach you how best to model your database, but do try to err on the side of abiding by the normal forms, particularly as you are still mastering the concept.