Chapter 4. Working with Database Structures

This chapter shows you how to create your own databases, add and remove structures such as tables and indexes, and make choices about column types in your tables. It focuses on the syntax and features of SQL, and not the semantics of conceiving, specifying, and refining a database design; you'll find an introductory description of database design techniques in Chapter 2. To work through this chapter, you need to understand how to work with an existing database and its tables, as discussed in Chapter 3.

This chapter lists the structures in the sample `sakila` database. If you followed the instructions for loading the database in "Entity Relationship Modeling Examples", you'll already have the database available and know how to restore it after you've modified its structures.

When you finish this chapter, you'll have all the basics required to create, modify, and delete database structures. Together with the techniques you learned in Chapter 3, you'll have the skills to carry out a wide range of basic operations. Chapters 5 and 7 cover skills that allow you to do more advanced operations with MySQL.

Creating and Using Databases

When you've finished designing a database, the first practical step to take with MySQL is to create it. You do this with the `CREATE DATABASE` statement. Suppose you want to create a database with the name `lucy`. Here's the statement you'd type:

```
mysql> CREATE DATABASE lucy;

Query OK, 1 row affected (0.10 sec)
```

We assume here that you know how to connect using the MySQL client, as described in Chapter 1. We also assume that you're able to connect as the root user or as another user who can create, delete, and modify structures (you'll find a detailed discussion on user privileges in Chapter 8). Note that when you create the database, MySQL says that one row was affected. This isn't in fact a normal row in any specific database, but a new entry added to the list that you see with the `SHOW DATABASES` command.

Once you've created the database, the next step is to use it—that is, choose it as the database you're working with. You do this with the MySQL `USE` command:

```
mysql> USE lucy;

Database changed
```

This command must be entered on one line and need not be terminated with a semicolon, though we usually do so automatically through habit. Once you've used (selected) the database, you can start creating tables, indexes, and other structures using the steps discussed in the next section.

Before we move on to creating other structures, let's discuss a few features and limitations of creating databases. First, let's see what happens if you try to create a database that already exists:

```
mysql> CREATE DATABASE lucy;

ERROR 1007 (HY000): Can't create database 'lucy'; datab
```

You can avoid this error by adding the `IF NOT EXISTS` keyword phrase to the statement:

```
mysql> CREATE DATABASE IF NOT EXISTS lucy;

Query OK, 0 rows affected (0.00 sec)
```

You can see that MySQL didn't complain, but it didn't do anything either: the `0 rows affected` message indicates that no data was changed. This addition is useful when you're adding SQL statements to a script: it prevents the script from aborting on error.

Let's look at how to choose database names and use character case. Database names define physical directory (or folder) names on disk. On some operating systems, directory names are case-sensitive; on others, case doesn't matter. For example, Unix-like systems such as Linux and macOS are typically case-sensitive, whereas Windows isn't. The result is that database names have the same restrictions: when case matters to the operating system, it matters to MySQL. For example, on a Linux machine, `LUCY`, `lucy`, and `Lucy` are different database names; on Windows, they refer to just one database. Using incorrect capitalization under Linux or macOS will cause MySQL to complain:

```
mysql> SELECT SaKilA.AcTor_id FROM ACTor;

ERROR 1146 (42S02): Table 'sakila.ACTor' doesn't exist
```

But under Windows, this will normally work.

TIP

To make your SQL machine-independent, we recommend that you consistently use lowercase names for databases (and for tables, columns, aliases, and indexes). That's not a requirement, though, and as earlier examples in this book have demonstrated, you're welcome to use whatever naming convention you are comfortable with. Just be consistent and remember how MySQL behaves on different OSs.

This behavior is controlled by the `lower_case_table_names` parameter. If it's set to `0`, table names are stored as specified, and comparisons are case-sensitive. If it's set to `1`, table names are stored in lowercase on disk, and comparisons are not case-sensitive. If this parameter is set to `2`, table names are stored as given but compared in lowercase. On Windows, the default value is `1`. On macOS, the default is `2`. On Linux, a value of `2` is not supported; the server forces the value to `0` instead.

There are other restrictions on database names. They can be at most 64 characters in length. You also shouldn't use MySQL reserved words, such as `SELECT`, `FROM`, and `USE`, as names for structures; these can confuse the MySQL parser, making it impossible to interpret the meaning of your statements. You can get around this restriction by enclosing the reserved word in backticks (`), but it's more trouble remembering to do so than it's worth. In addition, you can't use certain characters in the names—specifically, the forward slash, backward slash, semicolon, and period characters—and a database name can't end in whitespace. Again, the use of these characters confuses the MySQL parser and can result in unpredictable behavior. For example, here's what happens when you insert a semicolon into a database name:

```
mysql> CREATE DATABASE IF NOT EXISTS lu;cy;


Query OK, 1 row affected (0.00 sec)

ERROR 1064 (42000): You have an error in your SQL synta
that corresponds to your MySQL server version for the r
near 'cy' at line 1
```

Since more than one SQL statement can be on a single line, the result is that a database `lu` is created, and then an error is generated by the very short, unexpected SQL statement `cy;`. If you really want to create a database with a semicolon in its name, you can do that with backticks:

```
mysql> CREATE DATABASE IF NOT EXISTS `lu;cy`;


Query OK, 1 row affected (0.01 sec)
```

And you can see that you now have two new databases:

```
mysql> SHOW DATABASES LIKE `lu%`;


+----------------+
| Database (lu%) |
+----------------+
```

```
| lu             |
| lu;cy          |
+----------------+
2 rows in set (0.01 sec)
```

Creating Tables

This section covers topics on table structures. We show you how to:

- Create tables, through introductory examples.
- Choose names for tables and table-related structures.
- Understand and choose column types.
- Understand and choose keys and indexes.
- Use the proprietary MySQL `AUTO_INCREMENT` feature.

When you finish this section, you'll have completed all of the basic material on creating database structures; the remainder of this chapter covers the sample `sakila` database and how to alter and remove existing structures.

Basics

For the examples in this section, we'll assume that the database `sakila` hasn't yet been created. If you want to follow along with the examples and you have already loaded the database, you can drop it for this section and reload it later; dropping it removes the database, its tables, and all of the data, but the original is easy to restore by following the steps in "Entity Relationship Modeling Examples". Here's how you drop it temporarily:

```
mysql> DROP DATABASE sakila;


Query OK, 23 rows affected (0.06 sec)
```

The `DROP` statement is discussed further at the end of this chapter in "Deleting Structures".

To begin, create the database `sakila` using the statement:

```
mysql> CREATE DATABASE sakila;


Query OK, 1 row affected (0.00 sec)
```

Then select the database with:

```
mysql> USE sakila;


Database changed
```

We're now ready to begin creating the tables that will hold our data. Let's create a table to hold actor details. For now, we're going to have a simplified structure, and we'll add more complexity later. Here's the statement we use:

```
mysql> CREATE TABLE actor (
    -> actor_id SMALLINT UNSIGNED NOT NULL DEFAULT 0,
    -> first_name VARCHAR(45) DEFAULT NULL,
    -> last_name VARCHAR(45),
    -> last_update TIMESTAMP,
    -> PRIMARY KEY (actor_id)
    -> );
```

```
Query OK, 0 rows affected (0.01 sec)
```

Don't panic—even though MySQL reports that zero rows were affected, it created the table:

```
mysql> SHOW tables;


+------------------+
| Tables_in_sakila |
+------------------+
| actor            |
+------------------+
1 row in set (0.01 sec)
```

Let's consider all this in detail. The `CREATE TABLE` command has three major sections:

1. The `CREATE TABLE` statement, which is followed by the table name to create. In this example, it's `actor`.
2. A list of one or more columns to be added to the table. In this example, we've added quite a few: `actor_id SMALLINT UNSIGNED NOT NULL DEFAULT 0`, `first_name VARCHAR(45) DEFAULT NULL`, `last_name VARCHAR(45)`, and `last_update TIMESTAMP`. We'll discuss these in a moment.
3. Optional key definitions. In this example, we've defined a single key: `PRIMARY KEY (actor_id)`. We'll discuss keys and indexes in detail later in this chapter.

Notice that the `CREATE TABLE` component is followed by an opening parenthesis that's matched by a closing parenthesis at the end of the statement. Notice also that the other components are separated by commas. There are other elements that you can add to a `CREATE TABLE` statement, and we'll discuss some in a moment.

Let's discuss the column specifications. The basic syntax is as follows: *`name type`* `[NOT NULL | NULL] [DEFAULT` *`value`*`]`. The *`name`* field is the column name, and it has the same limitations as database names, as discussed in the previous section. It can be at most 64 characters in length, backward and forward slashes aren't allowed, periods aren't allowed, it can't end in whitespace, and case sensitivity is dependent on the underlying operating system. The *`type`* field defines how and what is stored in the column; for example, we've seen that it can be set to `VARCHAR` for strings, `SMALLINT` for numbers, or `TIMESTAMP` for a date and time.

If you specify `NOT NULL`, a row isn't valid without a value for the column; if you specify `NULL` or omit this clause, a row can exist without a value for the column. If you specify a *`value`* with the `DEFAULT` clause, it'll be used to populate the column when you don't otherwise provide data; this is particularly useful when you frequently reuse a default value such as a country name. The *`value`* must be a constant (such as `0`, `"cat"`, or `20060812045623`), except if the column is of the type `TIMESTAMP`. Types are discussed in detail in "Column Types".

The `NOT NULL` and `DEFAULT` features can be used together. If you specify `NOT NULL` and add a `DEFAULT` value, the default is used when you don't provide a value for the column. Sometimes, this works fine:

```
mysql> INSERT INTO actor(first_name) VALUES ('John');
```

```
Query OK, 1 row affected (0.01 sec)
```

And sometimes it doesn't:

```
mysql> INSERT INTO actor(first_name) VALUES ('Elisabeth
```

```
ERROR 1062 (23000): Duplicate entry '0' for key 'actor.
```

Whether it works or not is dependent on the underlying constraints and conditions of the database: in this example, `actor_id` has a default value of `0`, but it's also the primary key. Having two rows with the same primary key value isn't permitted, and so the second attempt to insert a row with no values (and a resulting primary key value of `0`) fails. We discuss primary keys in detail in "Keys and Indexes".

Column names have fewer restrictions than database and table names. What's more, the names are case-insensitive and portable across all platforms. All characters are allowed in column names, though if you want terminate them with whitespace or include periods or other special characters, such as a semicolon or dash, you'll need to enclose the name in backticks (`). Again, we recommend that you consistently choose lowercase names for developer-driven choices (such as database, alias, and table names) and avoid characters that require you to remember to use backticks.

Naming columns and other database objects is something of a personal preference when starting anew (you can get some inspiration by looking at the example databases) or a matter of following standards when working on an existing codebase. In general, aim to avoid repetition: in a table named `actor`, use the column name `first_name` rather than `actor_first_name`, which would look redundant when preceded by the

table name in a complex query (`actor.actor_first_name` versus `actor.first_name`). An exception to this is when using the ubiquitous `id` column name; either avoid using this or prepend the table name for clarity (e.g., `actor_id`). It's good practice to use the underscore character to separate words. You could use another character, like a dash or slash, but you'd have to remember to enclose the names with backticks (e.g., `actor-id`). You can also omit the word-separating formatting altogether, but "CamelCase" is arguably harder to read. As with database and table names, the longest permitted length for a column name is 64 characters.

Collation and Character Sets

When you're comparing or sorting strings, how MySQL evaluates the result depends on the character set and collation used. Character sets, or charsets, define what characters can be stored; for example, you may need to store non-English characters such as ю or ü. A collation defines how strings are ordered, and there are different collations for different languages: for example, the position of the character ü in the alphabet is different in two German orderings, and different again in Swedish and Finnish. Because not everyone wants to store English strings, it's important that a database server be able to manage non-English characters and different ways of sorting characters.

We understand that discussion of collations and charsets may feel to be too advanced when you're just starting out learning MySQL. We also think, however, that these are topics worth covering, as mismatched charsets and collations may result in unexpected situations including loss of data and incorrect query results. If you prefer, you can skip this section and some of the later discussion in this chapter and come back to these topics when you want to learn about them specifically. That won't affect your understanding of other material in this book.

In our previous string-comparison examples, we ignored the collation and charset issue and just let MySQL use its defaults. In versions of MySQL prior to 8.0, the default character set is `latin1` , and the default collation is `latin1_swedish_ci` . MySQL 8.0 changed the defaults, and now the default charset is `utf8mb4` , and the default collation is `utf8mb4_0900_ai_ci` . MySQL can be configured to use different character sets and collation orders at the connection, database, table, and column levels. The outputs shown here are from MySQL 8.0.

You can list the character sets available on your server with the `SHOW CHARACTER SET` command. This shows a short description of each character set, its default collation, and the maximum number of bytes used for each character in that character set:

```
mysql> SHOW CHARACTER SET;
```

```
+----------+---------------------------------+---------
| Charset  | Description                     | Default
+----------+---------------------------------+---------
| armscii8 | ARMSCII-8 Armenian              | armscii8
| ascii    | US ASCII                        | ascii_ge
| big5     | Big5 Traditional Chinese        | big5_chi
| binary   | Binary pseudo charset           | binary
| cp1250   | Windows Central European        | cp1250_g
| cp1251   | Windows Cyrillic                | cp1251_g
| ...
| ujis     | EUC-JP Japanese                 | ujis_jap
| utf16    | UTF-16 Unicode                  | utf16_ge
| utf16le  | UTF-16LE Unicode                | utf16le_
| utf32    | UTF-32 Unicode                  | utf32_ge
| utf8     | UTF-8 Unicode                   | utf8_ger
| utf8mb4  | UTF-8 Unicode                   | utf8mb4_
+----------+---------------------------------+---------
41 rows in set (0.00 sec)
```

For example, the `latin1` character set is actually the Windows code page 1252 character set that supports West European languages. The default collation for this character set is `latin1_swedish_ci`, which follows Swedish conventions to sort accented characters (English is handled as you'd expect). This collation is case-insensitive, as indicated by the letters `ci`. Finally, each character takes up 1 byte. By comparison, if you use the default `utf8mb4` character set, each character will take up to 4 bytes of storage. Sometimes, it makes sense to change the default. For example, there's no reason to store base64-encoded data (which, by definition, is ASCII) in `utf8mb4`.

Similarly, you can list the collation orders and the character sets they apply to:

```
mysql> SHOW COLLATION;
```

```
+--------------------+----------+-----+---------+...+-
| Collation          | Charset  | Id  | Default |...|
+--------------------+----------+-----+---------+...+-
| armscii8_bin       | armscii8 |  64 |         |...|
| armscii8_general_ci| armscii8 |  32 | Yes     |...|
| ascii_bin          | ascii    |  65 |         |...|
| ascii_general_ci   | ascii    |  11 | Yes     |...|
| ...                                           |...|
| utf8mb4_0900_ai_ci | utf8mb4  | 255 | Yes     |...|
| utf8mb4_0900_as_ci | utf8mb4  | 305 |         |...|
| utf8mb4_0900_as_cs | utf8mb4  | 278 |         |...|
| utf8mb4_0900_bin   | utf8mb4  | 309 |         |...|
| ...                                           |...|
| utf8_unicode_ci    | utf8     | 192 |         |...|
| utf8_vietnamese_ci | utf8     | 215 |         |...|
+--------------------+----------+-----+---------+...+-
272 rows in set (0.02 sec)
```

NOTE

The number of character sets and collations available depends on how the MySQL server was built and packaged. The examples we show are from a default MySQL 8.0 installation, and the same numbers can be seen on Linux and Windows. MariaDB 10.5, however, has 322 collations but 40 character sets.

You can see the current defaults on your server as follows:

```
mysql> SHOW VARIABLES LIKE 'c%';
```

```
+--------------------------+----------------------------
| Variable_name            | Value
+--------------------------+----------------------------
| ...
| character_set_client     | utf8mb4
| character_set_connection | utf8mb4
| character_set_database   | utf8mb4
| character_set_filesystem | binary
| character_set_results    | utf8mb4
| character_set_server     | utf8mb4
| character_set_system     | utf8
```

```
| character_sets_dir       | /usr/share/mysql-8.0/chars
| ...
| collation_connection     | utf8mb4_0900_ai_ci
| collation_database       | utf8mb4_0900_ai_ci
| collation_server         | utf8mb4_0900_ai_ci
| ...
+--------------------------+--------------------------
21 rows in set (0.00 sec)
```

When you're creating a database, you can set the default character set and sort order for the database and its tables. For example, if you want to use the `utf8mb4` character set and the `utf8mb4_ru_0900_as_cs` (case-sensitive) collation order, you would write:

```
mysql> CREATE DATABASE rose DEFAULT CHARACTER SET utf8m
    -> COLLATE utf8mb4_ru_0900_as_cs;
```

```
Query OK, 1 row affected (0.00 sec)
```

Usually, there's no need to do this if you've installed MySQL correctly for your language and region and if you're not planning on internationalizing your application. With `utf8mb4` being the default since MySQL 8.0, there's even less need to change the charset. You can also control the character set and collation for individual tables or columns, but we won't go into the details of how to do that here. We will discuss how collations affect string types in "String types".

Other Features

This section briefly describes other features of the `CREATE TABLE` statement. It includes an example using the `IF NOT EXISTS` feature, and a list of advanced features and where to find more about them in this book. The statement shown is the full representation of the table taken from the `sakila` database, unlike the previous simplified example.

You can use the `IF NOT EXISTS` keyword phrase when creating a table, and it works much as it does for databases. Here's an example that won't report an error even when the `actor` table exists:

```
mysql> CREATE TABLE IF NOT EXISTS actor (
    -> actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREME
    -> first_name VARCHAR(45) NOT NULL,
    -> last_name VARCHAR(45) NOT NULL,
    -> last_update TIMESTAMP NOT NULL DEFAULT
    -> CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    -> PRIMARY KEY  (actor_id),
    -> KEY idx_actor_last_name (last_name));
```

```
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

You can see that zero rows are affected, and a warning is reported. Let's take a look:

```
mysql> SHOW WARNINGS;
```

```
+-------+------+-----------------------------+
| Level | Code | Message                     |
+-------+------+-----------------------------+
| Note  | 1050 | Table 'actor' already exists |
+-------+------+-----------------------------+
1 row in set (0.01 sec)
```

There are a wide range of additional features you can add to a `CREATE TABLE` statement, only a few of which are present in this example. Many of these are advanced and aren't discussed in this book, but you can find more information in the MySQL Reference Manual in the section on the `CREATE TABLE` statement. These additional features include the following:

The `AUTO_INCREMENT` feature for numeric columns

This feature allows you to automatically create unique identifiers for a table. We discuss it in detail in "The AUTO_INCREMENT Feature".

Column comments

You can add a comment to a column; this is displayed when you use the `SHOW CREATE TABLE` command that we discuss later in this section.

Foreign key constraints

You can tell MySQL to check whether data in one or more columns matches data in another table. For example, the `sakila` database has a foreign key constraint on the `city_id` column of the `address` table, referring to the `city` table's `city_id` column. That means it's impossible to have an address in a city not present in the `city` table. We introduced foreign key constraints in Chapter 2, and we'll take a look at what engines support foreign key constraints in "Alternative Storage Engines". Not every storage engine in MySQL supports foreign keys.

Creating temporary tables

If you create a table using the keyword phrase `CREATE TEMPORARY TABLE`, it'll be removed (dropped) when the connection is closed. This is useful for copying and reformatting data because you don't have to remember to clean up. Sometimes temporary tables are also used as an optimization to hold some intermediate data.

Advanced table options

You can control a wide range of features of the table using table options. These include the starting value of `AUTO_INCREMENT`, the way indexes and rows are stored, and options to override the information that the MySQL query optimizer gathers from the table. It's also possible to specify *generated columns*, containing data like sum of two other columns, as well as indexes on such columns.

Control over index structures

Some storage engines in MySQL allow you to specify and control what type of internal structure—such as a B-tree or hash table—MySQL uses for its indexes. You can also tell MySQL that you want a full-text or spatial data index on a column, allowing special types of search.

Partitioning

MySQL supports different partitioning strategies, which you can select at table creation time or later. We will not be covering partitioning in this book.

You can see the statement used to create a table using the `SHOW CREATE TABLE` statement introduced in Chapter 3. This often shows you output that includes some of the advanced features we've just discussed; the output rarely matches what you actually typed to create the table. Here's an example for the `actor` table:

```
mysql> SHOW CREATE TABLE actor\G


*************************** 1. row ***************************
       Table: actor
Create Table: CREATE TABLE `actor` (
  `actor_id` smallint unsigned NOT NULL AUTO_INCREMENT,
  `first_name` varchar(45) NOT NULL,
  `last_name` varchar(45) NOT NULL,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIME
        ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`actor_id`),
  KEY `idx_actor_last_name` (`last_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
        COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

You'll notice that the output includes content added by MySQL that wasn't in our original `CREATE TABLE` statement:

- The names of the table and columns are enclosed in backticks. This isn't necessary, but it does avoid any parsing problems that can be caused by the use of reserved words and special characters, as discussed previously.
- An additional default `ENGINE` clause is included, which explicitly states the table type that should be used. The setting in a default installation of MySQL is `InnoDB`, so it has no effect in this example.
- An additional `DEFAULT CHARSET` clause is included, which tells MySQL what character set is used by the columns in the table. Again, this has no effect in a default installation.

Column Types

This section describes the column types you can use in MySQL. It explains when each should be used and any limitations it has. The types are grouped by

their purpose. We'll cover the most widely used data types and mention more advanced or less used types in passing. That doesn't mean they have no use, but consider learning about them as an exercise. Most likely, you will not remember each of the data types and its particular intricacies, and that's okay. It's worth rereading this chapter later and consulting the MySQL documentation on the topic to keep your knowledge up-to-date.

Integer types

We will start with numeric data types, and more specifically with integer types, or the types holding specific whole numbers. First, the two most popular integer types:

`INT[(width)] [UNSIGNED] [ZEROFILL]`

This is the most commonly used numeric type; it stores integer (whole number) values in the range –2,147,483,648 to 2,147,483,647. If the optional `UNSIGNED` keyword is added, the range is 0 to 4,294,967,295. The keyword `INT` is short for `INTEGER`, and they can be used interchangeably. An `INT` column requires 4 bytes of storage space.

`INT`, as well as other integer types, has two properties specific to MySQL: optional *`width`* and `ZEROFILL` arguments. They are not part of a SQL standard, and as of MySQL 8.0 are deprecated. Still, you will surely notice them in a lot of codebases, so we will briefly cover both of them.

The *`width`* parameter specifies the display width, which can be read by applications as part of the column metadata. Unlike parameters in a similar position for other data types, this parameter has no effect on the storage characteristics of a particular integer type and does not constrain the usable range of values. `INT(4)` and `INT(32)` are the same for the purpose of data storage.

`ZEROFILL` is an additional argument that is used to left-pad the values with zeros up to the length specified by the *`width`* parameter. If you use `ZEROFILL`, MySQL automatically adds `UNSIGNED` to the declaration (since zero-filling makes sense only in the context of positive numbers).

In a few applications where `ZEROFILL` and *`width`* are useful, the `LPAD()` function can be used, or numbers can be stored formatted in `CHAR` columns.

`BIGINT[(width)] [UNSIGNED] [ZEROFILL]`

In the world of growing data sizes, having tables with counts of rows in the billions is getting more common. Even simple `id` -type columns might need a wider range than a regular `INT` provides. `BIGINT` solves that problem. It is a large integer type with a signed range of –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. An unsigned `BIGINT` can store numbers from 0 to 18,446,744,073,709,551,615. Columns of this type will require 8 bytes of storage.

Internally, all calculations within MySQL are done using signed `BIGINT` or `DOUBLE` values. The important consequence of that is that you should be very careful when dealing with extremely large numbers. There are two issues to be aware of. First, unsigned big integers larger than 9,223,372,036,854,775,807 should only be used with bit functions. Second, if the result of an arithmetical operation is larger than 9,223,372,036,854,775,807, unexpected results might be observed.

For example:

```
mysql> CREATE TABLE test_bigint (id BIGINT UNSIGN
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> INSERT INTO test_bigint VALUES (1844674407
```

```
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO test_bigint VALUES (1844674407
```

```
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO test_bigint VALUES (1844674407
```

```
ERROR 1690 (22003): BIGINT value
is out of range in '(184467440737095516 * 100)'
```

Even though 18,446,744,073,709,551,600 is less than 18,446,744,073,709,551,615, since a signed `BIGINT` is used for multiplication internally, the out-of-range error is observed.

TIP

The `SERIAL` data type can be used as an alias for `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE`. Unless you must optimize for data size and performance, consider using `SERIAL` for your `id`-like columns. Even the `UNSIGNED INT` can run out of range much quicker than you'd expect, and often at the worst possible time.

Keep in mind that although it's possible to store every integer as a `BIGINT`, that's wasteful in terms of storage space. Moreover, as we discussed, the *`width`* parameter doesn't constrain the range of values. To save space and put constraints on stored values, you should use different integer types:

`SMALLINT[(width)] [UNSIGNED] [ZEROFILL]`

Stores small integers, with a range from –32,768 to 32,767 signed and from 0 to 65,535 unsigned. It takes 2 bytes of storage.

`TINYINT[(width)] [UNSIGNED] [ZEROFILL]`

The smallest numeric data type, storing even smaller integers. The range of this type is –128 to 127 signed and 0 to 255 unsigned. It takes only 1 byte of storage.

`BOOL[(width)]`

Short for `BOOLEAN`, and a synonym for `TINYINT(1)`. Usually, Boolean types accept only two values: true or false. However, since

`BOOL` in MySQL is an integer type, you can store values from –128 to 127 in a `BOOL`. The value 0 will be treated as false, and all nonzero values as true. It's also possible to use special `true` and `false` aliases for 1 and 0, respectively. Here are some examples:

```
mysql> CREATE TABLE test_bool (i BOOL);

Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO test_bool VALUES (true),(false
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> INSERT INTO test_bool VALUES (1),(0),(-128
Query OK, 4 rows affected (0.02 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> SELECT i, IF(i,'true','false') FROM test_b
+------+----------------------+
| i    | IF(i,'true','false') |
+------+----------------------+
|    1 | true                 |
|    0 | false                |
|    1 | true                 |
|    0 | false                |
| -128 | true                 |
|  127 | true                 |
+------+----------------------+
6 rows in set (0.01 sec)
```

MEDIUMINT[(width)] [UNSIGNED] [ZEROFILL]

Stores values in the signed range of –8,388,608 to 8,388,607 and the unsigned range of 0 to 16,777,215. It takes 3 bytes of storage.

BIT[(M)]

Special type used to store bit values. *M* specifies the number of bits per value and defaults to 1 if omitted. MySQL uses a `b'value` syntax for binary values.

Fixed-point types

The `DECIMAL` and `NUMERIC` data types in MySQL are the same, so although we will only describe `DECIMAL` here, this description also applies to `NUMERIC` . The main difference between fixed-point and floating-point types is precision. For fixed-point types, the value retrieved is identical to the value stored; this isn't always the case with types that contain decimal points, such as the `FLOAT` and `DOUBLE` types described later. That is the most important property of the `DECIMAL` data type, which is a commonly used numeric type in MySQL:

DECIMAL[(width[,decimals])] [UNSIGNED] [ZEROFILL]

Stores a fixed-point number such as a salary or distance, with a total of *width* digits of which some smaller number are *decimals* that follow a decimal point. For example, a column declared as `price DECIMAL(6,2)` can be used to store values in the range –9,999.99 to 9,999.99. `price DECIMAL(10,4)` would allow values like 123,456.1234.

Prior to MySQL 5.7, if you tried to store a value outside this range, it would be stored as the closest value in the allowed range. For example, 100 would be stored as 99.99, and –100 would be stored as –99.99. Starting with version 5.7.5, however, the default SQL mode includes the mode `STRICT_TRANS_TABLES` , which prohibits this and other unsafe behaviors. Using the old behavior is possible, but could result in data loss.

SQL modes are special settings that control the behavior of MySQL when it comes to queries. For example, they can restrict "unsafe" behavior or affect how queries are interpreted. For the purpose of learning MySQL, we recommend that you stick to the defaults, as they

are safe. Changing SQL modes may be required for compatibility with legacy applications across MySQL releases.

The *width* parameter is optional, and a value of 10 is assumed when it is omitted. The number of *decimals* is also optional, and when omitted, a value of 0 is assumed; the maximum value of *decimals* may not exceed the value of *width*. The maximum value of *width* is 65, and the maximum value of *decimals* is 30.

If you're storing only positive values, you can use the `UNSIGNED` keyword as described for `INT`. If you want zero-padding, use the `ZEROFILL` keyword for the same behavior as described for `INT`. The keyword `DECIMAL` has three identical, interchangeable alternatives: `DEC`, `NUMERIC`, and `FIXED`.

Values in `DECIMAL` columns are stored using a binary format. This format uses 4 bytes for every nine digits.

Floating-point types

In addition to the fixed-point `DECIMAL` type described in the previous section, there are two other types that support decimal points: `DOUBLE` (also known as `REAL`) and `FLOAT`. They're designed to store approximate numeric values rather than the exact values stored by `DECIMAL`.

Why would you want approximate values? The answer is that many numbers with a decimal point are approximations of real quantities. For example, suppose you earn $50,000 per annum and you want to store it as a monthly wage. When you convert this to a per-month amount, it's $4,166 plus 66 and 2/3 cents. If you store this as $4,166.67, it's not exact enough to convert to a yearly wage (since 12 multiplied by $4,166.67 is $50,000.04). However, if you store 2/3 with enough decimal places, it's a closer approximation. You'll find that it is accurate enough to correctly multiply to obtain the original value in a high-precision environment such as MySQL, using only a bit of rounding. That's where `DOUBLE` and `FLOAT` are useful: they let you store values such as 2/3 or pi with a large number of decimal places, allowing accurate approximate representations of exact quantities. You can later use the `ROUND()` function to restore the results to a given precision.

Let's continue the previous example using `DOUBLE`. Suppose you create a table as follows:

```
mysql> CREATE TABLE wage (monthly DOUBLE);

Query OK, 0 rows affected (0.09 sec)
```

You can now insert the monthly wage using:

```
mysql> INSERT INTO wage VALUES (50000/12);

Query OK, 1 row affected (0.00 sec)
```

And see what's stored:

```
mysql> SELECT * FROM wage;

+----------------+
| monthly        |
+----------------+
| 4166.666666666 |
+----------------+
1 row in set (0.00 sec)
```

However, when you multiply it to obtain a yearly value, you get a high-precision approximation:

```
mysql> SELECT monthly*12 FROM wage;

+--------------------+
| monthly*12         |
+--------------------+
| 49999.999999992004 |
+--------------------+
1 row in set (0.00 sec)
```

To get the original value back, you still need to perform rounding with the desired precision. For example, your business might require precision to five decimal places. In this case, you could restore the original value with:

```
mysql> SELECT ROUND(monthly*12,5) FROM wage;

+---------------------+
| ROUND(monthly*12,5) |
+---------------------+
|         50000.00000 |
+---------------------+
1 row in set (0.00 sec)
```

But precision to eight decimal places would not result in the original value:

```
mysql> SELECT ROUND(monthly*12,8) FROM wage;

+---------------------+
| ROUND(monthly*12,8) |
+---------------------+
|      49999.99999999 |
+---------------------+
1 row in set (0.00 sec)
```

It's important to understand the imprecise and approximate nature of floating-point data types.

Here are the details of the `FLOAT` and `DOUBLE` types:

`FLOAT[(width, decimals)] [UNSIGNED] [ZEROFILL]` or `FLOAT[(precision)] [UNSIGNED] [ZEROFILL]`

> Stores floating-point numbers. It has two optional syntaxes: the first allows an optional number of *`decimals`* and an optional display *`width`*, and the second allows an optional *`precision`* that controls the accuracy of the approximation measured in bits. Without parameters (the typical usage), the type stores small, 4-byte, single-precision floating-point values. When *`precision`* is between 0 and 24, the default behavior occurs. When *`precision`* is between 25 and 53, the type behaves like `DOUBLE`. The *`width`* parameter has no

effect on what is stored, only on what is displayed. The `UNSIGNED` and `ZEROFILL` options behave as for `INT`.

`DOUBLE[(width, decimals)] [UNSIGNED] [ZEROFILL]`

Stores floating-point numbers. It allows specification of an optional number of *`decimals`* and an optional display *`width`*. Without parameters (the typical usage), the type stores normal 8-byte, double-precision floating-point values. The *`width`* parameter has no effect on what is stored, only on what is displayed. The `UNSIGNED` and `ZEROFILL` options behave as for `INT`. The `DOUBLE` type has two identical synonyms: `REAL` and `DOUBLE PRECISION`.

String types

String data types are used to store text and, less obviously, binary data. MySQL supports the following string types:

`[NATIONAL] VARCHAR(width) [CHARACTER SET charset_name] [COLLATE collation_name]`

Probably the single most commonly used string type, `VARCHAR` stores variable-length strings up to a maximum *`width`*. The maximum value of *`width`* is 65,535 characters. Most of the information applicable to this type will apply to other string types as well.

The `CHAR` and `VARCHAR` types are very similar, but there are a few important distinctions. `VARCHAR` incurs one or two extra bytes of overhead to store the value of the string, depending on whether the value is smaller or larger than 255 bytes. Note that this size is different from the string length in characters, as certain characters might require up to 4 bytes of space. It might seem obvious, then, that `VARCHAR` is less efficient. However, that is not always true. As `VARCHAR` can store strings of arbitrary length (up to the *`width`* defined), shorter strings will require less storage space than a `CHAR` of similar length.

Another difference between `CHAR` and `VARCHAR` is their handling of trailing spaces. `VARCHAR` retains trailing spaces up to the specified column width and will truncate the excess, producing a warning. As will be shown later, `CHAR` values are right-padded to the column width, and the trailing spaces aren't preserved. For `VARCHAR`, trailing

spaces are significant unless they are trimmed and will count as unique values. Let's demonstrate:

```
mysql> CREATE TABLE test_varchar_trailing(d VARCH
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> INSERT INTO test_varchar_trailing VALUES (
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT d, LENGTH(d) FROM test_varchar_trai
```

```
+------+-----------+
| d    | LENGTH(d) |
+------+-----------+
| a    |         1 |
| a    |         2 |
+------+-----------+
2 rows in set (0.00 sec)
```

The second row we inserted has a trailing space, and since the *width* for column `d` is 2, that space counts toward the uniqueness of a row. If we try inserting a row with two trailing spaces, however:

```
mysql> INSERT INTO test_varchar_trailing VALUES (
```

```
ERROR 1062 (23000): Duplicate entry 'a '
for key 'test_varchar_trailing.d'
```

MySQL refuses to accept the new row. `VARCHAR(2)` implicitly truncates trailing spaces beyond the set *width*, so the value stored changes from `"a "` (with a double space after *a*) to `"a "` (with a

single space after *a*). Since we already have a row with such a value, a duplicate entry error is reported. This behavior for `VARCHAR` and `TEXT` can be controlled by changing the column collation. Some collations, like `latin1_bin`, have the `PAD SPACE` attribute, meaning that upon retrieval they are padded to the *`width`* with spaces. This doesn't affect storage, but does affect uniqueness checks as well as how the `GROUP BY` and `DISTINCT` operators work, which we'll discuss in Chapter 5. You can check whether a collation is `PAD SPACE` or `NO PAD` by running the `SHOW COLLATION` command, as we've shown in "Collation and Character Sets". Let's see the effect in action by creating a table with a `PAD SPACE` collation:

```
mysql> CREATE TABLE test_varchar_pad_collation(
    -> data VARCHAR(5) CHARACTER SET latin1
    -> COLLATE latin1_bin UNIQUE);
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> INSERT INTO test_varchar_pad_collation VAL
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO test_varchar_pad_collation VAL
```

```
ERROR 1062 (23000): Duplicate entry 'a '
for key 'test_varchar_pad_collation.data'
```

The `NO PAD` collation is a new addition of MySQL 8.0. In prior releases of MySQL, which you may still often see in use, every collation implicitly has the `PAD SPACE` attribute. Therefore, in MySQL 5.7 and prior releases, your only option to preserve trailing spaces is to use a binary type: `VARBINARY` or `BLOB`.

NOTE

Both the `CHAR` and `VARCHAR` data types disallow storage of values longer than *`width`*, unless strict SQL mode is disabled (i.e., if neither `STRICT_ALL_TABLES` or `STRICT_TRANS_TABLES` is enabled). With the protection disabled, values longer than *`width`* are truncated, and a warning is shown. We don't recommend enabling legacy behavior, as it might result in data loss.

Sorting and comparison of the `VARCHAR`, `CHAR`, and `TEXT` types happens according to the collation of the character set assigned. You can see that it is possible to specify the character set, as well as the collation for each individual string-type column. It's also possible to specify the `binary` character set, which effectively converts `VARCHAR` into `VARBINARY`. Don't mistake the `binary` charset for a `BINARY` attribute for a charset; the latter is a MySQL-only shorthand to specify a binary (`_bin`) collation.

What's more, it's possible to specify a collation directly in the `ORDER BY` clause. Available collations will depend on the character set of the column. Continuing with the `test_varchar_pad_collation` table, it's possible to store an ä symbol there and then see the effect collations make on the string ordering:

```
mysql> INSERT INTO test_varchar_pad_collation VAL
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM test_varchar_pad_collation
    -> ORDER BY data COLLATE latin1_german1_ci;
```

```
+------+
| data |
+------+
| a    |
| ä    |
| z    |
```

```
+------+
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM test_varchar_pad_collation
    -> ORDER BY data COLLATE latin1_swedish_ci;
```

```
+------+
| data |
+------+
| a    |
| z    |
| ä    |
+------+
3 rows in set (0.00 sec)
```

The `NATIONAL` (or its equivalent short form, `NCHAR`) attribute is a standard SQL way to specify that a string-type column must use a predefined character set. MySQL uses `utf8` as this charset. It's important to note that MySQL 5.7 and 8.0 disagree on what exactly `utf8` is, however: the former uses it as an alias for `utf8mb3`, and the latter for `utf8mb4`. Thus, it is best to not use the `NATIONAL` attribute, as well as ambiguous aliases. The best practice with any text-related columns and data is to be as unambiguous and specific as possible.

`[NATIONAL] CHAR(width) [CHARACTER SET charset_name] [COLLATE collation_name]`

`CHAR` stores a fixed-length string (such as a name, address, or city) of length *`width`*. If a *`width`* is not provided, `CHAR(1)` is assumed. The maximum value of *`width`* is 255. As with `VARCHAR`, values in `CHAR` columns are always stored at the specified length. A single letter stored in a `CHAR(255)` column will take 255 bytes (in the `latin1` charset) and will be padded with spaces. The padding is removed when reading the data, unless the `PAD_CHAR_TO_FULL_LENGTH` SQL mode is enabled. It's worth mentioning again that this means that strings stored in `CHAR` columns will lose all of their trailing spaces.

In the past, the *`width`* of a `CHAR` column was often associated a size in bytes. That's not always the case now, and it's definitely not the case

by default. Multibyte character sets, such as the default `utf8mb4` in MySQL 8.0, can result in much larger values. InnoDB will actually encode fixed-length columns as variable-length columns if their maximum size exceeds 768 bytes. Thus, in MySQL 8.0, by default InnoDB will store a `CHAR(255)` column as it would a `VARCHAR` column. Here's an example:

```
mysql> CREATE TABLE test_char_length(
    ->   utf8char CHAR(10) CHARACTER SET utf8mb4
    -> , asciichar CHAR(10) CHARACTER SET binary
    -> );
```

```
Query OK, 0 rows affected (0.04 sec)
```

```
mysql> INSERT INTO test_char_length VALUES ('Plai
```

```
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO test_char_length VALUES ('的開
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT LENGTH(utf8char), LENGTH(asciichar)
```

```
+------------------+------------------+
| LENGTH(utf8char) | LENGTH(asciichar) |
+------------------+------------------+
|               10 |                10 |
|               15 |                10 |
+------------------+------------------+
2 rows in set (0.00 sec)
```

As the values are left-aligned and right-padded with spaces, and any trailing spaces aren't considered for `CHAR` at all, it's impossible to

compare strings consisting of spaces alone. If you find yourself in a situation in which that's important, `VARCHAR` is the data type to use.

`BINARY[(width)]` and `VARBINARY(width)`

These types are very similar to `CHAR` and `VARCHAR` but store binary strings. Binary strings have the special `binary` character set and collation, and sorting them is dependent on the numeric values of the bytes in the values stored. Instead of character strings, byte strings are stored. In the earlier discussion of `VARCHAR` we described the `binary` charset and `BINARY` attribute. Only the `binary` charset "converts" a `VARCHAR` or `CHAR` into its respective `BINARY` form. Applying the `BINARY` attribute to a charset will not change the fact that character strings are stored. Unlike with `VARCHAR` and `CHAR`, *`width`* here is exactly the number of bytes. When *`width`* is omitted for `BINARY`, it defaults to 1.

Like with `CHAR`, data in the `BINARY` column is padded on the right. However, being a binary data, it's padded using zero bytes, usually written as `0x00` or `\0`. `BINARY` treats a space as a significant character, not padding. If you need to store data that might end in zero bytes that are significant to you, use the `VARBINARY` or `BLOB` types.

It is important to keep the concept of binary strings in mind when working with both of these data types. Even though they'll accept strings, they aren't synonyms for data types using text strings. For example, you cannot change the case of the letters stored, as that concept doesn't really apply to binary data. That becomes quite clear when you consider the actual data stored. Let's look at an example:

```
mysql> CREATE TABLE test_binary_data (
    ->   d1 BINARY(16)
    -> , d2 VARBINARY(16)
    -> , d3 CHAR(16)
    -> , d4 VARCHAR(16)
    -> );

Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO test_binary_data VALUES (
    ->   'something'
```

```
    -> , 'something'
    -> , 'something'
    -> , 'something');

Query OK, 1 row affected (0.00 sec)

mysql> SELECT d1, d2, d3, d4 FROM test_binary_dat
```

```
*************************** 1. row **************
d1: 0x736F6D657468696E6700000000000000
d2: 0x736F6D657468696E67
d3: something
d4: something
1 row in set (0.00 sec)
```

```
mysql> SELECT UPPER(d2), UPPER(d4) FROM test_bina
```

```
*************************** 1. row **************
UPPER(d2): 0x736F6D657468696E67
UPPER(d4): SOMETHING
1 row in set (0.01 sec)
```

Note how the MySQL command-line client actually shows values of binary types in hex format. We believe that this is much better than the silent conversions that were performed prior to MySQL 8.0, which might've resulted in misunderstanding. To get the actual text data back, you have to explicitly cast the binary data to text:

```
mysql> SELECT CAST(d1 AS CHAR) d1t, CAST(d2 AS CH
    -> FROM test_binary_data;
```

```
+-----------------+-----------+
| d1t             | d2t       |
+-----------------+-----------+
| something       | something |
```

```
+------------------+-----------+
1 row in set (0.00 sec)
```

You can also see that `BINARY` padding was converted to spaces when casting was performed.

`BLOB[(width)]` and `TEXT[(width)] [CHARACTER SET charset_name] [COLLATE collation_name]`

`BLOB` and `TEXT` are commonly used data types for storing large data. You may think of `BLOB` as a `VARBINARY` holding as much data as you like, and the same for `TEXT` and `VARCHAR`. The `BLOB` and `TEXT` types can store up to 65,535 bytes or characters, respectively. As usual, note that multibyte charsets do exist. The *`width`* attribute is optional, and when it is specified, MySQL actually will change the `BLOB` or `TEXT` data type to whatever the smallest type capable of holding that amount of data is. For example, `BLOB(128)` will result in `TINYBLOB` being used:

```
mysql> CREATE TABLE test_blob(data BLOB(128));

Query OK, 0 rows affected (0.07 sec)

mysql> DESC test_blob;

+-------+----------+------+-----+---------+------
| Field | Type     | Null | Key | Default | Extra
+-------+----------+------+-----+---------+------
| data  | tinyblob | YES  |     | NULL    |
+-------+----------+------+-----+---------+------
1 row in set (0.00 sec)
```

For the `BLOB` type and related types, data is treated exactly as it would be in the case of `VARBINARY`. That is, no character set is assumed, and comparison and sorting are based on the numeric values of the actual bytes stored. For `TEXT`, you may specify the exact desired charset and collation. For both types and their variants, no padding is performed on `INSERT`, and no trimming is performed on `SELECT`, making them ideal for storing data exactly as it is. In addition, a `DEFAULT` clause is not permitted, and when an index is

created on a `BLOB` or `TEXT` column, a prefix must be defined limiting the length of the indexed values. We talk more about that in "Keys and Indexes".

One potential difference between `BLOB` and `TEXT` is their handling of trailing spaces. As we've shown already, `VARCHAR` and `TEXT` may pad strings depending on the collation used. `BLOB` and `VARBINARY` both use the `binary` character set with a single `binary` collation with no padding and are impervious to collation mixups and related issues. Sometimes, it can be a good choice to use these types for additional safety. In addition to that, prior to MySQL 8.0, these were the only types that preserved trailing spaces.

`TINYBLOB` and `TINYTEXT [CHARACTER SET charset_name] [COLLATE collation_name]`

These are identical to `BLOB` and `TEXT`, respectively, except that a maximum of 255 bytes or characters can be stored.

`MEDIUMBLOB` and `MEDIUMTEXT [CHARACTER SET charset_name] [COLLATE collation_name]`

These are identical to `BLOB` and `TEXT`, respectively, except that a maximum of 16,777,215 bytes or characters can be stored. The types `LONG` and `LONG VARCHAR` map to the `MEDIUMTEXT` data type for compatibility.

`LONGBLOB` and `LONGTEXT [CHARACTER SET charset_name] [COLLATE collation_name]`

These are identical to `BLOB` and `TEXT`, respectively, except that a maximum of 4 GB of data can be stored. Note that this is a hard limit even in case of `LONGTEXT`, and thus the number of characters in multibyte charsets can be less than 4,294,967,295. The effective maximum size of the data that can be stored by a client will be limited by the amount of available memory as well as the value of the `max_packet_size` variable, which defaults to 64 MiB.

`ENUM(value1[,value2[, …]]) [CHARACTER SET charset_name] [COLLATE collation_name]`

This type stores a list, or *enumeration*, of string values. A column of type `ENUM` can be set to a value from the list *`value1`*, *`value2`*, and so on, up to a maximum of 65,535 different values. While the values

are stored and retrieved as strings, what's stored in the database is an integer representation. The `ENUM` column can contain `NULL` values (stored as `NULL`), the empty string `''` (stored as `0`), or any of the valid elements (stored as `1`, `2`, `3`, and so on). You can prevent `NULL` values from being accepted by declaring the column as `NOT NULL` when creating the table.

This type offers a compact way of storing values from a list of predefined values, such as state or country names. Consider this example using fruit names; the name can be any one of the predefined values `Apple`, `Orange`, or `Pear` (in addition to `NULL` and the empty string):

```
mysql> CREATE TABLE fruits_enum
    -> (fruit_name ENUM('Apple', 'Orange', 'Pear'
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> INSERT INTO fruits_enum VALUES ('Apple');
```

```
Query OK, 1 row affected (0.00 sec)
```

If you try inserting a value that's not in the list, MySQL produces an error to tell you that it didn't store the data you asked:

```
mysql> INSERT INTO fruits_enum VALUES ('Banana');
```

```
ERROR 1265 (01000): Data truncated for column 'fr
```

A list of several allowed values isn't accepted either:

```
mysql> INSERT INTO fruits_enum VALUES ('Apple,Ora
```

```
ERROR 1265 (01000): Data truncated for column 'fr
```

Displaying the contents of the table, you can see that no invalid values were stored:

```
mysql> SELECT * FROM fruits_enum;
```

```
+------------+
| fruit_name |
+------------+
| Apple      |
+------------+
1 row in set (0.00 sec)
```

Earlier versions of MySQL produced a warning instead of an error and stored an empty string in place of an invalid value. That behavior can be enabled by disabling the default strict SQL mode. It's also possible to specify a default value other than the empty string:

```
mysql> CREATE TABLE new_fruits_enum
    -> (fruit_name ENUM('Apple', 'Orange', 'Pear'
    -> DEFAULT 'Pear');
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> INSERT INTO new_fruits_enum VALUES();
```

```
Query OK, 1 row affected (0.02 sec)
```

```
mysql> SELECT * FROM new_fruits_enum;
```

```
+------------+
| fruit_name |
+------------+
| Pear       |
```

```
+------------+
1 row in set (0.00 sec)
```

Here, not specifying a value results in the default value `Pear` being stored.

`SET( value1 [, value2 [, …]]) [CHARACTER SET charset_name] [COLLATE collation_name]`

This type stores a set of string values. A column of type `SET` can be set to zero or more values from the list *`value1`*, *`value2`*, and so on, up to a maximum of 64 different values. While the values are strings, what's stored in the database is an integer representation. `SET` differs from `ENUM` in that each row can store only one `ENUM` value in a column, but can store multiple `SET` values. This type is useful for storing a selection of choices from a list, such as user preferences. Consider this example using fruit names; the name can be any combination of the predefined values:

```
mysql> CREATE TABLE fruits_set
    -> ( fruit_name SET('Apple', 'Orange', 'Pear'
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
mysql> INSERT INTO fruits_set VALUES ('Apple');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO fruits_set VALUES ('Banana');
```

```
ERROR 1265 (01000): Data truncated for column 'fr
```

```
mysql> INSERT INTO fruits_set VALUES ('Apple,Oran
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM fruits_set;

+--------------+
| fruit_name   |
+--------------+
| Apple        |
| Apple,Orange |
+--------------+
2 rows in set (0.00 sec)
```

> Again, note that we can store multiple values from the set in a single field and that an empty string is stored for invalid input.

As with numeric types, we recommend that you always choose the smallest possible type to store values. For example, if you're storing a city name, use `CHAR` or `VARCHAR` rather than, say, the `TEXT` type. Having shorter columns helps keep your table size down, which in turns helps performance when the server has to search through a table.

Using a fixed size with the `CHAR` type is often faster than using a variable size with `VARCHAR`, since the MySQL server knows where each row starts and ends and can quickly skip over rows to find the one it needs. However, with fixed-length fields, any space that you don't use is wasted. For example, if you allow up to 40 characters in a city name, then `CHAR(40)` will always use up 40 characters, no matter how long the city name actually is. If you declare the city name to be `VARCHAR(40)`, then you'll use up only as much space as you need, plus 1 byte to store the name's length. If the average city name is 10 characters long, this means that using a variable-length field will take up on average 29 fewer bytes per entry. This can make a big difference if you're storing millions of addresses.

In general, if storage space is at a premium or you expect large variations in the length of strings that are to be stored, use a variable-length field; if performance is a priority, use a fixed-length field.

Date and time types

These types serve the purpose of storing particular timestamps, dates, or time ranges. Particular care should be taken when dealing with time zones. We will

try to explain the details, but it's worth rereading this section and the documentation later, when you need to actually work with time zones. The date and time types in MySQL are:

DATE

Stores and displays a date in the format *YYYY-MM-DD* for the range 1000-01-01 to 9999-12-31. Dates must always be input as year, month, day triples, but the format of the input can vary, as shown in the following examples:

YYYY-MM-DD or YY-MM-DD

It's optional whether you provide two-digit or four-digit years. We strongly recommend that you use the four-digit version to avoid confusion about the century. In practice, if you use the two-digit version, you'll find that 70 to 99 are interpreted as 1970 to 1999, and 00 to 69 are interpreted as 2000 to 2069.

YYYY/MM/DD, YYYY:MM:DD, YY-MM-DD, or other punctuated formats

MySQL allows any punctuation characters to separate the components of a date. We recommend using dashes and, again, avoiding two-digit years.

YYYY-M-D, YYYY-MM-D, or YYYY-M-DD

When punctuation is used (again, any punctuation character is allowed), single-digit days and months can be specified as such. For example, February 2, 2006, can be specified as `2006-2-2`. The two-digit year equivalents are available, but not recommended.

YYYYMMDD or YYMMDD

Punctuation can be omitted in both date styles, but the digit sequences must be six or eight digits in length.

You can also input a date by providing both a date and time in the formats described later for `DATETIME` and `TIMESTAMP`, but only the date component is stored in a `DATE` column. Regardless of the input type, the storage and display type is always *YYYY-MM-DD*. The *zero*

date `0000-00-00` is allowed in all versions and can be used to represent an unknown or dummy value. If an input date is out of range, the zero date is stored. However, only MySQL versions up to and including 5.6 allow that by default. Both 5.7 and 8.0 by default set SQL modes that prohibit this behavior: `STRICT_TRANS_TABLES`, `NO_ZERO_DATE`, and `NO_ZERO_IN_DATE`.

If you're using an older version of MySQL, we recommend that you add these modes to your current session:

```
mysql> SET sql_mode=CONCAT(@@sql_mode,
    -> ',STRICT_TRANS_TABLES',
    -> ',NO_ZERO_DATE', ',NO_ZERO_IN_DATE');
```

TIP

You can also set the `sql_mode` variable on a global server level and in the configuration file. This variable must list every mode you want to be enabled.

Here are some examples of inserting dates on a MySQL 8.0 server with default settings:

```
mysql> CREATE TABLE testdate (mydate DATE);

Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO testdate VALUES ('2020/02/0');

ERROR 1292 (22007): Incorrect date value: '2020/0
for column 'mydate' at row 1

mysql> INSERT INTO testdate VALUES ('2020/02/1');

Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO testdate VALUES ('2020/02/31')
```

```
ERROR 1292 (22007): Incorrect date value: '2020/0
for column 'mydate' at row 1
```

```
mysql> INSERT INTO testdate VALUES ('2020/02/100'
```

```
ERROR 1292 (22007): Incorrect date value: '2020/0
for column 'mydate' at row 1
```

Once `INSERT` statements are executed, the table will have the following data:

```
mysql> SELECT * FROM testdate;
```

```
+------------+
| mydate     |
+------------+
| 2020-02-01 |
+------------+
1 row in set (0.00 sec)
```

MySQL protected you from having “bad” data stored in your table. Sometimes you may need to preserve the actual input and manually process it later. You can do that by removing the aforementioned SQL modes from the list of modes in the `sql_mode` variable. In that case, after running the previous `INSERT` statements, you would end up with the following data:

```
mysql> SELECT * FROM testdate;
```

```
+------------+
| mydate     |
+------------+
```

```
| 2020-02-00 |
| 2020-02-01 |
| 0000-00-00 |
| 0000-00-00 |
+------------+
4 rows in set (0.01 sec)
```

Note again that the date is displayed in the *YYYY-MM-DD* format, regardless of how it was input.

TIME [fraction]

Stores a time in the format *HHH:MM:SS* for the range –838:59:59 to 838:59:59. This is useful for storing the duration of some activity. The values that can be stored are outside the range of the 24-hour clock to allow large differences between time values (up to 34 days, 22 hours, 59 minutes, and 59 seconds) to be computed and stored. *fraction* in TIME and other related data types specifies the fractional seconds precision in the range 0 to 6. The default value is 0, meaning that no fractional seconds are preserved.

Times must always be input in the order *days*, *hours*, *minutes*, *seconds*, using the following formats:

DD HH:MM:SS[.fraction], *HH:MM:SS[.fraction]*, *DD HH:MM*, *HH:MM*, *DD HH*, *or SS[.fraction]*

DD represents a one-digit or two-digit value of days in the range 0 to 34. The *DD* value is separated from the hour value, *HH*, by a space, while the other components are separated by a colon. Note that *MM:SS* is not a valid combination, since it cannot be disambiguated from *HH:MM*. If the TIME definition doesn't specify *fraction* or sets it to 0, inserting fractional seconds will result in values being rounded to the nearest second.

For example, if you insert 2 13:25:58.999999 into a TIME column with a *fraction* of 0, the value 61:25:59 is stored, since the sum of 2 days (48 hours) and 13 hours is 61 hours. Starting with MySQL 5.7, the default SQL mode set prohibits insertion of incorrect values. However, it is possible to enable the older behavior. Then, if you try inserting a value that's out of bounds, a warning is generated, and the value is

limited to the maximum time available. Similarly, if you try inserting an invalid value, a warning is generated and the value is set to zero. You can use the `SHOW WARNINGS` command to report the details of the warning generated by the previous SQL statement. Our recommendation is to stick to the default strict SQL mode. Unlike with the `DATE` type, there's seemingly no benefit to allowing incorrect `TIME` entries, apart from easier error management on the application side and maintaining legacy behaviors.

Let's try all these out in practice:

```
mysql> CREATE TABLE test_time(id SMALLINT, m
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> INSERT INTO test_time VALUES(1, "2 13
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO test_time VALUES(2, "35 1
```

```
ERROR 1292 (22007): Incorrect time value: '3
for column 'mytime' at row 1
```

```
mysql> INSERT INTO test_time VALUES(3, "900.
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM test_time;
```

```
+------+----------+
| id   | mytime   |
+------+----------+
|    1 | 61:25:59 |
|    3 | 00:09:00 |
+------+----------+
2 rows in set (0.00 sec)
```

`H:M:S` , and single-, double-, and triple-digit combinations

You can use different combinations of digits when inserting or updating data; MySQL converts them into the internal time format and displays them consistently. For example, `1:1:3` is equivalent to `01:01:03` . Different numbers of digits can be mixed; for example, `1:12:3` is equivalent to `01:12:03` . Consider these examples:

```
mysql> CREATE TABLE mytime (testtime TIME);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
mysql> INSERT INTO mytime VALUES
    -> ('-1:1:1'), ('1:1:1'),
    -> ('1:23:45'), ('123:4:5'),
    -> ('123:45:6'), ('-123:45:6');
```

```
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM mytime;
```

```
+------------+
| testtime   |
+------------+
|  -01:01:01 |
|   01:01:01 |
|   01:23:45 |
|  123:04:05 |
|  123:45:06 |
```

```
| -123:45:06 |
+------------+
5 rows in set (0.01 sec)
```

Note that hours are shown with two digits for values within the range –99 to 99.

HHMMSS , MMSS , and SS

Punctuation can be omitted, but the digit sequences must be two, four, or six digits in length. Note that the rightmost pair of digits is always interpreted as a *SS* (seconds) value, the second rightmost pair (if present) as *MM* (minutes), and the third rightmost pair (if present) as *HH* (hours). The result is that a value such as `1222` is interpreted as 12 minutes and 22 seconds, not 12 hours and 22 minutes.

You can also input a time by providing both a date and time in the formats described for `DATETIME` and `TIMESTAMP` , but only the time component is stored in a `TIME` column. Regardless of the input type, the storage and display type is always *HH:MM:SS* . The *zero time* `00:00:00` can be used to represent an unknown or dummy value.

TIMESTAMP[(fraction)]

Stores and displays a date and time pair in the format *YYYY-MM-DD HH:MM:SS[.fraction][time zone offset]* for the range 1970-01-01 00:00:01.000000 to `2038-01-19 03:14:07.999999` . This type is very similar to the `DATETIME` type, but there are a few differences. Both types accept a time zone modifier to the input value MySQL 8.0, and both types will store and present the data in the same way to any client in the same time zone. However, the values in `TIMESTAMP` columns are internally always stored in the UTC time zone, making it possible to get a local time zone automatically for clients in different time zones. That on its own is a very important distinction to remember. Arguably, `TIMESTAMP` is more convenient to use when dealing with different time zones.

Prior to MySQL 5.6, only the `TIMESTAMP` type supported automatic initialization and update. Moreover, only a single such column per a given table could do that. However, starting with 5.6, both `TIMESTAMP` and `DATETIME` support the behaviors, and any number of columns can do so.

Values stored in a `TIMESTAMP` column always match the template *`YYYY-MM-DD HH:MM:SS[.fraction][time zone offset]`*, but the values can be provided in a wide range of formats:

`YYYY-MM-DD HH:MM:SS` *or* *`YY-MM-DD HH:MM:SS`*

The date and time components follow the same relaxed restrictions as the `DATE` and `TIME` components described previously. This includes allowance for any punctuation characters, including (unlike for `TIME`) flexibility in the punctuation used in the time component. For example, `⍰0⍰` is valid.

`YYYYMMDDHHMMSS` *or* *`YYMMDDHHMMSS`*

Punctuation can be omitted, but the string should be either 12 or 14 digits in length. We recommend using only the unambiguous 14-digit version, for the reasons discussed for the `DATE` type. You can specify values with other lengths without providing separators, but we don't recommend doing so.

Let's look at the automatic-update feature in more detail. You control this by adding the following attributes to the column definition when creating a table, or later, as we'll explain in "Altering Structures":

1. If you want the timestamp to be set only when a new row is inserted into the table, add `DEFAULT CURRENT_TIMESTAMP` to the end of the column declaration.
2. If you don't want a default timestamp but would like the current time to be used whenever the data in a row is updated, add `ON UPDATE CURRENT_TIMESTAMP` to the end of the column declaration.
3. If you want both of the above—that is, you want the timestamp set to the current time in each new row and whenever an existing row is modified—add `DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP` to the end of the column declaration.

If you do not specify `DEFAULT NULL` or `NULL` for a `TIMESTAMP` column, it will have `0` as the default value.

`YEAR[(4)]`

Stores a four-digit year in the range 1901 to 2155, as well as the *zero year*, 0000. Illegal values are converted to the zero year. You can input year values as either strings (such as `'2005'`) or integers (such as `2005`). The `YEAR` type requires 1 byte of storage space.

In earlier versions of MySQL, it was possible to specify the *`digits`* parameter, passing either `2` or `4`. The two-digit version stored values from 70 to 69, representing 1970 to 2069. MySQL 8.0 doesn't support the two-digit `YEAR` type, and specifying the *`digits`* parameter for display purposes is deprecated.

`DATETIME[(fraction)]`

Stores and displays a date and time pair in the format *`YYYY-MM-DD HH:MM:SS[.fraction][time zone offset]`* for the range `1000-01-01 00:00:00` to `9999-12-31 23:59:59`. As for `TIMESTAMP`, the value stored always matches the template *`YYYY-MM-DD HH:MM:SS`*, but the value can be input in the same formats listed in the `TIMESTAMP` description. If you assign only a date to a `DATETIME` column, the zero time `00:00:00` is assumed. If you assign only a time to a `DATETIME` column, the zero date `0000-00-00` is assumed. This type has the same automatic update features as `TIMESTAMP`. Unless the `NOT NULL` attribute is specified for a `DATETIME` column, a `NULL` value is the default; otherwise, the default is `0`. Unlike for `TIMESTAMP`, `DATETIME` values aren't converted to the UTC time zone for storage.

Other types

Currently, as of MySQL 8.0, the spatial and `JSON` data types fall under this broad category. Using these is a quite advanced topic, and we won't cover them in depth.

Spatial data types are concerned with storing geometrical objects, and MySQL has types corresponding to OpenGIS classes. Working with these types is a topic worth a book on its own.

The `JSON` data type allows native storage of valid JSON documents. Before MySQL 5.7, JSON was usually stored in a `TEXT` or a similar column. However, that has a lot of disadvantages: for example, documents aren't validated, and no storage optimization is performed (all JSON is just stored in

its text form). With the native `JSON` type, it's stored in binary format. If we were to summarize in one sentence: use the `JSON` data type for JSON, dear reader.

Keys and Indexes

You'll find that almost all tables you use will have a `PRIMARY KEY` clause declared in their `CREATE TABLE` statement, and sometimes multiple `KEY` clauses. The reasons why you need a primary key and secondary keys were discussed in Chapter 2. This section discusses how primary keys are declared, what happens behind the scenes when you do so, and why you might want to also create other keys and indexes on your data.

A *primary key* uniquely identifies each row in a table. Even more importantly, for the default InnoDB storage engine, a primary key is also used as a *clustered index*. That means that all of the actual table data is stored in an index structure. That is different from MyISAM, which stores data and indexes separately. When a table is using a clustered index, it's called a clustered table. As we said, in a clustered table each row is stored within an index, compared to being stored in what's usually called a *heap*. Clustering a table results in its rows being sorted according to the clustered index ordering and actually physically stored within the leaf pages of that index. There can't be more than one clustered index per table. For such tables, secondary indexes refer to records in the clustered index instead of the actual table rows. That generally results in improved query performance, though it can be detrimental to writes. InnoDB does not allow you to choose between clustered and nonclustered tables; this is a design decision that you cannot change.

Primary keys are generally a recommended part of any database design, but for InnoDB they are necessary. In fact, if you do not specify a `PRIMARY KEY` clause when creating an InnoDB table, MySQL will use the first `UNIQUE NOT NULL` column as a base for the clustered index. If no such column is available, a hidden clustered index is created, based on ID values assigned by InnoDB to each row.

Given that InnoDB is MySQL's default storage engine and a de facto standard nowadays, we will concentrate on its behavior in this chapter. Alternative storage engines like MyISAM, MEMORY, or MyRocks will be discussed in "Alternative Storage Engines".

As mentioned previously, when a primary key is defined, it becomes a clustered index, and all data in the table is stored in the leaf blocks of that index. InnoDB uses B-tree indexes (more specifically, the B+tree variant), with the exception of indexes on spatial data types, which use the R-tree structure. Other storage engines might implement different index types, but when a table's storage engine is not specified, you can assume that all indexes are B-trees.

Having a clustered index, or in other words having index-organized tables, speeds up queries and sorts involving the primary key columns. However, a downside is that modifying columns in a primary key is expensive. Thus, a good design will require a primary key based on columns that are frequently used for filtering in queries but are rarely modified. Remember that having no primary key at all will result in InnoDB using an implicit cluster index; thus, if you're not sure what columns to pick for a primary key, consider using a synthetic `id` -like column. For example, the `SERIAL` data type might fit well in that case.

Stepping away from InnoDB's internal details, when you declare a primary key for a table in MySQL, it creates a structure that stores information about where the data from each row in the table is stored. This information is called an *index*, and its purpose is to speed up searches that use the primary key. For example, when you declare `PRIMARY KEY (actor_id)` in the `actor` table in the `sakila` database, MySQL creates a structure that allows it to find rows that match a specific `actor_id` (or a range of identifiers) extremely quickly.

This is useful to match actors to films or films to categories, for example. You can display the indexes available on a table using the `SHOW INDEX` (or `SHOW INDEXES`) command:

```
mysql> SHOW INDEX FROM category\G


*************************** 1. row ***************************
        Table: category
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: category_id
    Collation: A
```

```
  Cardinality: 16
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
1 row in set (0.00 sec)
```

The *cardinality* is the number of unique values in the index; for an index on a primary key, this is the same as the number of rows in the table.

Note that all columns that are part of a primary key must be declared as `NOT NULL`, since they must have a value for the row to be valid. Without the index, the only way to find rows in the table is to read each one from disk and check whether it matches the `category_id` you're searching for. For tables with many rows, this exhaustive, sequential searching is extremely slow. However, you can't just index everything; we'll come back to this point at the end of this section.

You can create other indexes on the data in a table. You do this so that other searches (whether on other columns or combinations of columns) are fast, and to avoid sequential scans. For example, take the `actor` table again. Apart from having a primary key on `actor_id`, it also has a secondary key on `last_name` to improve searching by an actor's last name:

```
mysql> SHOW CREATE TABLE actor\G


*************************** 1. row ***************************
       Table: actor
Create Table: CREATE TABLE `actor` (
  `actor_id` smallint unsigned NOT NULL AUTO_INCREMENT,
  ...
  `last_name` varchar(45) NOT NULL,
  ...
  PRIMARY KEY (`actor_id`),
  KEY `idx_actor_last_name` (`last_name`)
) ...
1 row in set (0.00 sec)
```

You can see the keyword `KEY` is used to tell MySQL that an extra index is needed. Alternatively, you can use the word `INDEX` in place of `KEY`. Following that keyword is an index name, and then the column to index is included in parentheses. You can also add indexes after tables are created—in fact, you can pretty much change anything about a table after its creation. This is discussed in "Altering Structures".

You can build an index on more than one column. For example, consider the following table, which is a modified table from `sakila`:

```
mysql> CREATE TABLE customer_mod (
    -> customer_id smallint unsigned NOT NULL AUTO_INCR
    -> first_name varchar(45) NOT NULL,
    -> last_name varchar(45) NOT NULL,
    -> email varchar(50) DEFAULT NULL,
    -> PRIMARY KEY (customer_id),
    -> KEY idx_names_email (first_name, last_name, emai
```

```
Query OK, 0 rows affected (0.06 sec)
```

You can see that we've added a primary key index on the `customer_id` identifier column, and we've also added another index—called `idx_names_email`—that includes the `first_name`, `last_name`, and `email` columns in this order. Let's now consider how you can use that extra index.

You can use the `idx_names_email` index for fast searching by combinations of the three name columns. For example, it's useful in the following query:

```
mysql> SELECT * FROM customer_mod WHERE
    -> first_name = 'Rose' AND
    -> last_name = 'Williams' AND
    -> email = 'rose.w@nonexistent.edu';
```

We know it helps the search, because all the columns listed in the index are used in the query. You can use the `EXPLAIN` statement to check whether what you think should happen is in fact happening:

```
mysql> EXPLAIN SELECT * FROM customer_mod WHERE
    -> first_name = 'Rose' AND
    -> last_name = 'Williams' AND
    -> email = 'rose.w@nonexistent.edu'\G


*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: customer_mod
   partitions: NULL
         type: ref
possible_keys: idx_names_email
          key: idx_names_email
      key_len: 567
          ref: const,const,const
         rows: 1
     filtered: 100.00
        Extra: Using index
1 row in set, 1 warning (0.00 sec)
```

You can see that MySQL reports that the `possible_keys` are `idx_names_email` (meaning that the index could be used for this query) and that the `key` that it's decided to use is `idx_names_email`. So, what you expect and what is happening are the same, and that's good news! You'll find out more about the `EXPLAIN` statement in Chapter 7.

The index we've created is also useful for queries on only the `first_name` column. For example, it can be used by the following query:

```
mysql> SELECT * FROM customer_mod WHERE
    -> first_name = 'Rose';
```

You can use `EXPLAIN` again to check whether the index is being used. The reason it can be used is because the `first_name` column is the first one listed in the index. In practice, this means that the index *clusters*, or stores together, information about rows for all people with the same first name, and so the index can be used to find anyone with a matching first name.

The index can also be used for searches involving combinations of first name and last name, for exactly the same reasons we've just discussed. The index

clusters together people with the same first name, and it clusters people with identical first names by last name. So, it can be used for this query:

```
mysql> SELECT * FROM customer_mod WHERE
    -> first_name = 'Rose' AND
    -> last_name = 'Williams';
```

However, the index can't be used for this query because the leftmost column in the index, `first_name`, does not appear in the query:

```
mysql> SELECT * FROM customer_mod WHERE
    -> last_name = 'Williams' AND
    -> email = 'rose.w@nonexistent.edu';
```

The index should help narrow down the set of rows to a smaller set of possible answers. For MySQL to be able to use an index, the query needs to meet both the following conditions:

1. The leftmost column listed in the `KEY` (or `PRIMARY KEY`) clause must be in the query.
2. The query must contain no `OR` clauses for columns that aren't indexed.

Again, you can always use the `EXPLAIN` statement to check whether an index can be used for a particular query.

Before we finish this section, here are a few ideas on how to choose and design indexes. When you're considering adding an index, think about the following:

- Indexes cost space on disk, and they need to be updated whenever data changes. If your data changes frequently, or lots of data changes when you do make a change, indexes will slow the process down. However, in practice, since `SELECT` statements (data reads) are usually much more common than other statements (data modifications), indexes are usually beneficial.
- Only add an index that'll be used frequently. Don't bother indexing columns before you see what queries your users and your applications need. You can always add indexes afterward.
- If all columns in an index are used in all queries, list the column with the highest number of duplicates at the left of the `KEY` clause. This minimizes

index size.

- The smaller the index, the faster it'll be. If you index large columns, you'll get a larger index. This is a good reason to ensure your columns are as small as possible when you design your tables.
- For long columns, you can use only a prefix of the values from a column to create the index. You can do this by adding a value in parentheses after the column definition, such as `KEY idx_names_email (first_name(3), last_name(2), email(10))`. This means that only the first 3 characters of `first_name` are indexed, then the first 2 characters of `last_name`, and then 10 characters from `email`. This is a significant savings over indexing 140 characters from the three columns! When you do this, your index will be less able to uniquely identify rows, but it'll be much smaller and still reasonably good at finding matching rows. Using a prefix is mandatory for long types like `TEXT`.

To wrap up this section, we need to discuss some peculiarities regarding secondary keys in InnoDB. Remember that all the table data is stored in the leaves of the clustered index. That means, using the `actor` example, that if we need to get the `first_name` data when filtering by `last_name`, even though we can use `idx_actor_last_name` for quick filtering, we will need to access the data by the primary key. As a consequence, each secondary key in InnoDB has all of the primary key columns appended to its definition implicitly. Having unnecessarily long primary keys in InnoDB thus results in significantly bloated secondary keys.

This can also be seen in the `EXPLAIN` output (note the `Extra: Using index` in the first output of the first command):

```
mysql> EXPLAIN SELECT actor_id, last_name FROM actor WH
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
   partitions: NULL
         type: ref
possible_keys: idx_actor_last_name
          key: idx_actor_last_name
      key_len: 182
          ref: const
```

```
          rows: 1
      filtered: 100.00
         Extra: Using index
1 row in set, 1 warning (0.00 sec)
```

```
mysql> EXPLAIN SELECT first_name FROM actor WHERE last_
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
   partitions: NULL
         type: ref
possible_keys: idx_actor_last_name
          key: idx_actor_last_name
      key_len: 182
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

Effectively, `idx_actor_last_name` is a *covering index* for the first query, meaning that InnoDB can extract all the required data from that index alone. However, for the second query, InnoDB will have to do an additional lookup of a clustered index to get the value for the `first_name` column.

The AUTO_INCREMENT Feature

MySQL's proprietary `AUTO_INCREMENT` feature allows you to create a unique identifier for a row without running a `SELECT` query. Here's how it works. Let's take the simplified `actor` table again:

```
mysql> CREATE TABLE actor (
    -> actor_id smallint unsigned NOT NULL AUTO_INCREME
    -> first_name varchar(45) NOT NULL,
    -> last_name varchar(45) NOT NULL,
```

```
    -> PRIMARY KEY (actor_id)
    -> );
```

```
Query OK, 0 rows affected (0.03 sec)
```

It's possible to insert rows into that table without specifying the `actor_id` :

```
mysql> INSERT INTO actor VALUES (NULL, 'Alexander', 'Ka
```

```
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO actor VALUES (NULL, 'Anatoly', 'Solc
```

```
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO actor VALUES (NULL, 'Nikolai', 'Grir
```

```
Query OK, 1 row affected (0.00 sec)
```

When you view the data in this table, you can see that each row has a value assigned for the `actor_id` column:

```
mysql> SELECT * FROM actor;
```

```
+----------+------------+-------------+
| actor_id | first_name | last_name   |
+----------+------------+-------------+
|        1 | Alexander  | Kaidanovsky |
|        2 | Anatoly    | Solonitsyn  |
|        3 | Nikolai    | Grinko      |
+----------+------------+-------------+
3 rows in set (0.00 sec)
```

Each time a new row is inserted, a unique value for the `actor_id` column is created for that new row.

Consider how this feature works. You can see that the `actor_id` column is declared as an integer with the clauses `NOT NULL AUTO_INCREMENT`. `AUTO_INCREMENT` tells MySQL that when a value isn't provided for this column, the value allocated should be one more than the maximum currently stored in the table. The `AUTO_INCREMENT` sequence begins at 1 for an empty table.

The `NOT NULL` clause is required for `AUTO_INCREMENT` columns; when you insert `NULL` (or 0, though this isn't recommended), the MySQL server automatically finds the next available identifier and assigns it to the new row. You can manually insert negative values if the column was not defined as `UNSIGNED`; however, for the next automatic increment, MySQL will simply use the largest (positive) value in the column, or start from 1 if there are no positive values.

The `AUTO_INCREMENT` feature has the following requirements:

- The column it is used on must be indexed.
- The column it is used on cannot have a `DEFAULT` value.
- There can be only one `AUTO_INCREMENT` column per table.

MySQL supports different storage engines; we'll talk more about these in "Alternative Storage Engines". When using the nondefault MyISAM table type, you can use the `AUTO_INCREMENT` feature on keys that comprise multiple columns. In effect, you can have multiple independent counters within a single `AUTO_INCREMENT` column. However, this is not possible with InnoDB.

While the `AUTO_INCREMENT` feature is useful, it isn't portable to other database environments, and it hides the logical steps for creating new identifiers. It can also lead to ambiguity; for example, dropping or truncating a table will reset the counter, but deleting selected rows (with a `WHERE` clause) doesn't reset the counter. Moreover, if a row is inserted inside a transaction but then that transaction is rolled back, an identifier will be used up anyway. As an example, let's create the table `count` that contains an auto-incrementing field `counter`:

```
mysql> CREATE TABLE count (counter INT AUTO_INCREMENT K
```

```
Query OK, 0 rows affected (0.13 sec)
```

```
mysql> INSERT INTO count VALUES (),(),(),(),(),();
```

```
Query OK, 6 rows affected (0.01 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM count;
```

```
+---------+
| counter |
+---------+
| 1       |
| 2       |
| 3       |
| 4       |
| 5       |
| 6       |
+---------+
6 rows in set (0.00 sec)
```

Inserting several values works as expected. Now, let's delete a few rows and then add six new rows:

```
mysql> DELETE FROM count WHERE counter > 4;
```

```
Query OK, 2 rows affected (0.00 sec)
```

```
mysql> INSERT INTO count VALUES (),(),(),(),(),();
```

```
Query OK, 6 rows affected (0.00 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM count;


+---------+
| counter |
+---------+
| 1       |
| 2       |
| 3       |
| 4       |
| 7       |
| 8       |
| 9       |
| 10      |
| 11      |
| 12      |
+---------+
10 rows in set (0.00 sec)
```

Here, we see that the counter is not reset and continues from 7. If, however, we truncate the table, thus removing all of the data, the counter is reset to 1:

```
mysql> TRUNCATE TABLE count;


Query OK, 0 rows affected (0.00 sec)


mysql> INSERT INTO count VALUES (),(),(),(),(),();


Query OK, 6 rows affected (0.01 sec)
Records: 6  Duplicates: 0  Warnings: 0


mysql> SELECT * FROM count;


+---------+
| counter |
+---------+
| 1       |
| 2       |
| 3       |
```

```
| 4       |
| 5       |
| 6       |
+---------+
6 rows in set (0.00 sec)
```

To summarize: `AUTO_INCREMENT` guarantees a sequence of transactional and monotonically increasing values. However, it does not in any way guarantee that each individual identifier provided will exactly follow the previous one. Usually, this behavior of `AUTO_INCREMENT` is clear enough and should not be a problem. However, if your particular use case requires a counter that guarantees no gaps, you should consider using some kind of workaround. Unfortunately, it'll likely be implemented on the application side.

Altering Structures

We've shown you all the basics you need for creating databases, tables, indexes, and columns. In this section, you'll learn how to add, remove, and change columns, databases, tables, and indexes in structures that already exist.

Adding, Removing, and Changing Columns

You can use the `ALTER TABLE` statement to add new columns to a table, remove existing columns, and change column names, types, and lengths.

Let's begin by considering how you modify existing columns. Consider an example in which we rename a table column. The `language` table has a column called `last_update` that contains the time the record was modified. To change the name of this column to `last_updated_time` , you would write:

```
mysql> ALTER TABLE language RENAME COLUMN last_update T
```

```
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

This particular example utilizes the *online DDL* feature of MySQL. What actually happens behind the scenes is that MySQL only modifies metadata and

doesn't need to actually rewrite the table in any way. You can see that by the lack of affected rows. Not all DDL statements can be performed online, so this won't be the case with many of the changes you make.

NOTE

DDL stands for data definition language, and in the context of SQL it's a subset of syntax and statements used to create, modify, and delete schema objects such as databases, tables, indexes, and columns. `CREATE TABLE` and `ALTER TABLE` are both DDL operations, for example.

Executing DDL statements requires special internal mechanisms, including special locking—this is a good thing, as you probably wouldn't like tables changing while your queries are running! These special locks are called metadata locks in MySQL, and we give a detailed overview of how they work in "Metadata Locks".

Note that all DDL statements, including those that execute through online DDL, require metadata locks to be obtained. In that sense, online DDL statements are not so "online," but they won't lock the target table entirely while they are running. Executing DDL statements on a running system under load is a risky venture: even a statement that should execute almost instantaneously may wreak havoc. We recommend that you read carefully about metadata locking in Chapter 6 and in the link to MySQL documentation, and experiment with running different DDL statements with and without concurrent load. That may not be too important while you're learning MySQL, but we think it's worth cautioning you up front. With that covered, let's get back to our `ALTER` of the `language` table.

You can check the result with the `SHOW COLUMNS` statement:

```
mysql> SHOW COLUMNS FROM language;

+------------------+------------------+------+-----+--
| Field            | Type             | Null | Key | D
+------------------+------------------+------+-----+--
| language_id      | tinyint unsigned | NO   | PRI | N
| name             | char(20)         | NO   |     | N
| last_updated_time | timestamp        | NO   |     | C
```

```
+------------------+-----------------+------+-----+--
3 rows in set (0.01 sec)
```

In the previous example we used the `ALTER TABLE` statement with the `RENAME COLUMN` keyword. That is a MySQL 8.0 feature. We could alternatively use `ALTER TABLE` with the `CHANGE` keyword for compatibility:

```
mysql> ALTER TABLE language CHANGE last_update last_upc
    -> NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CUF
```

```
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

In this example, you can see that we provided four parameters to the `ALTER TABLE` statement with the `CHANGE` keyword:

1. The table name, `language`
2. The original column name, `last_update`
3. The new column name, `last_updated_time`
4. The column type, `TIMESTAMP` , with a lot of extra attributes, which are necessary to avoid changing the original definition

You must provide all four; that means you need to respecify the type and any clauses that go with it. In this example, as we're using MySQL 8.0 with the default settings, `TIMESTAMP` no longer has explicit defaults. As you can see, using `RENAME COLUMN` is much easier than `CHANGE` .

If you want to modify the type and clauses of a column, but not its name, you can use the `MODIFY` keyword:

```
mysql> ALTER TABLE language MODIFY name CHAR(20) DEFAUL
```

```
Query OK, 0 rows affected (0.14 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

You can also do this with the `CHANGE` keyword, but by specifying the same column name twice:

```
mysql> ALTER TABLE language CHANGE name name CHAR(20) D
```

```
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Be careful when you're modifying types:

- Don't use incompatible types, since you're relying on MySQL to successfully convert data from one format to another (for example, converting an `INT` column to a `DATETIME` column isn't likely to do what you hoped).
- Don't truncate the data unless that's what you want. If you reduce the size of a type, the values will be edited to match the new width, and you can lose data.

Suppose you want to add an extra column to an existing table. Here's how to do it with the `ALTER TABLE` statement:

```
mysql> ALTER TABLE language ADD native_name CHAR(20);
```

```
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

You must supply the `ADD` keyword, the new column name, and the column type and clauses. This example adds the new column, `native_name` , as the last column in the table, as shown with the `SHOW COLUMNS` statement:

```
mysql> SHOW COLUMNS FROM artist;
```

```
+------------------+------------------+------+-----+--
| Field            | Type             | Null | Key | D
+------------------+------------------+------+-----+--
| language_id      | tinyint unsigned | NO   | PRI | N
| name             | char(20)         | YES  |     | r
```

```
| last_updated_time | timestamp        | NO   |     | C
| native_name       | char(20)         | YES  |     | N
+-------------------+------------------+------+-----+--
4 rows in set (0.00 sec)
```

If you want it to instead be the first column, use the `FIRST` keyword as follows:

```
mysql> ALTER TABLE language ADD native_name CHAR(20) FI
```

```
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> SHOW COLUMNS FROM language;
```

```
+-------------------+------------------+------+-----+--
| Field             | Type             | Null | Key | D
+-------------------+------------------+------+-----+--
| native_name       | char(20)         | YES  |     | N
| language_id       | tinyint unsigned | NO   | PRI | N
| name              | char(20)         | YES  |     | n
| last_updated_time | timestamp        | NO   |     | C
+-------------------+------------------+------+-----+--
4 rows in set (0.01 sec)
```

If you want it added in a specific position, use the `AFTER` keyword:

```
mysql> ALTER TABLE language ADD native_name CHAR(20) AF
```

```
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> SHOW COLUMNS FROM language;
```

```
+-------------------+------------------+------+-----+--
| Field             | Type             | Null | Key | D
```

```
+------------------+-----------------+------+-----+--
| language_id      | tinyint unsigned | NO   | PRI | N
| name             | char(20)         | YES  |     | r
| native_name      | char(20)         | YES  |     | N
| last_updated_time | timestamp       | NO   |     | C
+------------------+-----------------+------+-----+--
4 rows in set (0.00 sec)
```

To remove a column, use the `DROP` keyword followed by the column name. Here's how to get rid of the newly added `native_name` column:

```
mysql> ALTER TABLE language DROP native_name;


Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

This removes both the column structure and any data contained in that column. It also removes the column from any indexes it was in; if it's the only column in the index, the index is dropped, too. You can't remove a column if it's the only one in a table; to do this, you drop the table instead, as explained in "Deleting Structures". Be careful when dropping columns, because when the structure of a table changes, you will generally have to modify any `INSERT` statements that you use to insert values in a particular order. For more on this, see "The INSERT Statement".

MySQL allows you to specify multiple alterations in a single `ALTER TABLE` statement by separating them with commas. Here's an example that adds a new column and adjusts another:

```
mysql> ALTER TABLE language ADD native_name CHAR(255),


Query OK, 6 rows affected (0.06 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

Note that this time, you can see that six records were changed. In the previous `ALTER TABLE` commands, MySQL reported that no rows were affected. The difference is that this time, we're not performing an online DDL operation, because changing any column's type will always result in a table being rebuilt.

We recommend reading about online DDL operations in the Reference Manual when planning your changes. Combining online and offline operations will result in an offline operation.

When not using online DDL or when any of the modifications is "offline," it's very efficient to join multiple modifications in a single operation. That potentially saves the cost of creating a new table, copying data from the old table to the new table, dropping the old table, and renaming the new table with the name of the old table for each modification individually.

Adding, Removing, and Changing Indexes

As we discussed previously, it's often hard to know what indexes are useful before the application you're building is used. You might find that a particular feature of the application is much more popular than you expected, causing you to evaluate how to improve performance for the associated queries. You'll therefore find it useful to be able to add, alter, and remove indexes on the fly after your application is deployed. This section shows you how. Note that modifying indexes does not affect the data stored in a table.

We'll start with adding a new index. Imagine that the `language` table is frequently queried using a `WHERE` clause that specifies the `name`. To speed up these queries, you've decided to add a new index, which you've named `idx_name`. Here's how you add it after the table is created:

```
mysql> ALTER TABLE language ADD INDEX idx_name (name);
```

```
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Again, you can use the terms `KEY` and `INDEX` interchangeably. You can check the results with the `SHOW CREATE TABLE` statement:

```
mysql> SHOW CREATE TABLE language\G
```

```
*************************** 1. row ***************************
       Table: language
Create Table: CREATE TABLE `language` (
```

```
  `language_id` tinyint unsigned NOT NULL AUTO_INCREMEN
  `name` char(255) DEFAULT NULL,
  `last_updated_time` timestamp NOT NULL
    DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTA
  PRIMARY KEY (`language_id`),
  KEY `idx_name` (`name`)
) ENGINE=InnoDB AUTO_INCREMENT=8
    DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

As expected, the new index forms part of the table structure. You can also specify a primary key for a table after it's created:

```
mysql> CREATE TABLE no_pk (id INT);

Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO no_pk VALUES (1),(2),(3);

Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE no_pk ADD PRIMARY KEY (id);

Query OK, 0 rows affected (0.13 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Now let's consider how to remove an index. To remove a non-primary key index, you do the following:

```
mysql> ALTER TABLE language DROP INDEX idx_name;

Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

You can drop a primary key index as follows:

```
mysql> ALTER TABLE no_pk DROP PRIMARY KEY;

Query OK, 3 rows affected (0.07 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

MySQL won't allow you to have multiple primary keys in a table. If you want to change the primary key, you'll have to remove the existing index before adding the new one. However, we know that it's possible to group DDL operations. Consider this example:

```
mysql> ALTER TABLE language DROP PRIMARY KEY,
    -> ADD PRIMARY KEY (language_id, name);

Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

You can't modify an index once it's been created. However, sometimes you'll want to; for example, you might want to reduce the number of characters indexed from a column or add another column to the index. The best method to do this is to drop the index and then create it again with the new specification. For example, suppose you decide that you want the `idx_name` index to include only the first 10 characters of the `artist_name`. Simply do the following:

```
mysql> ALTER TABLE language DROP INDEX idx_name,
    -> ADD INDEX idx_name (name(10));

Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Renaming Tables and Altering Other Structures

We've seen how to modify columns and indexes in a table; now let's see how to modify tables themselves. It's easy to rename a table. Suppose that you want to rename `language` to `languages`. Use the following command:

```
mysql> ALTER TABLE language RENAME TO languages;

Query OK, 0 rows affected (0.04 sec)
```

The `TO` keyword is optional.

There are several other things you can do with `ALTER` statements, including:

- Change the default character set and collation order for a database, a table, or a column.
- Manage and change constraints. For example, you can add and remove foreign keys.
- Add partitioning to a table, or alter the current partitioning definition.
- Change the storage engine of a table.

You can find more about these operations in the MySQL Reference Manual, in the sections on the `ALTER DATABASE` and `ALTER TABLE` statements. An alternative shorter notation for the same statement is `RENAME TABLE`:

```
mysql> RENAME TABLE languages TO language;

Query OK, 0 rows affected (0.04 sec)
```

One thing that it's not possible to alter is a name of a particular database. However, if you're using the InnoDB engine, you can use `RENAME` to move tables between databases:

```
mysql> CREATE DATABASE sakila_new;

Query OK, 1 row affected (0.05 sec)

mysql> RENAME TABLE sakila.language TO sakila_new.langu

Query OK, 0 rows affected (0.05 sec)
```

```
mysql> USE sakila;

Database changed

mysql> SHOW TABLES LIKE 'lang%';

Empty set (0.00 sec)

mysql> USE sakila_new;

Database changed

mysql> SHOW TABLES LIKE 'lang%';

+------------------------------+
| Tables_in_sakila_new (lang%) |
+------------------------------+
| language                     |
+------------------------------+
1 row in set (0.00 sec)
```

Deleting Structures

In the previous section, we showed how you can delete columns and rows from a database; now we'll describe how to remove databases and tables.

Dropping Databases

Removing, or *dropping*, a database is straightforward. Here's how you drop the `sakila` database:

```
mysql> DROP DATABASE sakila;
```

```
Query OK, 25 rows affected (0.16 sec)
```

The number of rows returned in the response is the number of tables removed. You should take care when dropping a database, since all its tables, indexes, and columns are deleted, as are all the associated disk-based files and directories that MySQL uses to maintain them.

If a database doesn't exist, trying to drop it causes MySQL to report an error. Let's try dropping the `sakila` database again:

```
mysql> DROP DATABASE sakila;
```

```
ERROR 1008 (HY000): Can't drop database 'sakila'; datab
```

You can avoid the error, which is useful when including the statement in a script, by using the `IF EXISTS` phrase:

```
mysql> DROP DATABASE IF EXISTS sakila;
```

```
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

You can see that a warning is reported since the `sakila` database has already been dropped.

Removing Tables

Removing tables is as easy as removing a database. Let's create and remove a table from the `sakila` database:

```
mysql> CREATE TABLE temp (id SERIAL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> DROP TABLE temp;
```

```
Query OK, 0 rows affected (0.03 sec)
```

Don't worry: the `0 rows affected` message is misleading. You'll find the table is definitely gone.

You can use the `IF EXISTS` phrase to prevent errors. Let's try dropping the `temp` table again:

```
mysql> DROP TABLE IF EXISTS temp;

Query OK, 0 rows affected, 1 warning (0.01 sec)
```

As usual, you can investigate the warning with the `SHOW WARNINGS` statement:

```
mysql> SHOW WARNINGS;

+-------+------+-----------------------------+
| Level | Code | Message                     |
+-------+------+-----------------------------+
| Note  | 1051 | Unknown table 'sakila.temp' |
+-------+------+-----------------------------+
1 row in set (0.00 sec)
```

You can drop more than one table in a single statement by the separating table names with commas:

```
mysql> DROP TABLE IF EXISTS temp, temp1, temp2;

Query OK, 0 rows affected, 3 warnings (0.00 sec)
```

In this case there are three warnings because none of these tables existed.