O'REILLY®

Second Edition

Learning MySQL

Get a Handle on Your Data

Early Release
RAW & UNEDITED

Vinicius M. Grippa
& Sergey Kuzmichev

Learning MySQL

SECOND EDITION

Get a Handle on Your Data

Vinicius M. Grippa and Sergey Kuzmichev

Learning MySQL

by Vinicius M. Grippa and Sergey Kuzmichev

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Andy Kwan

Development Editor: Corbin Collins

Production Editor: Beth Kelly

Copyeditor: Rachel Head

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

November 2006: First Edition

September 2021: Second Edition

Revision History for the Second Edition

- 2021-08-25: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492085928 for release details.

978-1-492-08592-8

[LSI]

Preface

Database management systems are part of the core of many companies nowadays. Even if a business is not technology-focused, it needs to store, access, and manipulate data in a fast, secure, and reliable way. With the Covid-19 pandemic, areas that traditionally resisted digital transformation, like the judiciary systems in many countries are now being integrated through technology due to travel and meeting restrictions. And online shopping and working from home are more popular than ever before.

But it’s not just disasters that have propelled such far-reaching changes. With the advent of 5G, we will soon have many more machines connected to the internet than humans. Vast amounts of data are already being harvested, stored, and used to train machine learning models, artificial intelligence, and much more. We are living at the beginning of the next revolution.

And with the need to store more data, several database types have emerged to help with this mission, especially from the unstructured data world, including NOSQL databases like MongoDB, Cassandra, and Redis. However, we can see that traditional SQL databases remain strong. There is no sign that they will vanish in the near future. And in the SQL world, the one that is undoubtedly the most popular open source solution is MySQL.

Both authors of this book worked with many customers from all parts of the world. Along the way, we have learned lots of lessons and experienced a vast number of cases going from mission-critical monolith applications to simpler microservices applications. This book if full of the tips and advice we think most readers will find helpful for their daily activities.

Who This Book Is for

This book is primarily for people using MySQL for the first time or learning it as a second database. If you are entering the database area for the first time, the first chapters will introduce you to the database design, concepts, and how to deploy MySQL into different operating systems and in the cloud.

For those coming from another ecosystem like Postgres, Oracle, or SQL Server, the book covers backup, high-availability, and disaster and recovery strategies.

We hope all readers will also find this book to be a good companion for learning or reviewing fundamentals, from the architecture to “production environment” bits of advice.

How This Book Is Organized

We introduce many topics, from the basic installation process, database design, backups, and recovery to CPU performance analysis and bug investigation. We divide the book into four main parts:

1. Starting with MySQL
2. Using MySQL
3. MySQL in Production
4. Miscellaneous Topics

Let's look at how we've organized the chapters.

Starting with MySQL

Chapter 1 explains how to install and configure the MySQL software on different operating systems. This chapter provides far more detail than most books do. We know that those initiating their career with MySQL are often unfamiliar with various Linux distributions and installation options. Running the "MySQL hello world" requires far more steps than compiling a hello world in any programming language does. You will see how to deploy MySQL in Linux, Windows, macOS, and Docker, and how to deploy instances quickly for testing.

Using MySQL

Before we dive into creating and using databases, we look at proper database design in Chapter 2. You will learn how to access your database's features and see how the information items in your database relate to each other. You will see that lousy database designs are challenging to change and can lead to performance problems. We will introduce the concept of strong and weak entities and their relationships (*foreign keys*). This chapter also shows how to download and configure database examples such as Sakila, World, and Employee.

In Chapter 3, we explore the famous SQL commands that are part of the CRUD (create, read, update, and delete). We will see how to read data from an existing MySQL database, store data in it, and manipulate existing data.

In Chapter 4, we explain how to create a new MySQL database and create and modify tables, indexes, and other database structures.

Chapter 5 covers more advanced operations such as using nested queries and using different MySQL database engines. This chapter will give you the capability of performing more complex queries.

MySQL in production

With the concepts for installing and manipulating data in hand, the next step is to understand how MySQL handles simultaneous access to the same data. The ideas of isolation, transaction, and deadlocks are explored in the Chapter 6.

In Chapter 7, you will see more complex queries that you can perform in MySQL. And you will see how to observe the query plan to check whether the query is efficient or not. Finally, we explain the different engines available in MySQL (InnoDB and MyISAM are the most famous ones).

With Chapter 8 you will see how to create and delete users in the database. This step is one of the most important in terms of security since users with more privileges than needed can cause considerable damage to the database and the company's reputation. You will see how to establish security policies, give and remove privileges, and restrict access to specific network IPs.

The Chapter 9 covers the MySQL configuration file and its options. This file contains the MySQL configuration like its buffer pool, the size of the redo log files, and all necessary settings to customize MySQL. Those familiar with MySQL will recognize the */etc/my.cnf* configuration file. You will also see that it is possible to configure user access using special option files.

Databases without backup policies are headed for disaster sooner or later. In Chapter 10 we discuss the different types of backups (*logical* vs. *physical*), the options available to execute this task, and the ones that are more appropriate for large production databases.

At the end of the production part, Chapter 11 discusses the essential parameters you need to pay attention to when setting a new server. We provide formulas for that and identify whether the parameter value is the correct one for the database workload.

Miscellaneous topics

With the essential established, it is time to go beyond. Chapter 12 teaches how to monitor your database and collect data from it. Since database workload behavior can change according to the number of users, transactions, and data manipulated, identifying which resource is saturated and what is causing the problem is crucial.

Chapter 13 explains how to create a replication between two servers to provide high availability. We also introduce the cluster concept, highlighting two: InnoDB Cluster and Galera/PXC cluster.

Chapter 14 expands the MySQL universe to the cloud. You will see how to deploy DBaaS (also known as managed database service) in the most prominent three cloud providers: AWS, Azure, and Google Cloud.

In Chapter 15, we show the most commonly used tools to distribute the queries among different MySQL servers to extract even more performance from MySQL.

The last chapter, Chapter 16, introduces more advanced analysis methods, tools, and a bit of programming. In this chapter, we talk about MySQL shell, flame graphs, and how to analyze bugs.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, and more) is available for download at the Learning MySQL 2nd Edition GitHub repository.

Chapter 2. Modeling and Designing Databases

When implementing a new database, it's easy to fall into the trap of quickly getting something up and running without dedicating adequate time and effort to the design. This carelessness frequently leads to costly redesigns and reimplementations down the road. Designing a database is like drafting the blueprints for a house; it's silly to start building without detailed plans. Notably, good design allows you to extend the original building without pulling everything down and starting from scratch. And as you will see, bad designs are directly related to poor database performance.

How Not to Develop a Database

Database design is probably not the most exciting task in the world, but indeed it is becoming one of the most important ones. Before we describe how to go about the design process, let's look at an example of database design on the run.

Imagine we want to create a database to store student grades for a university computer science department. We could create a `Student_Grades` table to store grades for each student and each course. The table would have columns for the given names and the surname of each student and each course they have taken, the course name, and the percentage result (shown as `Pctg`). We'd have a different row for each student for each of their courses:

```
+------------+---------+---------------------------+------+
| GivenNames | Surname | CourseName                | Pctg |
+------------+---------+---------------------------+------+
| John Paul  | Bloggs  | Data Science              |   72 |
| Sarah      | Doe     | Programming 1             |   87 |
| John Paul  | Bloggs  | Computing Mathematics     |   43 |
| John Paul  | Bloggs  | Computing Mathematics     |   65 |
| Sarah      | Doe     | Data Science              |   65 |
| Susan      | Smith   | Computing Mathematics     |   75 |
| Susan      | Smith   | Programming 1             |   55 |
| Susan      | Smith   | Computing Mathematics     |   80 |
+------------+---------+---------------------------+------+
```

The list is nice and compact, we can easily access grades for any student or any course, and it looks similar to a spreadsheet. However, we could have more than one student called Susan Smith; there are two entries for Susan Smith and the Computing Mathematics course in the sample data. Which Susan Smith got an 80? A common way to differentiate duplicate data entries is to assign a unique number to each entry. Here, we can assign a unique `Student ID` number to each student:

```
+------------+------------+---------+---------------------------+------+
| StudentID  | GivenNames | Surname | CourseName                | Pctg |
+------------+------------+---------+---------------------------+------+
| 12345678   | John Paul  | Bloggs  | Data Science              |   72 |
| 12345121   | Sarah      | Doe     | Programming 1             |   87 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics     |   43 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics     |   65 |
| 12345121   | Sarah      | Doe     | Data Science              |   65 |
| 12345876   | Susan      | Smith   | Computing Mathematics     |   75 |
| 12345876   | Susan      | Smith   | Programming 1             |   55 |
| 12345303   | Susan      | Smith   | Computing Mathematics     |   80 |
+------------+------------+---------+---------------------------+------+
```

Now we know which Susan Smith got 80; it is the one with the Student ID number 12345303.

There's another problem. In our table, John Paul Bloggs has failed the Computing Mathematics course once with 43 percent and passed it with 65 percent in his second attempt. In a relational database, the rows form a set, and there is no implicit ordering between them; we might guess that the pass happened after the failure, but we can't be sure. There's no guarantee that the newer grade will appear after the older one, so we need to add information about when each grade was awarded, say by adding a year and semester (`Sem`):

```
+------------+------------+---------+--------------------------+------+-----+------+
| StudentID  | GivenNames | Surname | CourseName               | Year | Sem | Pctg |
+------------+------------+---------+--------------------------+------+-----+------+
| 12345678   | John Paul  | Bloggs  | Data Science             | 2019 |   2 |   72 |
| 12345121   | Sarah      | Doe     | Programming 1            | 2020 |   1 |   87 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics    | 2019 |   2 |   43 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics    | 2020 |   1 |   65 |
| 12345121   | Sarah      | Doe     | Data Science             | 2020 |   1 |   65 |
| 12345876   | Susan      | Smith   | Computing Mathematics    | 2019 |   1 |   75 |
| 12345876   | Susan      | Smith   | Programming 1            | 2019 |   2 |   55 |
| 12345303   | Susan      | Smith   | Computing Mathematics    | 2020 |   1 |   80 |
+------------+------------+---------+--------------------------+------+-----+------+
```

Notice that the `Student_Grades` table has become a bit bloated: We repeated the student ID, given names, and surname for every grade. We could split up the information and create a `Student_Details` table:

```
+------------+------------+---------+
| StudentID  | GivenNames | Surname |
+------------+------------+---------+
| 12345121   | Sarah      | Doe     |
| 12345303   | Susan      | Smith   |
| 12345678   | John Paul  | Bloggs  |
| 12345876   | Susan      | Smith   |
+------------+------------+---------+
```

And we could keep less information in the `Student_Grades` table:

```
+------------+--------------------------+------+-----+------+
| StudentID  | CourseName               | Year | Sem | Pctg |
+------------+--------------------------+------+-----+------+
| 12345678   | Data Science             | 2019 |   2 |   72 |
| 12345121   | Programming 1            | 2020 |   1 |   87 |
| 12345678   | Computing Mathematics    | 2019 |   2 |   43 |
| 12345678   | Computing Mathematics    | 2020 |   1 |   65 |
| 12345121   | Data Science             | 2020 |   1 |   65 |
| 12345876   | Computing Mathematics    | 2019 |   1 |   75 |
| 12345876   | Programming 1            | 2019 |   2 |   55 |
| 12345303   | Computing Mathematics    | 2020 |   1 |   80 |
+------------+--------------------------+------+-----+------+
```

To look up a student's grades, we'd need to first look up her Student ID from the `Student_Details` table and then read the grades for that Student ID from the `Student_Grades` table.

There are still issues we haven't considered. For example, should we keep information on a student's enrollment date, postal and email addresses, fees, or attendance? Should we store different types of postal addresses? How should we store addresses so that things don't break when students change their addresses?

Implementing a database in this way is problematic; we keep running into things we hadn't thought about and have to keep changing our database structure. We can save a lot of reworking by carefully documenting the requirements and then working through them to develop a coherent design.

The Database Design Process

There are three major stages in the database design, each producing a progressively lower-level description:

1. Requirements analysis:: First, we determine and write down what we need from the database, which data we will store, and how the data items relate to each other. In practice, this might involve a detailed study

of the application requirements and talk to people in various roles that will interact with the database and application.

2. Conceptual design:: Once we know the database requirements, we distill them into a formal description of the database design. Later in this chapter we'll see how to use modeling to produce the conceptual design.
3. Logical design:: Finally, we map the database design onto an existing database management system and database tables.

At the end of the chapter, we'll look at how we can use the open-source MySQL Workbench tool to convert the conceptual design to a MySQL database schema.

The Entity Relationship Model

At a basic level, databases store information about distinct objects, or *entities*, and the associations, or *relationships*, between these entities. For example, a university database might store information about students, courses, and enrollment. A student and a course are entities, whereas enrollment is a relationship between a student and a course. Similarly, an inventory and sales database might store information about products, customers, and sales. A product and a customer are entities, and a sale is a relationship between a customer and a product. It is common to get confused between entity and relationships when starting, and we may end up designing relationships as entities and vice-versa. The best way to improve database design is by practicing a lot.

A popular approach to conceptual design uses the *Entity Relationship* (ER) model, which helps transform the requirements into a formal description of the entities and relationships in the database. We'll start by looking at how the ER modeling process works and then observe it in "Entity Relationship Modeling Examples" for three sample databases.

Representing Entities

To help visualize the design, the Entity-Relationship Modeling approach involves drawing an ER diagram. In the ER diagram, we represent an entity set by a rectangle containing the entity name. For our sales database example, our ER would show the product and customer entity sets, as shown in Figure 2-1.

Customer
Product

Figure 2-1. An entity set is represented by a named rectangle

We typically use the database to store specific characteristics, or *attributes*, of the entities. We could record the name, email address, postal address, and telephone number of each customer in a sales database. In a more elaborate customer relationship management (CRM) application, we could also store the names of the customer's spouse and children, the languages the customer speaks, the customer's history of interaction with our company, and so on. Attributes describe the entity they belong to.

We may form an attribute from smaller parts; for example, we compose a postal address from a street number, city, ZIP code, and country. We classify attributes as *composite* if they're composed of smaller parts in this way, and as *simple* otherwise.

Some attributes can have multiple values for a given entity. For example, a customer could provide several telephone numbers, so the telephone number attribute is multivalued.

Attributes help distinguish one entity from other entities of the same type. We could use the name attribute to differentiate between customers, but this could be an inadequate solution because several customers could have identical names. To tell them apart, we need an attribute (or a minimal combination of attributes) guaranteed to be unique to each customer. The identifying attribute or attributes form a unique key, and in this particular case, we call it a *Primary Key*.

In our example, we can assume that no two customers have the same email address, so that the email address can be the primary key. However, we need to think carefully about the implications of our choices. For example, if we decide to identify customers by their email addresses, it would be hard to allow a customer to have multiple email addresses. Any applications we build to use this database might treat each email address as a separate person. It might be hard to adapt everything to allow people to have multiple email addresses. Using the email address as the key also means that every customer must have an email address; otherwise, we couldn't distinguish between customers who don't have one.

Looking at the other attributes for one that can serve as an alternative key, we see that while it's possible that two customers would have the same telephone number (and so we cannot use the telephone number as a key), it's likely that people who have the same telephone number never have the same name so that we can use the combination of the telephone number and the name as a composite key.

Clearly, there may be several possible keys that could be used to identify an entity; we choose one of the alternatives, or *candidate*, keys to be our main, or *primary*, key. We usually choose based on how confident we are that the attribute will be non-empty and unique for each entity and how small the key is (shorter keys are faster to maintain and to perform lookup operations).

In the ER diagram, attributes are represented as labeled ovals and connected to their entity, as shown in Figure 2-2. Attributes comprising the primary key are shown underlined. The parts of any composite attributes are drawn connected to the composite attribute's oval, and multivalued attributes are shown as double-lined ovals.

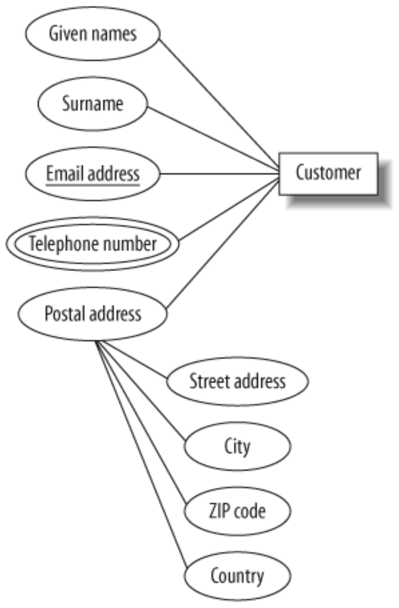


Figure 2-2. The ER diagram representation of the customer entity

Attribute values are chosen from a domain of legal values; for example, we could specify that a customer's given names and surname attributes can each be a string of up to 100 characters, while a telephone number can be a string of up to 40 characters. Similarly, a product price could be a positive rational number.

Attributes can be empty; for example, some customers may not provide their telephone numbers. The primary key of an entity (including the components of a multiattribute primary key) must never be unknown (technically, it must be `NOT NULL`); for example, if it's possible for a customer to not provide an email address, we cannot use the email address as the key.

You should think carefully when classifying an attribute as multivalued: are all the values equivalent, or do they in fact represent different things? For example, when listing multiple telephone numbers for a customer, would they be more usefully labeled separately as the customer's business phone number, home phone number, cell phone number, and so on?

Let’s look at another example. The sales database requirements may specify that a product has a name and a price. We can see that the product is an entity because it’s a distinct object. However, the product’s name and price aren’t distinct objects; they’re attributes that describe the product entity. Note that if we want to have different prices for different markets, then the price is no longer just related to the product entity, and we will need to model it differently.

For some applications, no combination of attributes can uniquely identify an entity (or it would be too unwieldy to use a large composite key), so we create an artificial attribute that’s defined to be unique and can therefore be used as a key: student numbers, Social Security numbers, driver’s license numbers, and library card numbers are examples of unique attributes created for various applications. In our inventory and sales application, it’s possible that we could stock different products with the same name and price. For example, we could sell two models of “Four-port USB 2.0 Hub,” both at $4.95 each. To distinguish between products, we can assign a unique product ID number to each item we stock; this would be the primary key. Each product entity would have name, price, and product ID attributes. This is shown in the ER diagram in Figure 2-3.

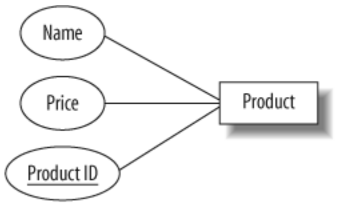


Figure 2-3. The ER diagram representation of the product entity

Representing Relationships

Entities can participate in relationships with other entities. For example, a customer can buy a product, a student can take a course, an employee can have an address, and so on.

Like entities, relationships can have attributes: we can define a sale to be a relationship between a customer entity (identified by the unique email address) and a given number of the product entity (identified by the unique product ID) that exists at a particular date and time (the timestamp).

Our database could then record each sale and tell us, for example, that at 3:13 p.m. on Wednesday, March 22, Marcos Albe bought one “Raspberry Pi 4”, one “500 GB SSD M.2 NVMe”, and two sets of “2000 Watt 5.1 Channel Sub-Woofer Speakers.”

Different numbers of entities can appear on each side of a relationship. For example, each customer can buy any number of products, and each product can be bought by any number of customers. This is known as a *many-to-many* relationship. We can also have *one-to-many* relationships. For example, one person can have several credit cards, but each credit card belongs to just one person. Looking at it the other way, a *one-to-many* relationship becomes a *many-to-one* relationship; for example, many credit cards belong to a single person. Finally, the serial number on a car engine is an example of a *one-to-one* relationship; each engine has just one serial number, and each serial number belongs to just one engine. We often use the shorthand terms *1:1*, *1:N*, and *M:N* for one-to-one, one-to-many, and many-to-many relationships, respectively.

The number of entities on either side of a relationship (the cardinality of the relationship) define the *key constraints* of the relationship. It’s important to think about the cardinality of relationships carefully. There are many relationships that may at first seem to be one-to-one, but turn out to be more complex. For example, people sometimes change their names; in some applications, such as police databases, this is of particular interest, and so it may be necessary to model a many-to-many relationship between a person entity and a name entity. Redesigning a database can be costly and time-consuming if you assume a relationship is simpler than it really is.

In an ER diagram, we represent a relationship set with a named diamond. The cardinality of the relationship is often indicated alongside the relationship diamond; this is the style we use in this book. (Another common style is to have an arrowhead on the line connecting the entity on the “1” side to the relationship diamond.) Figure 2-4 shows the relationship between the customer and product entities, along with the number and timestamp attributes of the sale relationship.

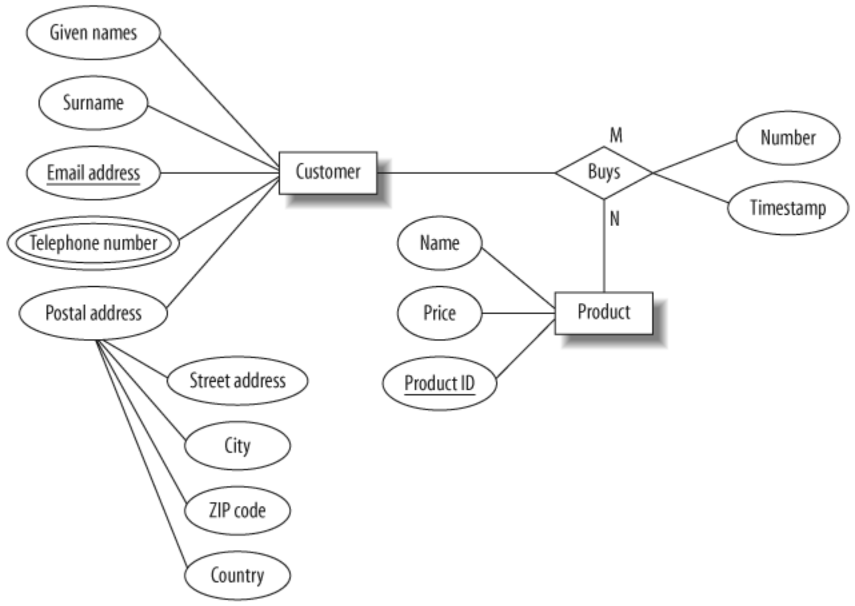


Figure 2-4. The ER diagram representation of the customer and product entities, and the sale relationship between them.

Partial and Total Participation

Relationships between entities can be optional or compulsory. In our example, we could decide that a person is considered to be a customer only if they have bought a product. On the other hand, we could say that a customer is a person whom we know about and whom we hope might buy something—that is, we can have people listed as customers in our database who never buy a product. In the first case, the `customer` entity has *total participation* in the bought relationship (all `customer`\s have bought a product, and we can't have a `customer` who hasn't bought a product), while in the second case it has *partial participation* (a `customer` can buy a product). These are referred to as the *participation constraints* of the relationship. In an ER diagram, we indicate total participation with a double line between the entity box and the relationship diamond.

Entity or Attribute?

From time to time, we encounter cases where we wonder whether an item should be an attribute or an entity on its own. For example, an email address could be modeled as an entity in its own right. When in doubt, consider these rules of thumb:

- *Is the item of direct interest to the database?*:: Objects of direct interest should be entities, and information that describes them should be stored in attributes. Our inventory and sales database is really interested in customers, and not their email addresses, so the email address would be best modeled as an attribute of the `customer` entity.

    1. *Does the item have components of its own?*:: If so, we must find a way of representing these components; a separate entity might be the best solution. In the student grades example at the start of the chapter, we stored the course name, year, and semester for each course that a student takes. It would be more compact to treat the course as a separate entity and to create a class ID number to identify each time a course is offered to students (the "offering").

    2. *Can the object have multiple instances?*:: If so, we must find a way to store data on each instance. The cleanest way to do this is to represent the object as a separate entity. In our sales

example, we must ask whether customers are allowed to have more than one email address; if they are, we should model the email address as a separate entity.

- *Is the object often nonexistent or unknown?*:: If so, it is effectively an attribute of only some of the entities, and it would be better to model it as a separate entity rather than as an attribute that is often empty. Consider a simple example: to store student grades for different courses, we could have an attribute for the student's grade in every possible course; this is shown in Figure 2-5. Because most students will have grades for only a few of these courses, it's better to represent the grades as a separate entity set, as in Figure 2-6.

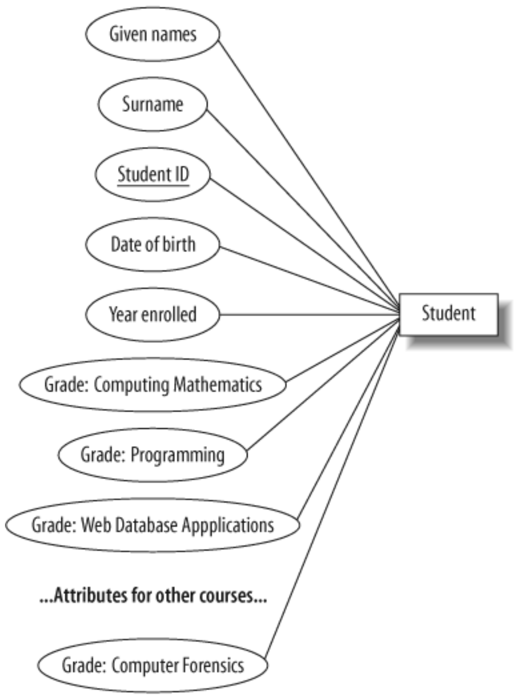

Given names
Surname
Student ID
Date of birth
Year enrolled
Student
Grade: Computing Mathematics
Grade: Programming
Grade: Web Database Applications
...Attributes for other courses...
Grade: Computer Forensics

Figure 2-5. The ER diagram representation of student grades as attributes of the student entity

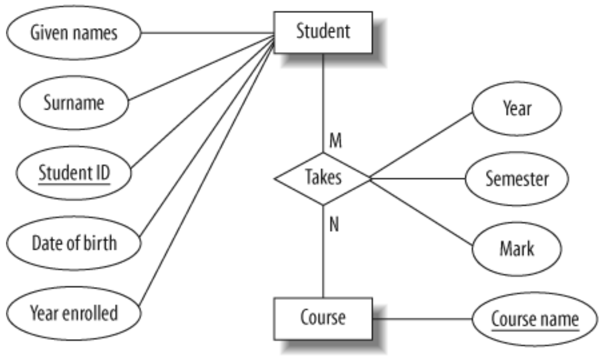


Figure 2-6. The ER diagram representation of student grades as a separate entity

Entity or Relationship?

An easy way to decide whether an object should be an entity or a relationship is to map nouns in the requirements to entities, and map verbs to relations. For example, in the statement "A degree program is made up of one or more courses," we can identify the entities "program" and "course," and the relationship "is made up of." Similarly, in the statement "A student enrolls in one program," we can identify the entities "student" and "program," and the relationship "enrolls in." Of course, we can choose different terms for entities and relationships than those that appear in the relationships, but it's a good idea not to deviate too far from the naming conventions used in the requirements so that the design can be checked against the requirements. All else being equal, try to keep the design simple, and avoid introducing trivial entities where possible; that is, there's no need to have a separate entity for the student's enrollment when we can model it as a relationship between the existing student and program entities.

Intermediate Entities

It is often possible to conceptually simplify many-to-many relationships by replacing the many-to-many relationship with a new *intermediate* entity (sometimes called an *associate* entity) and connecting the original entities through a many-to-one and a one-to-many relationship.

Consider this statement: "A passenger can book a seat on a flight." This is a many-to-many relationship between the entities "passenger" and "flight." The related ER diagram fragment is shown in Figure 2-7.

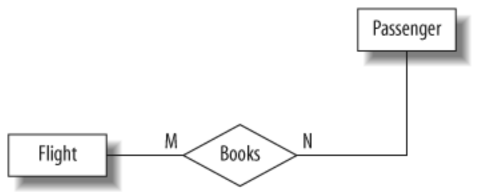


Figure 2-7. A passenger participates in an M:N relationship with flight

However, let's look at this from both sides of the relationship:

- Any given flight can have many passengers with a booking.
- Any given passenger can have bookings on many flights.

Hence, we can consider the many-to-many relationship to be in fact two one-to-many relationships, one each way. This points us to the existence of a hidden intermediate entity, the booking, between the flight and the passenger entities. The requirement could be better worded as: "A passenger can make a booking for a seat on a flight." The related ER diagram fragment is shown in Figure 2-8.

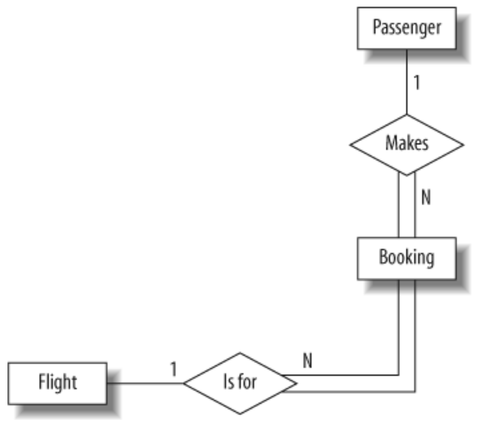


Figure 2-8. The intermediate booking entity between the passenger and flight entities

Each passenger can be involved in multiple bookings, but each booking belongs to a single passenger, so the cardinality of this relationship is 1:N. Similarly, there can be many bookings for a given flight, but each booking is for a single flight, so this relationship also has cardinality 1:N. Since each booking must be associated with a particular passenger and flight, the booking entity participates totally in the relationships with these entities. This total participation could not be captured effectively in the representation in Figure 2-7. (We described partial and total participation earlier in "Partial and Total Participation".)

Weak and Strong Entities

Context is very important in our daily interactions; if we know the context, we can work with a much smaller amount of information. For example, we generally call family members by only their first name or nickname. Where ambiguity exists, we add further information such as the surname to clarify our intent. In database design, we can omit some key information for entities that are dependent on other entities. For example, if we wanted to store the names of our customers' children, we could create a child entity and store only enough key information to identify it in the context of its parent. We could simply list a child's first name on the assumption that a customer will never have several children with the same first name. Here, the child entity is a weak entity, and its relationship with the customer entity is called an identifying relationship. Weak entities participate *totally* in the identifying relationship, since they can't exist in the database independently of their owning entity.

In the ER diagram, we show weak entities and identifying relationships with double lines, and the partial key of a weak entity with a dashed underline, as in Figure 2-9. A weak entity is uniquely identified in the context of its regular (or strong) entity, and so the full key for a weak entity is the combination of its own (partial) key with the key of its owning entity. To uniquely identify a child in our example, we need the first name of the child and the email address of the child's parent.

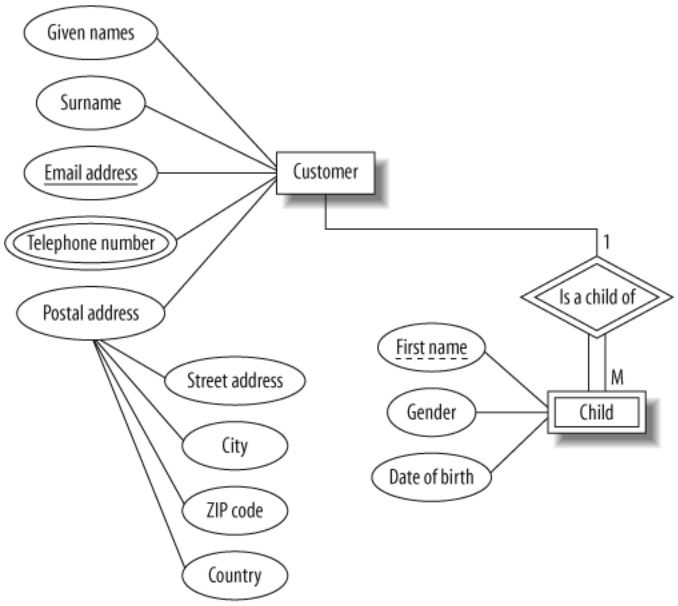


Figure 2-9. The ER diagram representation of a weak entity

Figure 2-10 shows a summary of the symbols we've explained for ER diagrams.

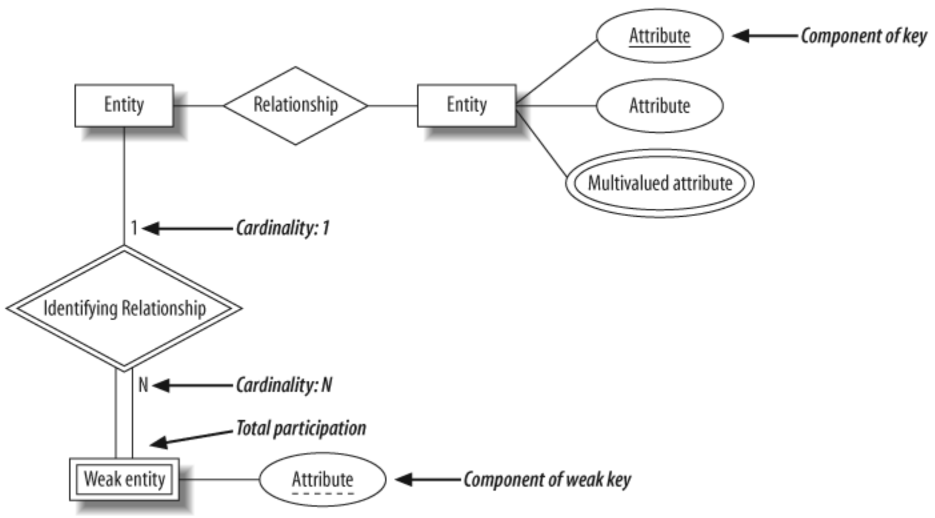


Figure 2-10. Quick summary of the ER diagram symbols

Database Normalization

Database normalization is an important concept when designing the relational data structure. Dr. Edgar F. Codd, the inventor of the relational database model, proposed the normal forms in the early ’70s, which is widely used by the industry nowadays. Even with the advent of the NoSQL databases, there is no evidence in the short or medium-term that relational databases will disappear, or that the normal forms will fall in disuse.

The main objective of the normal norms is to reduce data redundancy and improve data integrity. It also facilitates the process of redesigning and extending the database structure.

Officially, there are six normal forms, but most database architects only deal with the first three forms. That is because the normalization process is progressive, and we cannot achieve a higher level of database normalization unless the previous levels have been satisfied. Using all the six norms constricts too much of the database model, and in general, they become very complex to implement. In real workloads, usually, there are performance issues. This is one reason for *ETL* jobs to exist (ETL jobs denormalize the data to process it).

Let’s take a look at the first three normal forms.

1NF: First normal form

- Eliminate repeating groups in individual tables
- Create a separate table for each set of related data
- Identify each set of related data with a primary key

If a relation contains composite or multi-valued attribute, it violates the first normal form, or a relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is single-value attribute.

2NF: Second normal form

- Create separate tables for sets of values that apply to multiple records
- Relate these tables with a foreign key

Records should not depend on anything other than a table’s primary key (a compound key, if necessary).

3NF: Third normal form

- Eliminate fields that do not depend on the key

Values in a record that are not part of that record’s key do not belong in the table. In general, any time the contents of a group of fields may apply to more than a single record in the table, consider placing those fields in a separate table.

The following table lists the normal forms from the least normalized to the most normalized:

	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	EKNF (1982)	BCNF (1974)	4N
Primary key (no duplicate tuples)		Yes	Yes	Yes	Yes	Yes	Yes
No repeating groups		Yes	Yes	Yes	Yes	Yes	Yes
Atomic columns (cells have single value)	No	Yes	Yes	Yes	Yes	Yes	Yes
Every nontrivial functional dependency either does not begin with a proper subset of a candidate key or ends with a prime attribute (no partial functional dependencies of nonprime attributes on candidate keys)	No	No	Yes	Yes	Yes	Yes	Yes
Every nontrivial functional dependency begins with a superkey or ends with a prime attribute (no transitive functional dependencies of nonprime attributes on candidate keys)	No	No	No	Yes	Yes	Yes	Yes
Every nontrivial functional dependency either begins with a superkey or ends with an elementary prime attribute	No	No	No	No	Yes	Yes	Yes
Every nontrivial functional dependency begins with a superkey	No	No	No	No	No	Yes	Yes
Every nontrivial multivalued dependency begins with a superkey	No	No	No	No	No	No	Yes
Every join dependency has a superkey component	No	No	No	No	No	No	No
Every join dependency has only superkey components	No	No	No	No	No	No	No
Every constraint is a consequence of	No	No	No	No	No	No	No

domain constraints and key constraints							
Every join dependency is trivial	No	No	No	No	No	No	No

Normalizing an Example Table

Let’s see an example of normalizing a fictional student table.

Unnormalized table:

```
  Student#        Advisor Adv-Room        Class1  Class2  Class3
  1022        Jones          412      101-07      143-01  159-02
  4123        Smith          216      201-01      211-02  214-01
```

First normal form: No repeating groups

Tables should have only two dimensions. Since one student has several classes, these classes should be listed in a separate table. Fields Class1, Class2, and Class3 in the above records are indications of design trouble.

Spreadsheets often use the third dimension, but tables should not. Here’s another way to look at this problem: with a one-to-many relationship, don’t put the one side and the many side in the same table. Instead, create another table in first normal form by eliminating the repeating group (Class#), as shown here:

```
  Student#        Advisor Adv-Room        Class#
  1022        Jones          412      101-07
  1022        Jones          412      143-01
  1022        Jones          412          159-02
  4123        Smith          216      201-01
  4123        Smith          216      211-02
  4123        Smith          216      214-01
```

Second normal form: Eliminate redundant data

Note the multiple `Class#` values for each `Student#` value in the previous table. `Class#` is not functionally dependent on `Student#` (primary key), so this relationship is not in second normal form.

The following two tables demonstrate second normal form:

Students:

```
  Student#        Advisor Adv-Room
  1022        Jones          412
  4123        Smith       216
```

Registration:

```
  Student#        Class#
  1022        101-07
  1022        143-01
  1022        159-02
  4123        201-01
  4123        211-02
  4123        214-01
```

Third normal form: Eliminate data not dependent on key

In the last example, `Adv-Room` (the advisor's office number) is functionally dependent on the `Advisor` attribute. The solution is to move that attribute from the Students table to the Faculty table, as shown next.

Students:

```
Student#        Advisor
1022        Jones
4123        Smith
```

Faculty:

```
Name     Room     Dept
Jones    412      42
Smith    216      42
```

Entity Relationship Modeling Examples

Earlier in this chapter, we showed hypothetical cases to design a database and to understand an Entity-Relationship (ER) diagram and the normalization rules. Now we are going to see some ER examples from sample databases available for MySQL. To visualize the ER, we are going to use *MySQL Workbench*.

MySQL Workbench uses a physical ER representation. Physical ER diagram models are more granular, showing the processes necessary to add information to a database. Rather than using symbols, we use tables in the ER, making it closer to the real database. MySQL Workbench goes one step further and uses an enhanced entity-relationship (EER). EER diagrams are an expanded version of ER diagrams.

We are not going into the details of the EER, but the advantages of an EER diagram is that it provides all the elements of an ER diagram while adding the following:

- Attribute or relationship inheritances
- Category or union types
- Specialization and generalization
- Subclasses and superclasses

Let's start with the process to download the samples and visualize its EER in MySQL Workbench.

The first one is the Sakila database. Development of the Sakila sample database began in early 2005. Early designs were based on the Dell whitepaper database, which was used in the Dell whitepaper Three Approaches to MySQL Applications on Dell PowerEdge Servers.

Where Dell's sample database was designed to represent an online DVD store, the Sakila sample database is designed to represent a DVD rental store. The Sakila sample database still borrows film and actor names from the Dell sample database. The following commands will import the Sakila database to our MySQL instance:

```
# wget https://downloads.mysql.com/docs/sakila-db.tar.gz
# tar -xvf sakila-db.tar.gz
# mysql -uroot -pmsandbox < sakila-db/sakila-schema.sql
# mysql -uroot -pmsandbox < sakila-db/sakila-data.sql
```

Sakila also provides the EER model (*sakila.mwb* file). We can open the file with MySQL Workbench as shown in Figure 2-11.

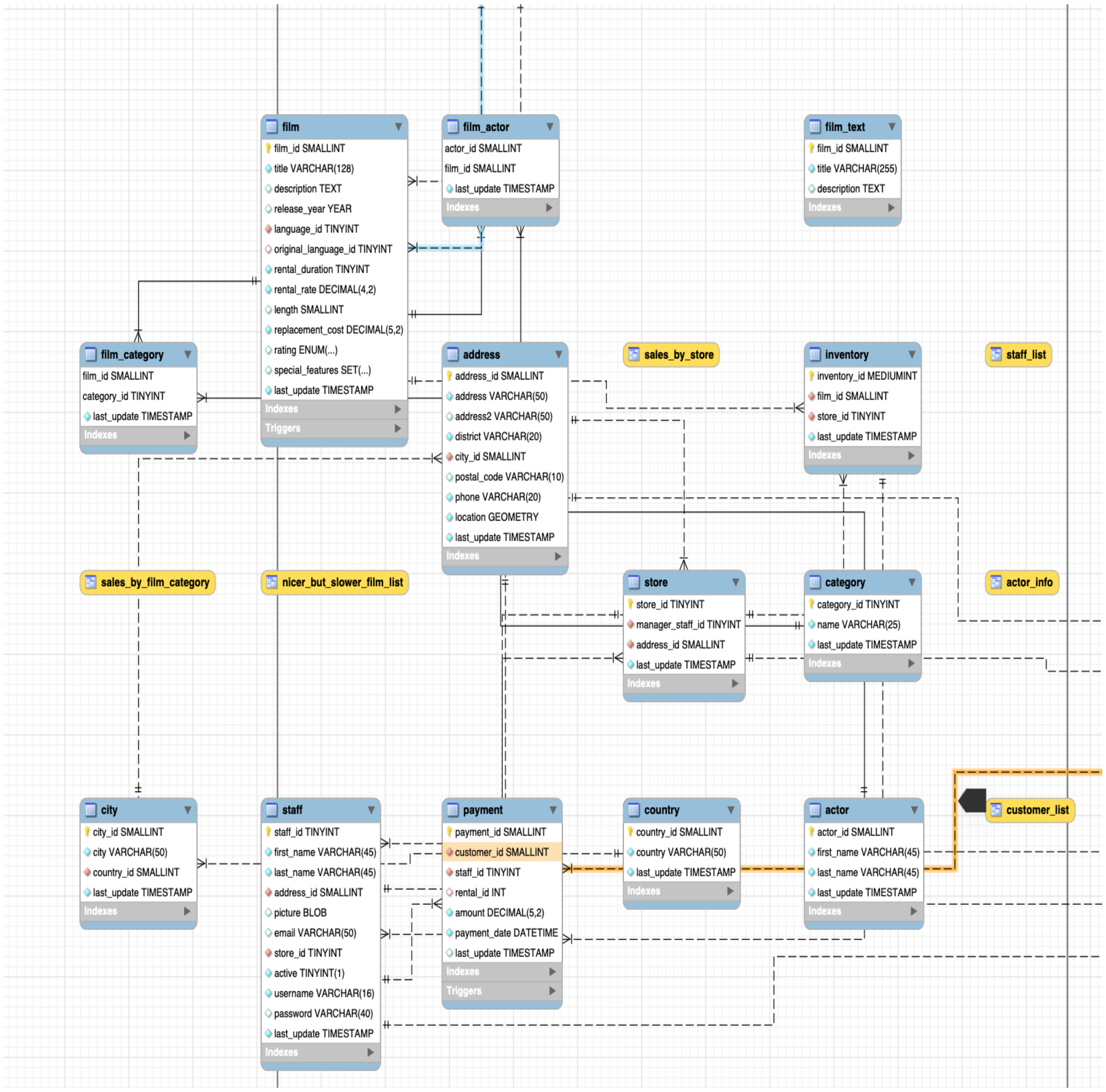


Figure 2-11. Sakila database EER, note the physical representation of the entities instead of using symbols

Next is the World database. Another database available from Oracle, the sample data used in the World database is Copyright Statistics Finland.

The following commands will import the World database to our MySQL instance:

```
# wget https://downloads.mysql.com/docs/world.sql.gz
# zcat world.sql.gz | mysql -uroot -pmsandbox
```

The World database does not come with the EER file as Sakila does. But we can create the EER model from the real database using MySQL Workbench using reverse engineering. We need to click the Databases menu and then Reverse Engineer as in

Figure 2-12:

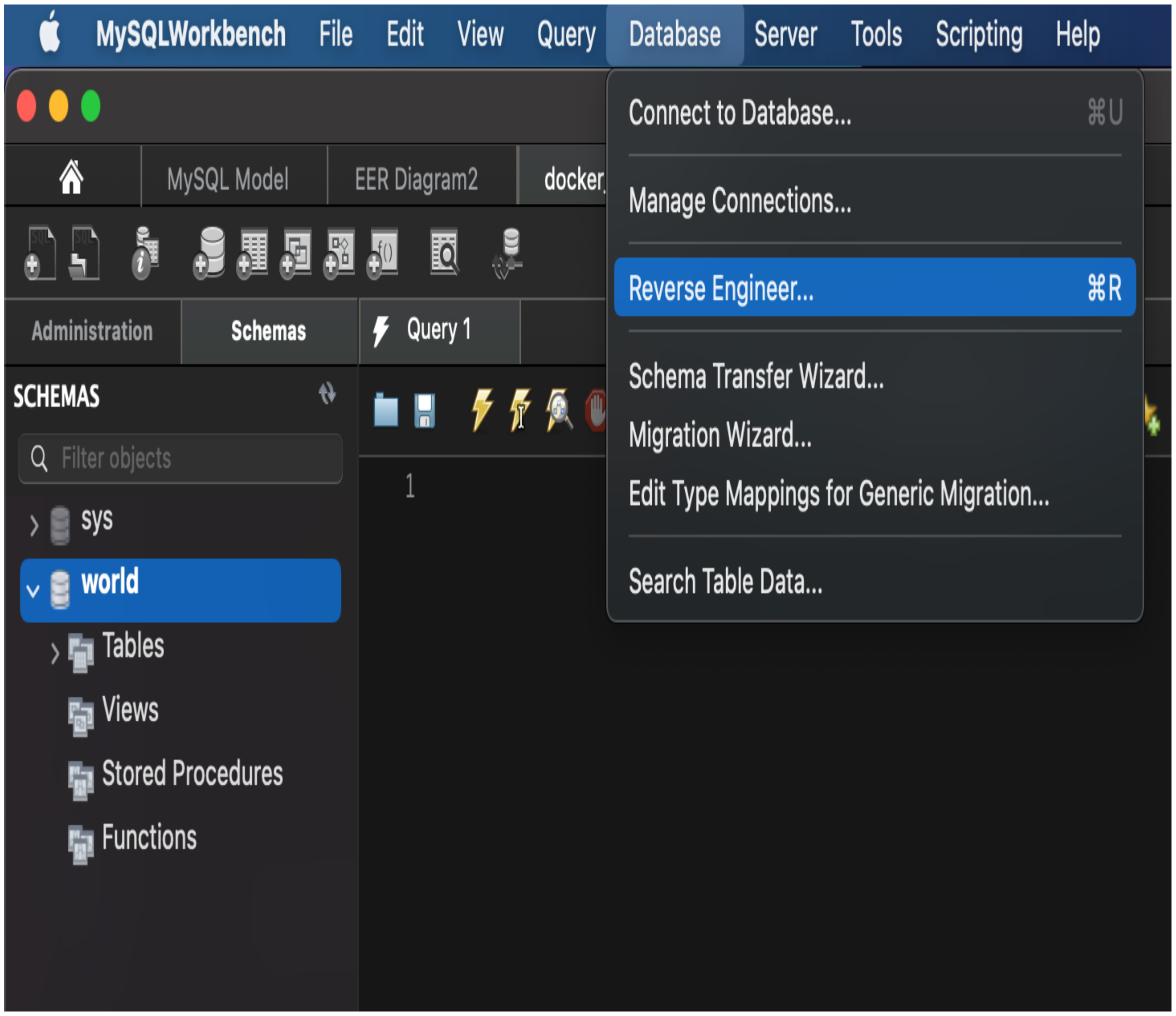
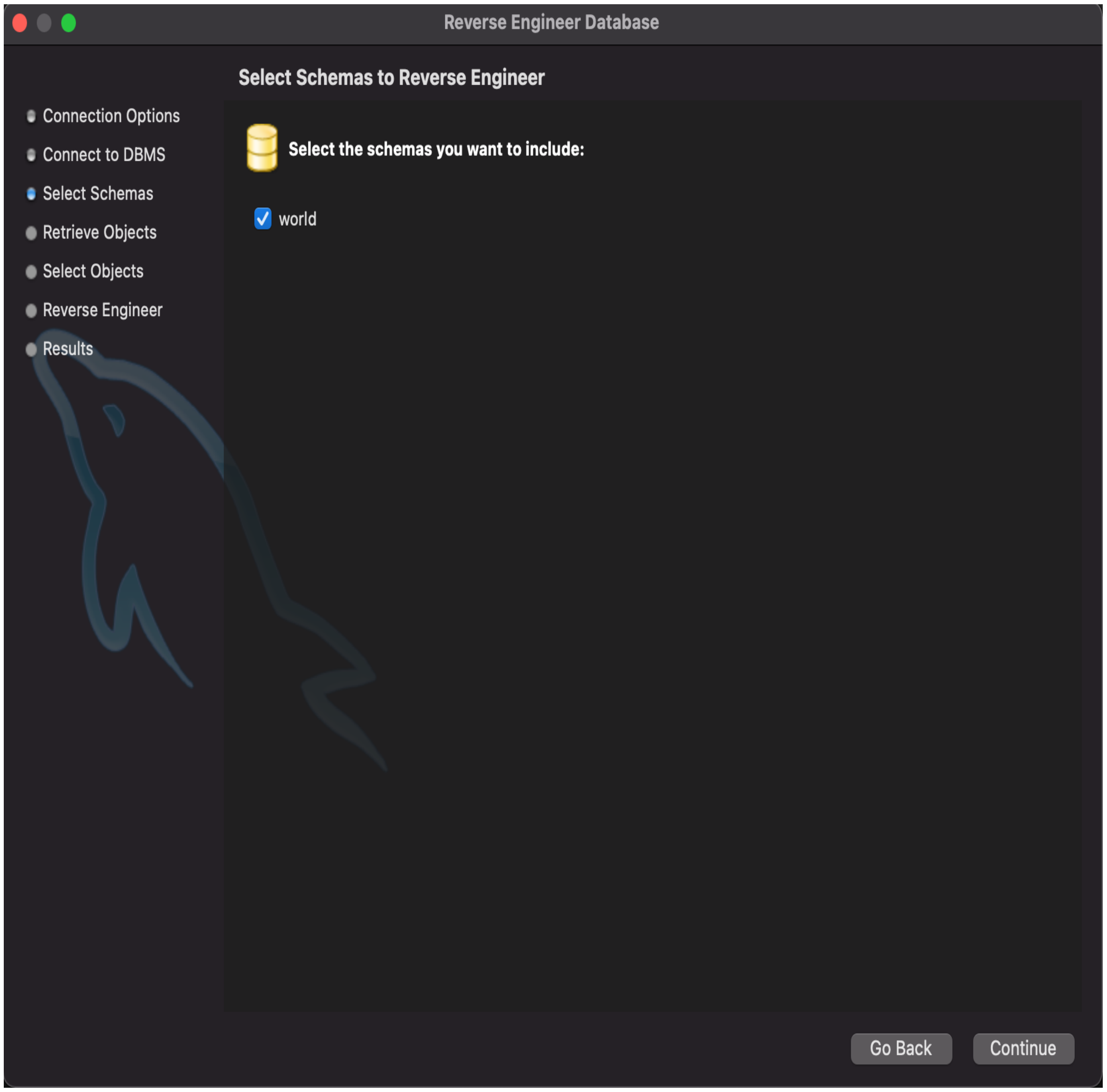


Figure 2-12. Reverse engineering from World database

Then, Workbench will connect to the database (if not connected already) and prompt us to choose the schema we want to reverse to ER as show in Figure 2-13:



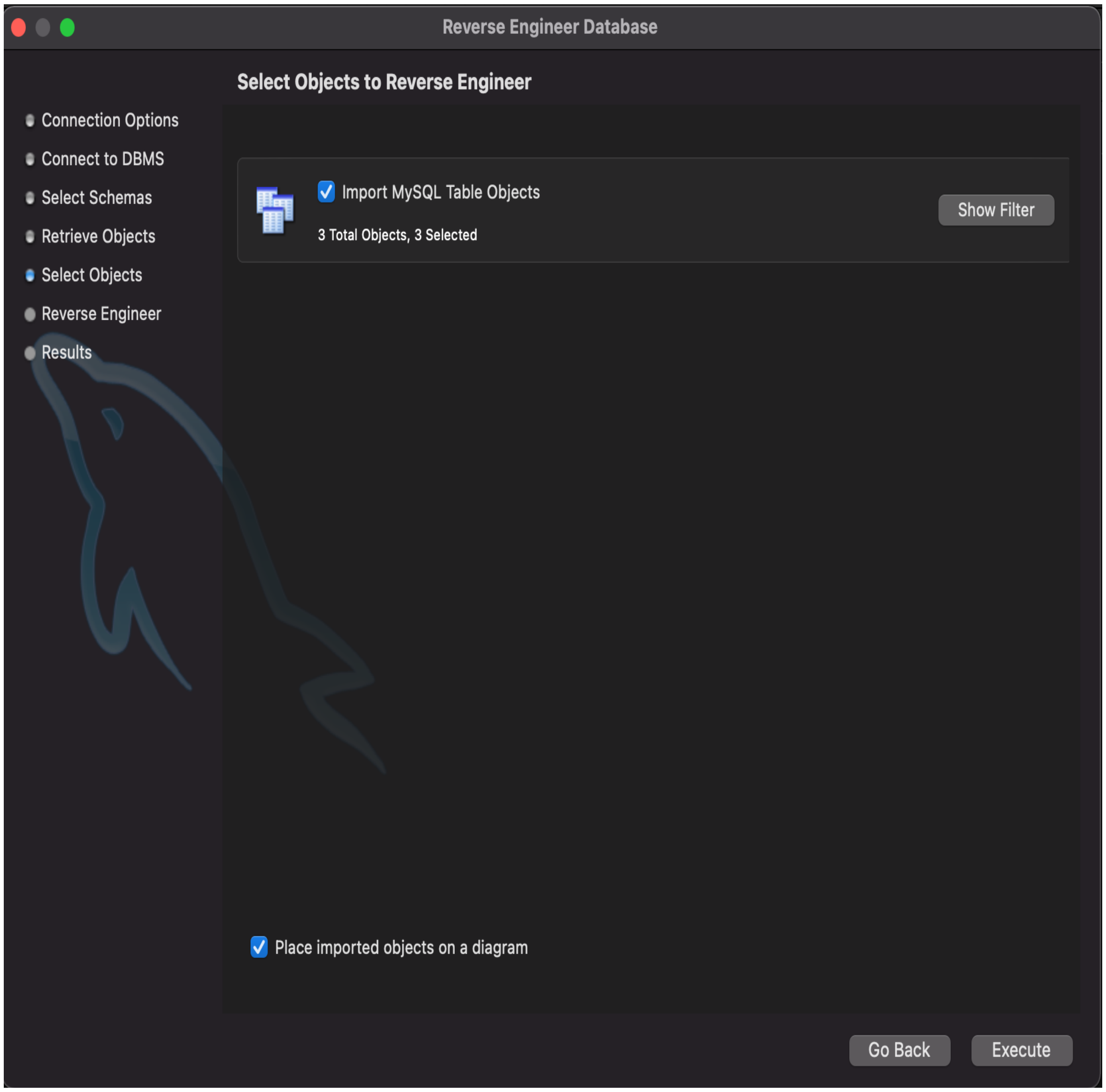
Select Schemas to Reverse Engineer

Select the schemas you want to include:

☑ world

Figure 2-13. Choosing the schema we want to see the ER

The last screen is to confirm its execution as shown in Figure 2-14:



Select Objects to Reverse Engineer

☑ Import MySQL Table Objects

3 Total Objects, 3 Selected

☑ Place imported objects on a diagram

Figure 2-14. Press Execute to start

And we have our ER from the World database on Figure 2-15:

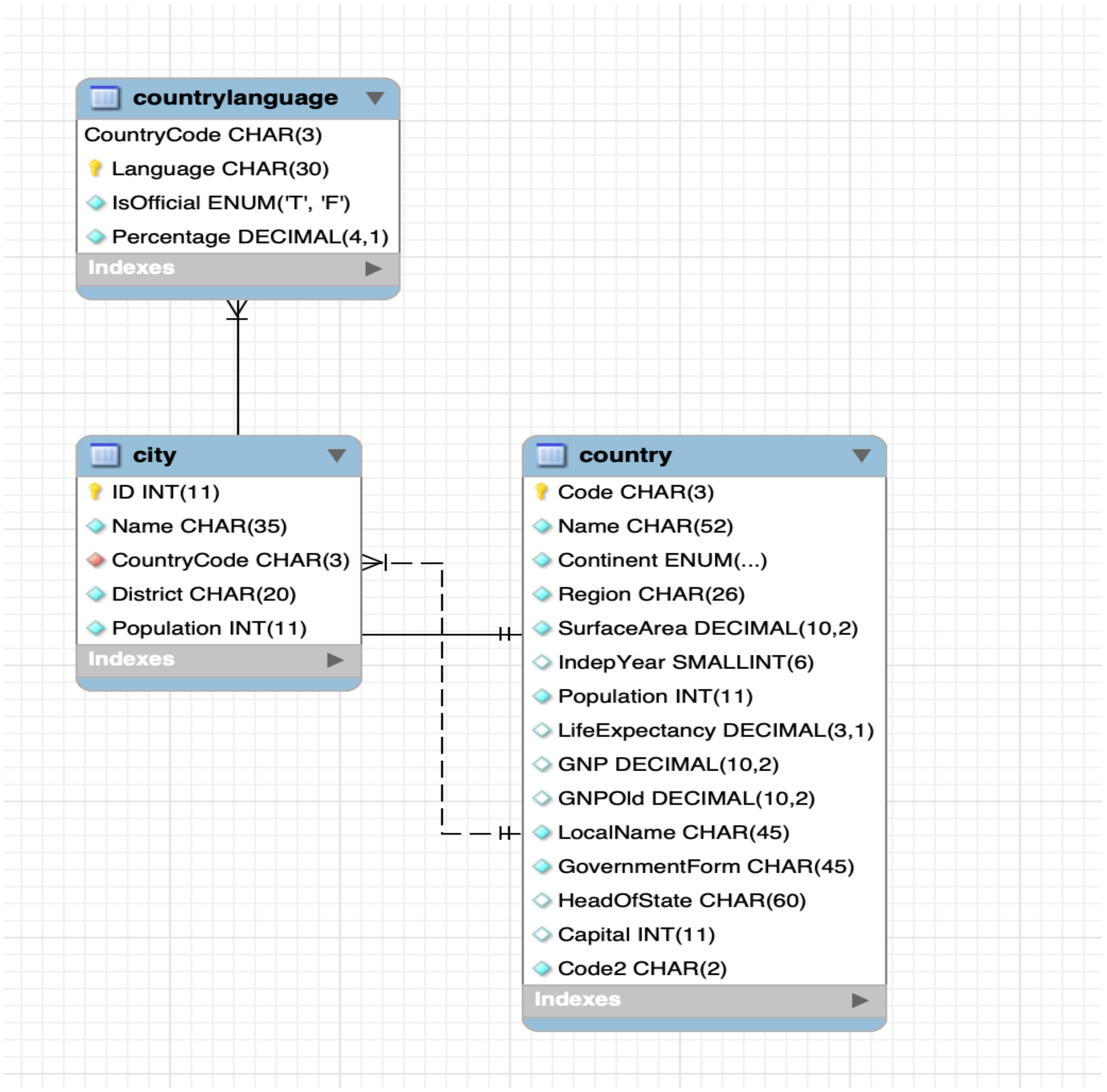


Figure 2-15. The ER from the World database

The last database we will import is the Employee database. To give proper credit, we need to mention the creators and supporters of this database. Fusheng Wang and Carlo Zaniolo created the original data at Siemens Corporate Research. Giuseppe Maxia made the relational schema, and Patrick Crews exported the data in relational format.

To import the data, first, we need to clone the Git repository:

```
# git clone https://github.com/datacharmer/test_db.git
# cd test_db
# cat employees.sql | mysql -uroot -psekret
```

And we can use the reverse engineering procedure from MySQL Workbench to create the ER model for the Employee database as shown in Figure 2-16:

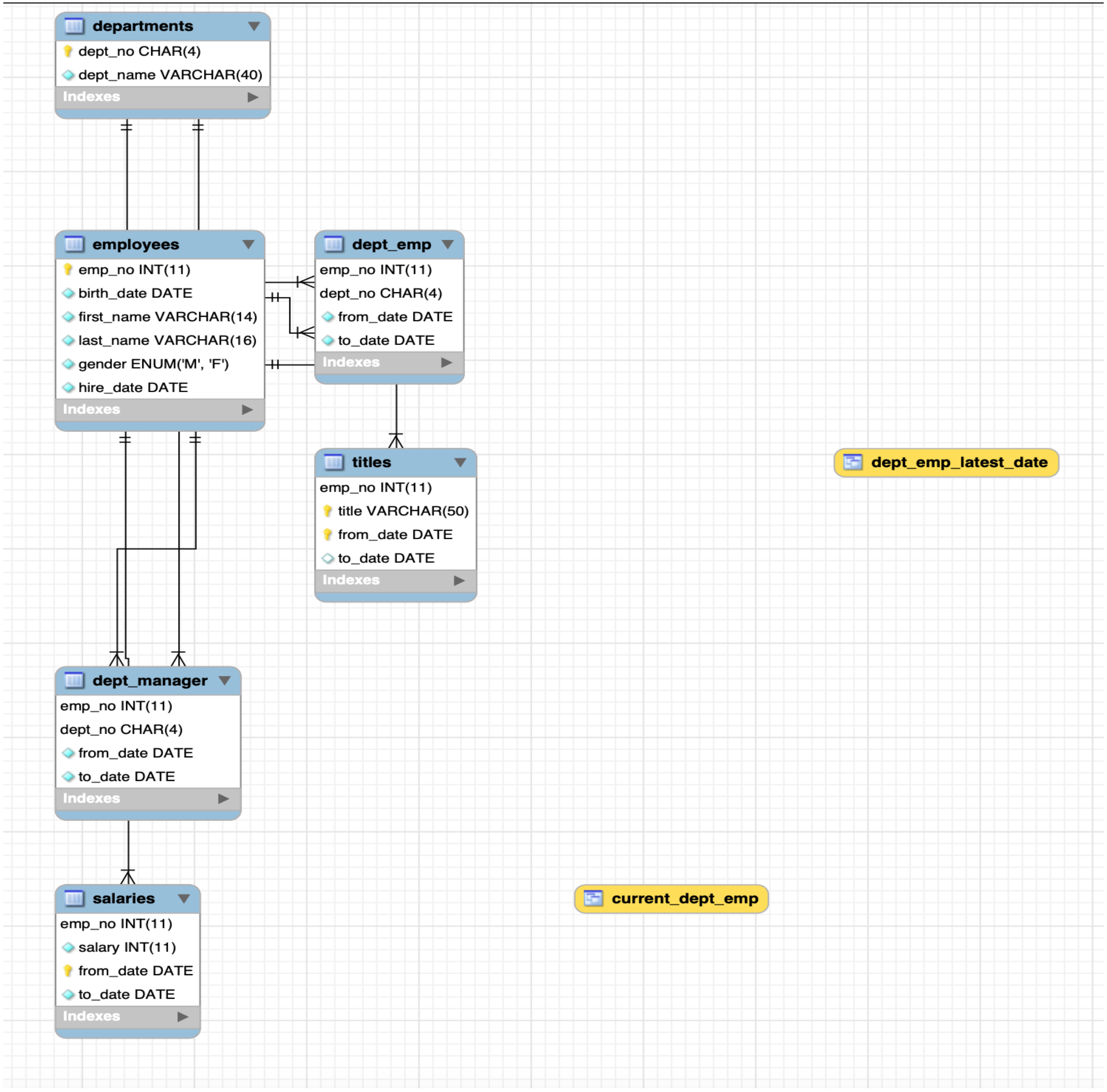


Figure 2-16. The ER from the Employee database

You'll find that having an overview of the ER diagrams and the database designs' explanations is sufficient to work with the material in this chapter. We'll show you how to create a database on your MySQL server in Chapter 3. Also, during our explanation of the *CRUD* statements, we will cover again the theory we saw in this chapter, especially when performing JOIN ("Joining Two Tables") operations.

Using the Entity Relationship Model

This section looks at the steps required to create an ER model and deploy it into database tables. We saw previously that MySQL Workbench lets us reverse engineer an existing database. But how do we model a new database and deploy it? We can automate this process with the MySQL Workbench tool.

Mapping Entities and Relationships to Database Tables

When converting an ER model to a database schema, we work through each entity and then through each relationship according to the rules discussed in the following sections to end up with a set of database tables.

Map the entities to database tables

For each strong entity, create a table comprising its attributes and designate the primary key. The parts of any composite attributes are also included here.

For each weak entity, create a table comprising its attributes and including the primary key of its owning entity. The owning entity’s primary key is a foreign key here because it’s a key not of this table but another table. The table’s primary key for the weak entity is the combination of the foreign key and the partial key of the weak entity. If the relationship with the owning entity has any attributes, add them to this table.

For each entity’s multivalued attribute, create a table comprising the entity’s primary key and the attribute.

Map the relationships to database tables

Each one-to-one relationship between two entities includes the primary key of one entity as a foreign key in the table belonging to the other. If one entity participates totally in the relationship, place the foreign key in its table. If both participate totally in the relationship, consider merging them into a single table.

For each non-identifying one-to-many relationship between two entities, including the entity’s primary key on the “1” side as a foreign key in the table for the entity on the “N” side. Add any attributes of the relationship in the table alongside the foreign key. Note that identifying one-to-many relationships (between a weak entity and its owning entity) are captured as part of the entity-mapping stage.

For each many-to-many relationship between two entities, create a new table containing each entity’s primary key as the primary key and add any attributes of the relationship. This step helps to identify intermediate entities.

For each relationship involving more than two entities, create a table with the primary keys of all the participating entities, and add any relationship attributes.

Creating a Bank Database ER Model

We’ve discussed database models for employees, sales, and CRM. Let’s see how we could model a bank database.

Following the mapping rules as just described, we first map entities to database tables. We collected all the requisites with the stakeholders, we defined our requirements for the online banking system, and we decided we need to have the following entities:

- Employees
- Branches
- Customers
- Accounts

Once we’ve identified the entities, we are going to create the tables and attributes for each table. We established primary keys to ensure every table has a unique identifier column for its records. Following, we need to define the relationships between the tables.

Many to many relationships (N:M):

We’ve established this type of relationship between Branches and Employees, Accounts, and Customers. An employee can work for any number of branches, and a branch could have any number of employees. Similarly, a customer could have many accounts, and an account could be a joint account between more than two customers.

To solve these relationships, we need two more intermediate entities. We create them as follows:

- Account_Customers
- Branch_Employees

Since the source table's primary key is also the primary key of the join table, the relation was thought to be an identifying relation.

One to many relationship (1:N)

This type of relation was established between Branches and Accounts, and customer and banking transactions. These relationships were non-identifying relationships as the parent table's primary key is only used as part of the foreign key in the child table.

Because we will work on a physical EER, we are also going to define the Primary Keys. It is common and recommended to use auto increment and unsigned fields for the Primary Key.

The Figure 2-17 shows the final representation of the bank model.

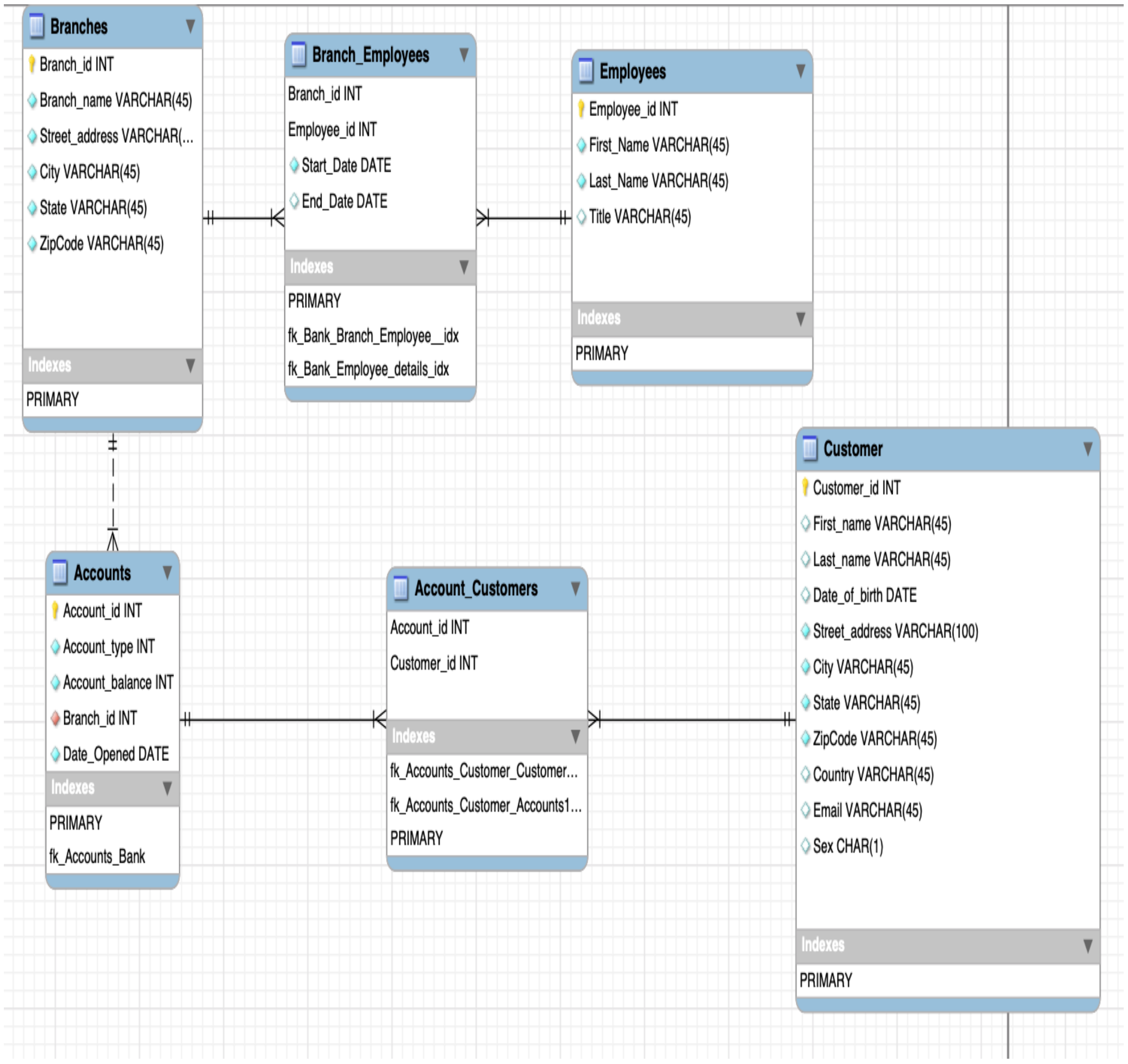


Figure 2-17. The EER model for the bank database

Note that there are items we haven't considered for this model. For example, our model does not support the inputting of family members of the employees. The model also does not support a customer with multiple addresses (work address and home address, for example). We did this intentionally to emphasize the importance of collecting the requisites prior to the database deployment.

You can download the model at the GitHub repository. The file is *bank_model.mwb*.

Converting the EER to a MySQL Database Using Workbench

It's a good idea to use a tool to draw our ER diagrams; this way, we can easily edit diagrams and refine our design until the final diagram is clear and unambiguous. Once we are comfortable with the model, we can deploy it. MySQL workbench allows the conversion of the EER model into DDL statements.

To convert the EER model into *DDL* statements to create a MySQL database, first, click the Database option, and then Forward Engineer, as shown in Figure 2-18:

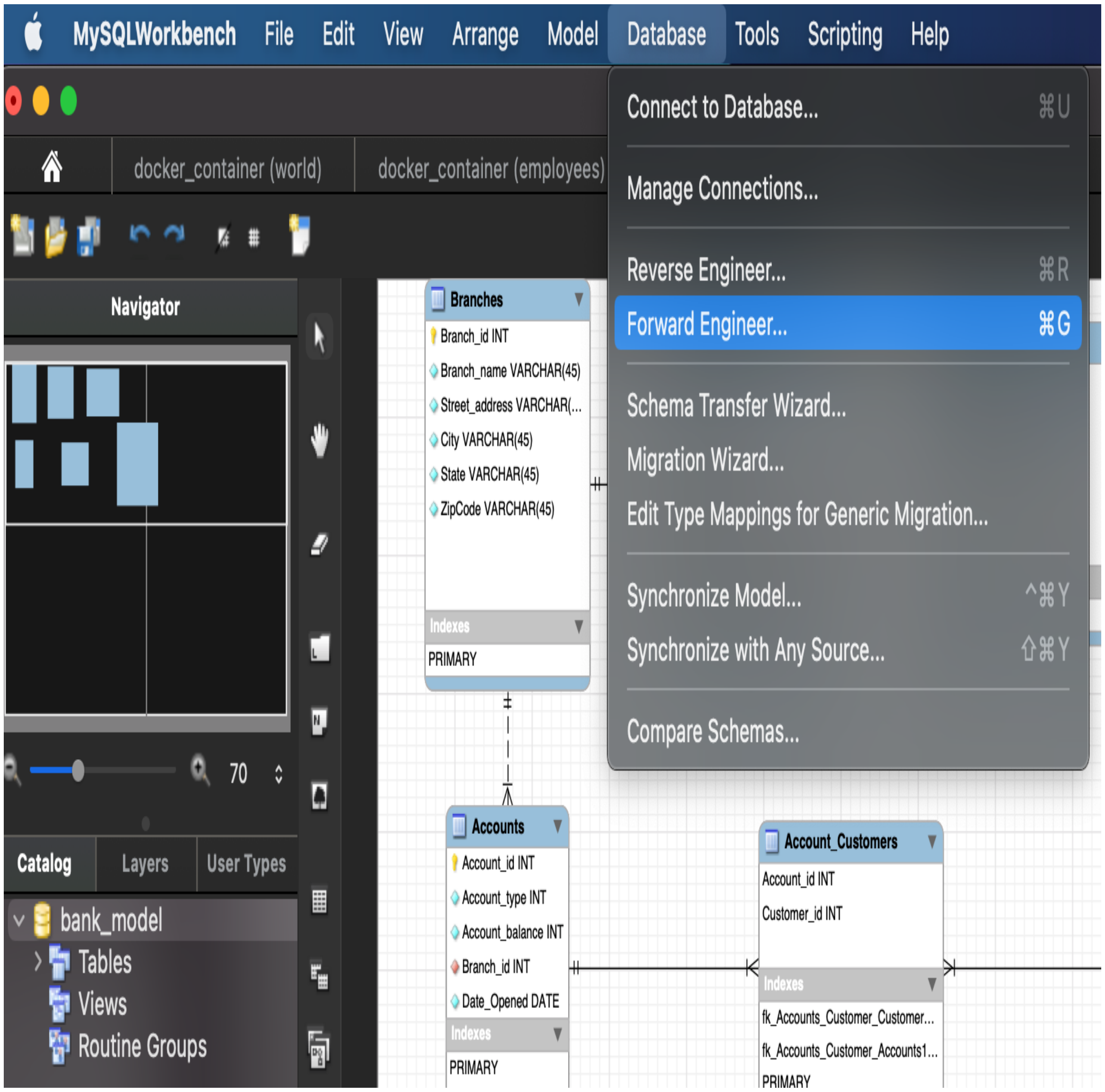
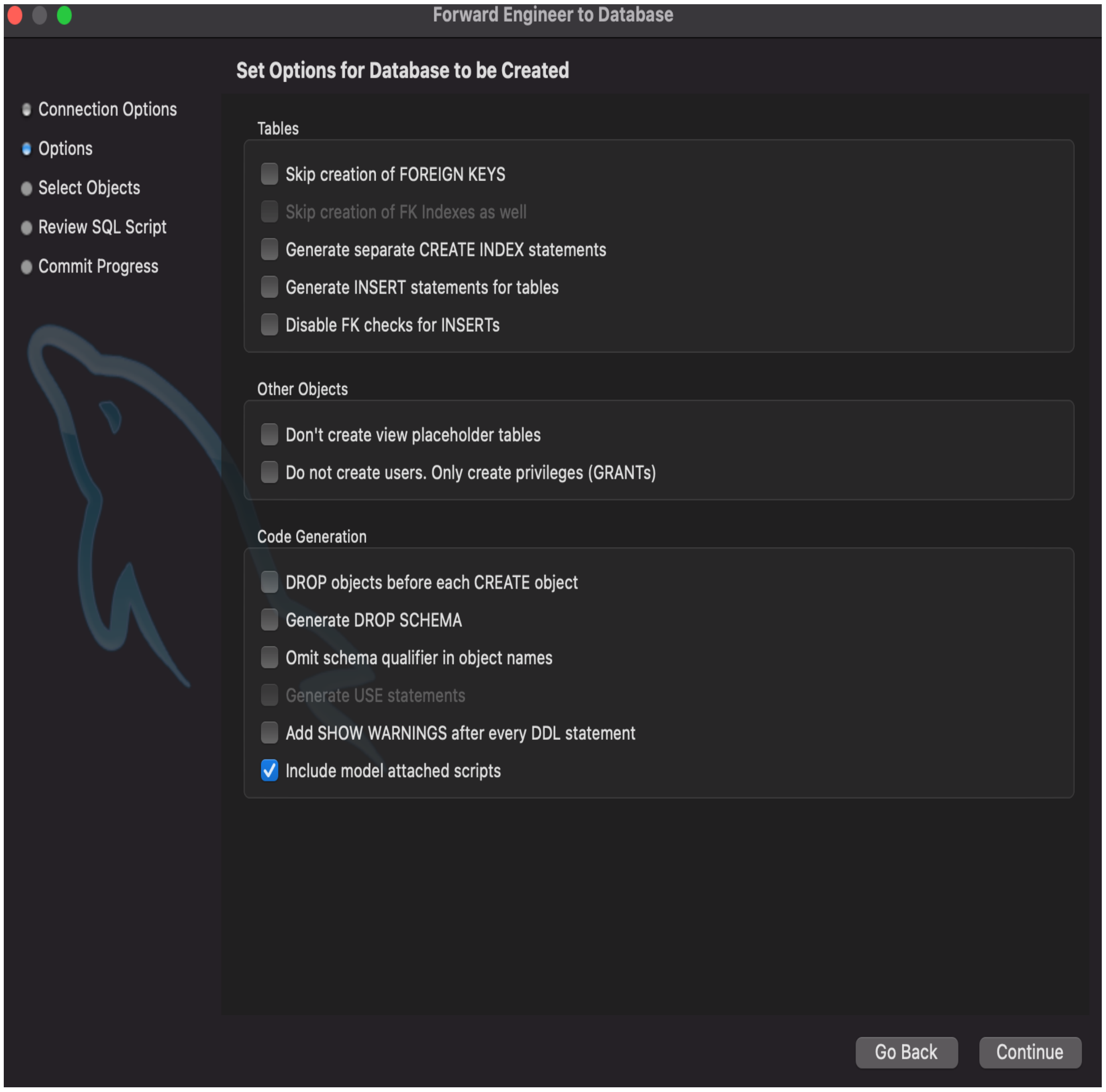


Figure 2-18. Finding the Forward Engineer option

We need to enter the credentials to connect to the database, and after that MySQL Workbench will present some options. For this model, we are going to use the standard options as shown in Figure 2-19:



Set Options for Database to be Created

Tables

- [ ] Skip creation of FOREIGN KEYS
- [ ] Skip creation of FK Indexes as well
- [ ] Generate separate CREATE INDEX statements
- [ ] Generate INSERT statements for tables
- [ ] Disable FK checks for INSERTs

Other Objects

- [ ] Don't create view placeholder tables
- [ ] Do not create users. Only create privileges (GRANTs)

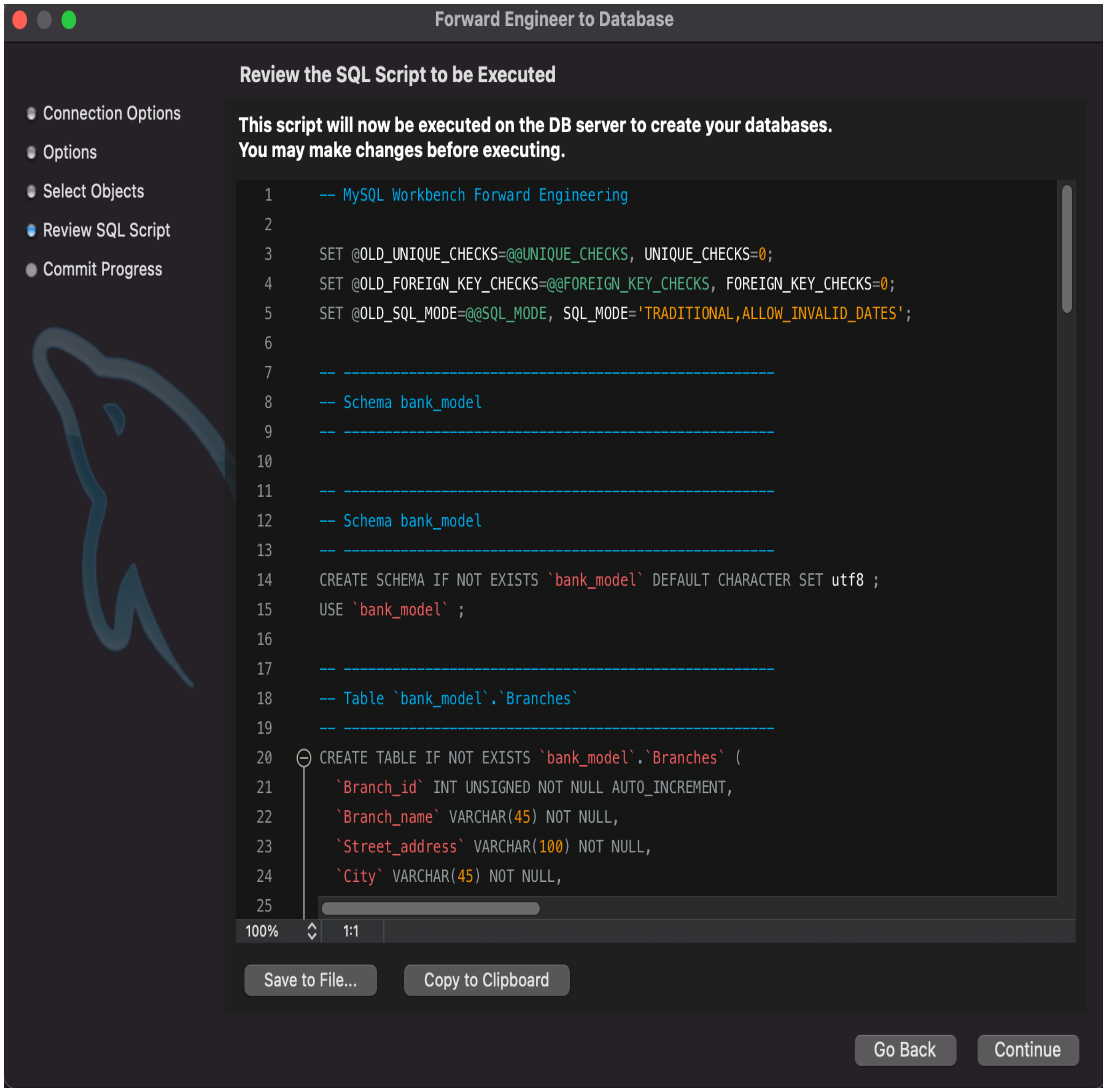
Code Generation

- [ ] DROP objects before each CREATE object
- [ ] Generate DROP SCHEMA
- [ ] Omit schema qualifier in object names
- [ ] Generate USE statements
- [ ] Add SHOW WARNINGS after every DDL statement
- [✓] Include model attached scripts

Figure 2-19. Setting the options

The Select Objects` option will ask which elements of the model we want to generate. Since we do not have anything special like triggers, stored procedures, users, and so on, we will only create the table objects and their relation; the rest of the options are zeroed.

MySQL workbench will present us with the SQL statements of our model, as shown in Figure 2-20:



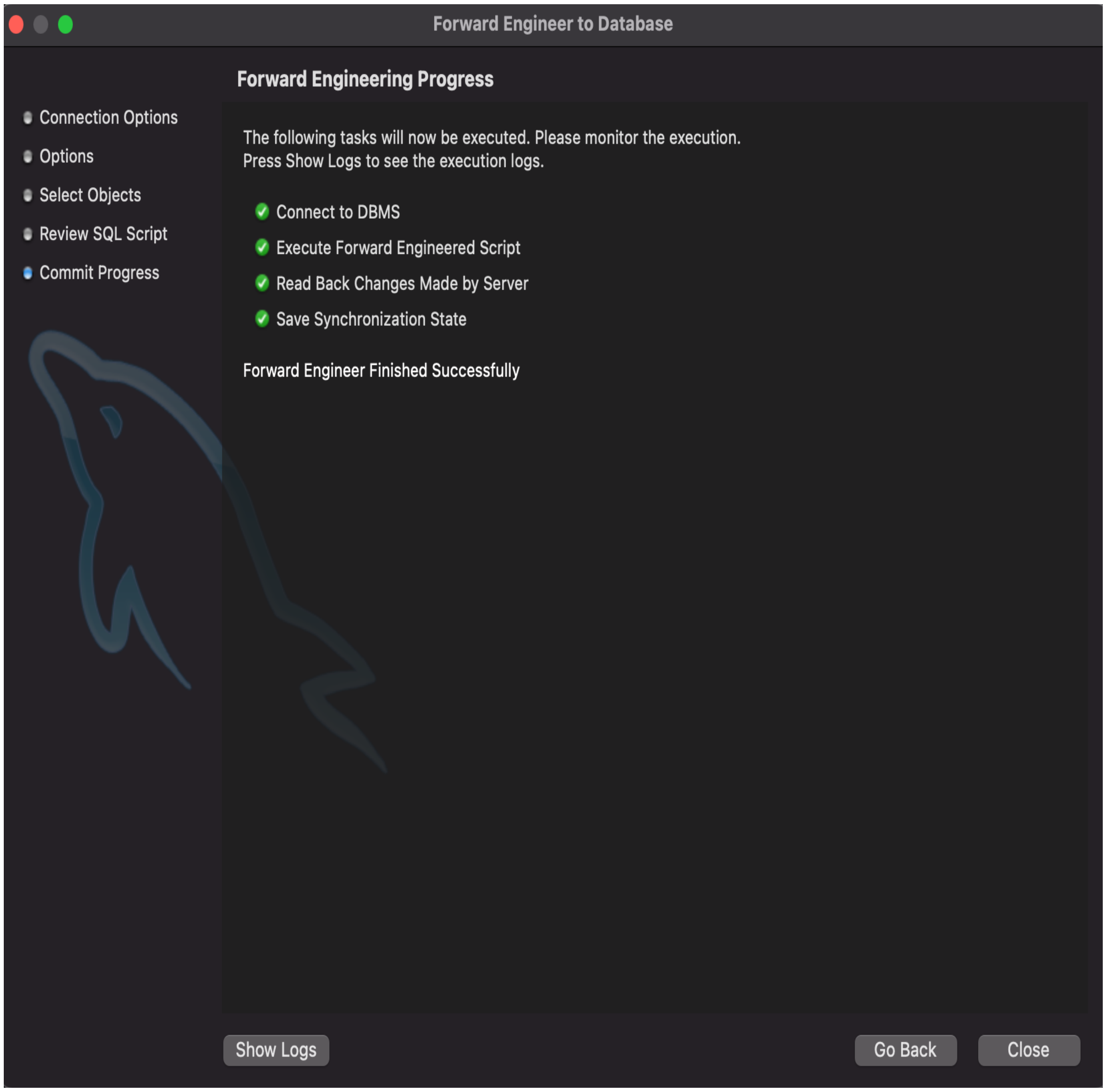
Review the SQL Script to be Executed

This script will now be executed on the DB server to create your databases.
You may make changes before executing.

```
-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

-- -----------------------------------------------------
-- Schema bank_model
-- -----------------------------------------------------


-- -----------------------------------------------------
-- Schema bank_model
-- -----------------------------------------------------
CREATE SCHEMA IF NOT EXISTS `bank_model` DEFAULT CHARACTER SET utf8 ;
USE `bank_model` ;

-- -----------------------------------------------------
-- Table `bank_model`.`Branches`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `bank_model`.`Branches` (
  `Branch_id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `Branch_name` VARCHAR(45) NOT NULL,
  `Street_address` VARCHAR(100) NOT NULL,
  `City` VARCHAR(45) NOT NULL,
```

Figure 2-20. We can see all the required DML statements to create the database

And the last step, MySQL Workbench will execute the statements in our MySQL server as demonstrated by Figure 2-21:



Forward Engineering Progress

The following tasks will now be executed. Please monitor the execution.
Press Show Logs to see the execution logs.

- ✓ Connect to DBMS
- ✓ Execute Forward Engineered Script
- ✓ Read Back Changes Made by Server
- ✓ Save Synchronization State

Forward Engineer Finished Successfully

Figure 2-21. MySQL Workbench starts running the script

We cover the details of the DDL statements in "Creating Tables".

Chapter 3. Basic SQL

As mentioned in Chapter 2, Dr. Edgar F. Codd conceived the relational database model in 1969 and its normal forms in the early 1970s. In IBM laboratories in San Jose, a major research project started in the early 1970s called System/R, intending to prove the relational model's viability. Simultaneously, in 1974, Dr. Donald Chamberlin and his colleagues were also working to define a database language. They developed Structured English Query Language (SEQUEL), which allowed users to query a relational database using clearly defined English-style sentences, which was later renamed to *SQL* (Structured Query Language or SQL Query Language) for legal reasons.

The first database management systems based on SQL became available commercially by the end of the '70s. With the growing activity surrounding the development of database languages, standardization emerged to simplify things. And the community elected SQL for standardization. Both the American ANSI and the international ISO took part in the standardization, and in 1986 the first SQL standard was approved. After that, several versions existed. It is common to refer to the SQL standards as "SQL:1999", "SQL:2003", "SQL:2008", and "SQL:2011", and they refer to the versions of the standard released in the corresponding years, with the last being the most recent version. We will use the phrase *the SQL standard* or *standard SQL* to mean the current version of the SQL standard at any time.

MySQL extends the standard SQL providing extra features. For example, MySQL implements the *STRAIGHT_JOIN* which is a syntax not recognized by other DBMSes.

This chapter introduces MySQL's SQL implementation, which we often refer to as *CRUD* operations. CRUD refers to *CREATE*, *READ*, *UPDATE*, and *DELETE* operations. We will show you how to read data from a database with the SELECT statement and choose what data we can retrieve and in which order it is displayed. We also show you the basics of modifying your databases with the INSERT statement to add data, UPDATE to change, and DELETE to remove it. And we explain how to use the nonstandard SHOW TABLES and SHOW COLUMNS statements to explore your database.

Using the Sakila Database

In Chapter 2, we showed you the principles of how to build a database diagram using the ER model. We also introduced the steps you take to convert an ER model to a format that makes sense for constructing a relational database. This section will show you the structure of the MySQL sakila database for you to start getting familiar with different database relational models. We won't explain the SQL statements used to create the database here; that's the subject of Chapter 4.

To begin exploring the sakila database, if you haven't imported it yet, check "Entity Relationship Modeling Examples" to perform the task.

To choose the sakila database as our current database, we will use the USE statement.

Type the following command:

```
mysql> USE sakila;

Database changed
mysql>
```

We can check which is the active database by typing the SELECT DATABASE(); command:

```
mysql> SELECT DATABASE();

+------------+
| DATABASE() |
+------------+
| sakila     |
+------------+
1 row in set (0.00 sec)
```

Now, let’s explore what tables make up the `sakila` database using the `SHOW TABLES` statement:

```
mysql> SHOW TABLES;
```

```
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| address                    |
| category                   |
| city                       |
| country                    |
| customer                   |
| customer_list              |
| film                       |
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| staff_list                 |
| store                      |
+----------------------------+
23 rows in set (0.00 sec)
```

So far, there have been no surprises. Let’s find out more about each of the tables that make up the `sakila` database. First, let’s use the `SHOW COLUMNS` statement to explore the `actor` table:

```
mysql> SHOW COLUMNS FROM actor;

+-------------+-------------------+------+-----+-----------------
--+---
---------------------------------------------+
| Field       | Type              | Null | Key | Default
|
Extra                                          |
+-------------+-------------------+------+-----+-----------------
--+---
---------------------------------------------+
| actor_id    | smallint unsigned | NO   | PRI | NULL
|
auto_increment                                 |
| first_name  | varchar(45)       | NO   |     | NULL
|
|
| last_name   | varchar(45)       | NO   | MUL | NULL
|
|
| last_update | timestamp         | NO   |     |
CURRENT_TIMESTAMP |
DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
+-------------+-------------------+------+-----+-----------------
--+---
---------------------------------------------+
4 rows in set (0.01 sec)
```

The `DESCRIBE` keyword is identical to `SHOW COLUMNS FROM`, and we can abbreviate it to just `DESC`, so we can write the previous query as follows:

```
mysql> DESC actor;

+-------------+-------------------+------+-----+-----------------
--+---
---------------------------------------------+
| Field       | Type              | Null | Key | Default
|
Extra                                          |
+-------------+-------------------+------+-----+-----------------
--+---
---------------------------------------------+
| actor_id    | smallint unsigned | NO   | PRI | NULL
|
auto_increment                                  |
| first_name  | varchar(45)       | NO   |     | NULL
|
|
| last_name   | varchar(45)       | NO   | MUL | NULL
|
|
| last_update | timestamp         | NO   |     |
CURRENT_TIMESTAMP |
DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
+-------------+-------------------+------+-----+-----------------
--+---------------------------------------------+
4 rows in set (0.00 sec)
```

Let’s examine the table structure more closely. The `actor` table contains four columns, `actor_id`, `first_name`, `last_name`, and `last_update`. We can also extract the types of the columns — a *smallint* for `actor_id`, a *varchar(45)* for `first_name` and `last_name`, and a *timestamp* for `last_update`. None of the columns accept to be `NULL` (empty), `actor_id` is the primary key and `last_name` is the first column of a nonunique index. Don’t worry about the details; all that’s important right now are the column names we will use for the SQL commands.

Let’s explore the table `city`. We will execute the `DESC` statement:

```
mysql> DESC city;

+-------------+-------------------+------+-----+-----------------
--+---
--------------------------------------------+
| Field       | Type              | Null | Key | Default
|
Extra                                           |
+-------------+-------------------+------+-----+-----------------
--+---
--------------------------------------------+
| city_id     | smallint unsigned | NO   | PRI | NULL
|
auto_increment                                  |
| city        | varchar(50)       | NO   |     | NULL
|
|
| country_id  | smallint unsigned | NO   | MUL | NULL
|
|
| last_update | timestamp         | NO   |     |
CURRENT_TIMESTAMP |
DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
+------------+-------------------+------+-----+----------------
--+---
--------------------------------------------+
4 rows in set (0.01 sec)
```

NOTE

The `DEFAULT_GENERATED` that you see in the `Extra` column indicates that this particular column uses a default value. This information is a MySQL 8 notation particularity, and it is not present in MariaDB 10.5:

```
mysql> *DESC city;*

+-------------+----------------------+------+-----+------------
--------
-+--------------------------------+
| Field       | Type                 | Null | Key | Default
| Extra                        |
+-------------+----------------------+------+-----+------------
--------
-+--------------------------------+
| city_id     | smallint(5) unsigned | NO   | PRI | NULL
| auto_increment               |
| city        | varchar(50)          | NO   |     | NULL
|                              |
| country_id  | smallint(5) unsigned | NO   | MUL | NULL
|                              |
| last_update | timestamp            | NO   |     |
current_timestamp()
| on update current_timestamp() |
+-------------+----------------------+------+-----+------------
--------
-+--------------------------------+
4 rows in set (0.00 sec)
```

Again, what’s important is getting familiar with the columns in each table, as we’ll make frequent use of these later when we’re learning about querying.

The next section shows you how to explore the data that MySQL stores in the `sakila` database and its tables.

The SELECT Statement and Basic Querying Techniques

By now you’ve learned how to install and configure MySQL and how to use the MySQL command line. Now that you understand the ER model, you’re ready to start exploring its data and learning the SQL language that all MySQL clients use. This section introduces the most commonly used SQL keyword: the `SELECT` keyword. We explain the fundamental elements of style and syntax and the features of the `WHERE` clause, Boolean operators, and sorting (much of this also applies to our later discussions of `INSERT`, `UPDATE`, and `DELETE`). This isn’t the end of our discussion of `SELECT`; you’ll find more in Chapter 5, where we show you how to use its advanced features.

Single Table SELECTs

The most basic form of `SELECT` reads the data in all rows and columns from a table. Connect to MySQL using the command line and choose the `sakila` database:

```
mysql> use sakila;

Database changed
```

Let’s retrieve all of the data in the `language` table:

```
mysql> SELECT * FROM language;

+-------------+----------+---------------------+
| language_id | name     | last_update         |
+-------------+----------+---------------------+
|           1 | English  | 2006-02-15 05:02:19 |
|           2 | Italian  | 2006-02-15 05:02:19 |
|           3 | Japanese | 2006-02-15 05:02:19 |
|           4 | Mandarin | 2006-02-15 05:02:19 |
|           5 | French   | 2006-02-15 05:02:19 |
|           6 | German   | 2006-02-15 05:02:19 |
+-------------+----------+---------------------+
6 rows in set (0.00 sec)
```

The output has six rows, and each row contains the values for all the columns present in the table. We now know that there are six languages, and we can see the languages, identifiers, and the last time we updated these languages.

A simple `SELECT` statement has four components:

1. The keyword `SELECT`.
2. The columns to be displayed. In our first example, we asked for all columns using the asterisk (`*`) symbol as a wildcard character.
3. The keyword `FROM`.
4. The table name; in this example, the table name is `language`.

Using everything we saw, we've asked for all columns from the `language` table, and that's what MySQL has returned to us.

Let's try another simple `SELECT`. This time, we'll retrieve all columns from the `city` table:

```
mysql> SELECT * FROM city;

+---------+----------------------------+------------+------------
------
---+
| city_id | city                       | country_id | last_update
|
+---------+----------------------------+------------+------------
------
---+
|       1 | A Corua (La Corua)         |         87 | 2006-02-15
04:45:25 |
|       2 | Abha                       |         82 | 2006-02-15
04:45:25 |
|       3 | Abu Dhabi                  |         101 | 2006-02-15
04:45:25 |
...
|     599 | Zhoushan                   |          23 | 2006-02-15
04:45:25 |
|     600 | Ziguinchor                 |          83 | 2006-02-15
04:45:25 |
+---------+----------------------------+-----------+------------
------
---+
600 rows in set (0.00 sec)
```

There are 600 cities, and the output has the same basic structure as our first example.

The second example gives you an insight into how the relationships between the tables work. Consider the first row of the results. If you observe the column `country_id`, you will see the value 87. We will see in the next section, but based on this value, we can check on the `country` table that the country's name with code 87 is Spain. We'll discuss how to write queries on relationships between tables later in this chapter in "Joining Two Tables".

Notice also that we have several different cities with the same `country_id`. Having multiple `country_id` values isn't a problem since we expect a country with many cities (one-to-many relationship).

You should now feel comfortable choosing a database, listing its tables, and retrieving all of the data from a table using the `SELECT` statement. To practice, you might want to experiment with the other tables from `sakila` database. Remember that you can use the `SHOW TABLES` statement to find out the databases' table names.

Choosing Columns

You've so far used the `*` wildcard character to retrieve all columns in a table. If you don't want to display all the columns, it's easy to be more specific by listing the columns you want, in the order you want them, separated by commas. For example, if you want only the `city` name from the `city` table, you'd type:

```
mysql> SELECT city FROM city;

+--------------------+
| city               |
+--------------------+
| A Corua (La Corua) |
| Abha               |
| Abu Dhabi          |
| Acua               |
| Adana              |
+--------------------+
5 rows in set (0.00 sec)
```

If you want both the `city` name and the `city_id`, in that order, you'd use:

```
mysql> SELECT city, city_id FROM city;

+--------------------+---------+
| city               | city_id |
+--------------------+---------+
| A Corua (La Corua) |       1 |
| Abha               |       2 |
| Abu Dhabi          |       3 |
| Acua               |       4 |
| Adana              |       5 |
+--------------------+---------+
5 rows in set (0.01 sec)
```

You can even list columns more than once:

```
mysql> SELECT city, city FROM city;

+--------------------+--------------------+
| city               | city               |
+--------------------+--------------------+
| A Corua (La Corua) | A Corua (La Corua) |
| Abha               | Abha               |
| Abu Dhabi          | Abu Dhabi          |
| Acua                | Acua                |
| Adana               | Adana               |
+---------------------+---------------------+
5 rows in set (0.00 sec)
```

Even though this appears pointless, it can be useful when combined with aliases in more advanced queries, as we show in Chapter 5.

You can specify databases, tables, and column names in a `SELECT` statement. This allows you to avoid the `USE` command and work with any database and table directly with `SELECT`; it also helps resolve ambiguities, as we show later in "Joining Two Tables". Consider an example: suppose you want to retrieve the `name` column from the `language` table in the `sakila` database. You can do this with the following command:

```
mysql> SELECT name FROM sakila.language;
```

```
+----------+
| name     |
+----------+
| English  |
| Italian  |
| Japanese |
| Mandarin |
| French   |
| German   |
+----------+
6 rows in set (0.01 sec)
```

The `sakila.language` component after the `FROM` keyword specifies the `sakila` database and its `language` table. There's no need to enter `USE sakila;` before running this query. This syntax can also be used with other SQL statements, including the `UPDATE`, `DELETE`, `INSERT`, and `SHOW` statements we discuss later in this chapter.

Selecting Rows with the WHERE Clause

This section introduces the `WHERE` clause and explains how to use operators to write expressions. You'll see these in `SELECT` statements and other statements such as `UPDATE` and `DELETE`; we’ll show you examples later in this chapter.

WHERE basics

The `WHERE` clause is a powerful tool that allows you to filter which rows are returned from a `SELECT` statement. You use it to return rows that match a condition, such as having a column value that exactly matches a string, a number greater or less than a value, or a string that is a prefix of another. Almost all our examples in this and later chapters contain `WHERE` clauses, and you’ll become very familiar with them.

The simplest `WHERE` clause is one that exactly matches a value. Consider an example where we want to find out the English language’s details in the `language` table. Here’s what you type:

```
mysql> SELECT * FROM sakila.language WHERE name = 'English';

+-------------+---------+---------------------+
| language_id | name    | last_update         |
+-------------+---------+---------------------+
|           1 | English | 2006-02-15 05:02:19 |
+-------------+---------+---------------------+
1 row in set (0.00 sec)
```

MySQL returns all rows that match our search criteria — in this case, just the one row and all its columns.

Let’s try another *exact-match* example. Suppose you want to find out the actor’s first name with an `actor_id` value of 4. You type:

```
mysql> SELECT first_name FROM actor WHERE actor_id = 4;

+------------+
| first_name |
+------------+
| JENNIFER   |
+------------+
1 row in set (0.00 sec)
```

We’ve chosen a column and a row; we included the column `first_name` after the `SELECT` keyword and the `WHERE actor_id = 4`.

If a value matches more than one row, the results will contain all matches. Suppose we ask for all the cities belonging to Brazil, which has the `country_id` equal 15. You type in:

```
mysql> SELECT city FROM city WHERE country_id = 15;

+----------------------+
| city                 |
+----------------------+
| Alvorada             |
| Angra dos Reis       |
| Anpolis              |
| Aparecida de Goinia  |
| Araatuba             |
| Bag                  |
| Belm                 |
| Blumenau             |
| Boa Vista            |
| Braslia              |
| Goinia               |
| Guaruj               |
| guas Lindas de Gois  |
| Ibirit               |
| Juazeiro do Norte    |
| Juiz de Fora         |
| Luzinia              |
| Maring               |
| Po                   |
| Poos de Caldas       |
| Rio Claro            |
| Santa Brbara dOeste  |
| Santo Andr           |
| So Bernardo do Campo |
| So Leopoldo          |
| Sorocaba             |
| Vila Velha           |
| Vitria de Santo Anto |
+----------------------+
28 rows in set (0.00 sec)
```

The results show the names of the 28 cities that belong to Brazil. If we could join the information we get from the `city` table with information we get from the `country` table, we could display the cities' names with their respective country. We'll see how to perform this type of query in "Joining Two Tables".

Now let's retrieve values that belong to a range. Retrieve multiple values is simple for numeric ranges, so let's start by finding all cities' names with a `city_id` less than 5. To do this, execute the following statement:

```
mysql> SELECT city FROM city WHERE city_id < 5;

+---------------------+
| city                |
+---------------------+
| A Corua (La Corua)  |
| Abha                |
| Abu Dhabi           |
| Acua                |
+---------------------+
4 rows in set (0.00 sec)
```

For numbers, the frequently used operators are equals (=), greater than (>), less than (<), less than or equal (<=), greater than or equal (>=), and not equal (<> or `!=`).

Consider one more example. If you want to find all languages that don't have a `language_id` of 2, you'd type:

```
mysql> SELECT language_id, name FROM sakila.language
    -> WHERE language_id <>2;

+-------------+----------+
| language_id | name     |
+-------------+----------+
|           1 | English  |
|           3 | Japanese |
|           4 | Mandarin |
|           5 | French   |
|           6 | German   |
+-------------+---------+
5 rows in set (0.00 sec)
```

The previous output shows us the first, third, and all subsequent languages. Note that you can use either `<>` or `!=` operators for the *not-equal* condition.

You can use the same operators for strings. By default, string comparisons are not *case-sensitive* and use the current *character set*. For example:

```
mysql> SELECT first_name FROM actor WHERE first_name < 'B';

+------------+
| first_name |
+------------+
| ALEC       |
| AUDREY     |
| ANNE       |
| ANGELA     |
| ADAM       |
| ANGELINA   |
| ALBERT     |
| ADAM       |
| ANGELA     |
| ALBERT     |
| AL         |
| ALAN       |
| AUDREY     |
+------------+
13 rows in set (0.00 sec)
```

By *case-sensitive* we mean that `B` and `b` will be considered the same filter. So this query will provide the same result:

```
mysql> SELECT first_name FROM actor WHERE first_name < 'b';

+------------+
| first_name |
+------------+
| ALEC       |
| AUDREY     |
| ANNE       |
| ANGELA     |
| ADAM       |
| ANGELINA   |
| ALBERT     |
| ADAM       |
| ANGELA     |
| ALBERT     |
| AL         |
| ALAN       |
| AUDREY     |
+------------+
13 rows in set (0.00 sec)
```

Another prevalent task we’ll want to perform with strings is to find matches that begin with a prefix, contain a string, or end in a suffix. For example, we might want to find all album names beginning with the word “Retro.” We can do this with the `LIKE` operator in a `WHERE` clause. Let’s see an example where we are searching for a film with a title that contains the word `family`:

```
mysql> SELECT title FROM film WHERE title LIKE '%family%';

+----------------+
| title          |
+----------------+
| CYCLONE FAMILY |
| DOGMA FAMILY   |
| FAMILY SWEET   |
+----------------+
3 rows in set (0.00 sec)
```

Let’s discuss in detail how this works.

The `LIKE` clause is used with strings and means that a match must meet the pattern in the string that follows. In our example, we’ve used `LIKE "%family%"`, which means the string contains `family`, and it can be preceded or followed by zero or more characters. Most strings used with `LIKE` contain the percentage character (`%`) as a wildcard character that matches all possible strings. You can use it to define a string that ends in a suffix — such as ` “%ing"` — or a string that starts with a particular substring, such as `Corruption%`.

For example, `"John%"` would match all strings starting with `"John"`, such as `John Smith` and `John Paul Getty`. The pattern `"%Paul"` matches all strings that have `"Paul"` at the end. Finally, the pattern `"%Paul%"` matches all strings that have `"Paul"` in them, including at the start or at the end.

If you want to match exactly one wildcard character in a `LIKE` clause, you use the underscore character (_). For example, if you want all movie titles where the actor's name begins with a three-letter word that starts with □0□NAT□1□, you use:

```
mysql> SELECT title FROM film_list WHERE actors LIKE 'NAT_%';

+----------------------+
| title                |
+----------------------+
| FANTASY TROOPERS     |
| FOOL MOCKINGBIRD     |
| HOLES BRANNIGAN      |
| KWAI HOMEWARD        |
| LICENSE WEEKEND      |
| NETWORK PEAK         |
| NUTS TIES            |
| TWISTED PIRATES      |
| UNFORGIVEN ZOOLANDER |
+----------------------+
9 rows in set (0.04 sec)
```

NOTE

Avoid using the percentage (%) wildcard in the beginning of the pattern like in the following example:

```
mysql> SELECT title FROM film WHERE title LIKE '%family%';
```

You will get the results, but MySQL will not use the index under this condition. Using the percentage wildcard will force MySQL to read the entire table to retrieve the results and this can cause a severe performance impact if the table has millions of rows.

Combining conditions with AND, OR, NOT, and XOR

So far, we've used the WHERE clause to test one condition, returning all rows that meet it. You can combine two or more conditions using the Boolean operators AND, OR, NOT, and XOR.

Let's start with an example. Suppose you want to find the titles of sci-fi movies that are rated PG. This is straightforward with the AND operator:

```
mysql> SELECT title FROM film_list WHERE category LIKE 'Sci-Fi'
    -> AND rating LIKE 'PG';

+----------------------+
| title                |
+----------------------+
| CHAINSAW UPTOWN      |
| CHARADE DUFFEL       |
| FRISCO FORREST       |
| GOODFELLAS SALUTE    |
| GRAFFITI LOVE        |
| MOURNING PURPLE      |
| OPEN AFRICAN         |
| SILVERADO GOLDFINGER |
| TITANS JERK          |
| TROJAN TOMORROW      |
| UNFORGIVEN ZOOLANDER |
| WONDERLAND CHRISTMAS |
+----------------------+
12 rows in set (0.07 sec)
```

The AND operation in the WHERE clause restricts the results to those rows that meet both conditions.

The OR operator is used to find rows that meet at least one of several conditions. To illustrate, imagine now that you want a list of children or family movies. You can do this with two OR and three LIKE clauses:

```
mysql> SELECT title FROM film_list WHERE category LIKE 'Children'
    -> OR category LIKE 'Family';

+------------------------+
| title                  |
+------------------------+
| AFRICAN EGG            |
| APACHE DIVINE          |
| ATLANTIS CAUSE         |
...
| WRONG BEHAVIOR         |
| ZOOLANDER FICTION      |
+------------------------+
129 rows in set (0.04 sec)
```

The `OR` operations in the `WHERE` clause restrict the answers to those that meet any of the two conditions. As an aside, we can observe that the results are ordered. This is merely a coincidence; in this case, they're reported in the order they were added to the database. We'll return to sorting output later in "ORDER BY Clauses".

You can combine `AND` and `OR`, but you need to make it clear whether you want to first `AND` the conditions or `OR` them.

Parentheses cluster parts of a statement together and help make expressions readable; you can use them just as you would in basic math. Let's say that now you want sci-fi or family movies that are rated PG. We can write our query as follows:

```
mysql> SELECT title FROM film_list WHERE (category like 'Sci-Fi'
    -> OR category LIKE Family) AND rating LIKE 'PG';
```

```
+------------------------+
| title                  |
+------------------------+
| BEDAZZLED MARRIED      |
| CHAINSAW UPTOWN        |
| CHARADE DUFFEL         |
| CHASING FIGHT          |
| EFFECT GLADIATOR       |
...
| UNFORGIVEN ZOOLANDER   |
| WONDERLAND CHRISTMAS   |
+------------------------+
30 rows in set (0.07 sec)
```

The parentheses make the evaluation order clear: we want movies from Sci-Fi or Family category, but all of them need to be PG-rated.

With the use of parentheses, it is possible to change the evaluation order. The easiest way to check is by playing around with calculations:

```
mysql> SELECT (2+2)*3;

+---------+
| (2+2)*3 |
+---------+
|      12 |
+---------+
1 row in set (0.00 sec)

mysql> SELECT 2+2*3;

+-------+
| 2+2*3 |
+-------+
|     8 |
+-------+
1 row in set (0.00 sec)
```

TIP

One of the most difficult problems to find is a query that is running with no syntax errors, and it is returning values different from expected. While the `AND` operator does not influence, the `OR` operator has a significant influence. For example, this statement will give us the following result:

```
mysql> SELECT * FROM sakila.city WHERE city_id = 3
    -> OR city_id=4 AND country_id = 60;

+---------+-----------+------------+---------------------+
| city_id | city      | country_id | last_update         |
+---------+-----------+------------+---------------------+
|       3 | Abu Dhabi |        101 | 2006-02-15 04:45:25 |
|       4 | Acua      |         60 | 2006-02-15 04:45:25 |
+---------+-----------+------------+---------------------+
2 rows in set (0.00 sec)
```

If we change the ordering of the operators, we will obtain a different result:

```
mysql> SELECT * FROM sakila.city WHERE country_id = 60
    -> AND city_id = 3 OR city_id=4;

+---------+------+------------+---------------------+
| city_id | city | country_id | last_update         |
+---------+------+------------+---------------------+
|       4 | Acua |         60 | 2006-02-15 04:45:25 |
+---------+------+------------+---------------------+
1 row in set (0.00 sec)
```

Using parentheses makes the queries much easier to understand. We recommend that you use parentheses whenever there’s a chance MySQL could misinterpret the intention; there’s no good reason to rely on MySQL’s implicit evaluation order.

The unary `NOT` operator negates a Boolean statement. Suppose you want a list of all languages except the one having a `language_id` of 2. You’d write the query:

```
mysql> SELECT language_id, name FROM sakila.language
    -> WHERE NOT (language_id =2);

+-------------+----------+
| language_id | name     |
+-------------+----------+
|           1 | English  |
|            3 | Japanese |
|            4 | Mandarin |
|            5 | French   |
|            6 | German   |
+-------------+----------+
5 rows in set (0.01 sec)
```

The expression in the parentheses says we want:

```
(language_id = 2)
```

And the `NOT` operation negates it, so we get everything but those that meet the parentheses' condition. There are several other ways you can write a `WHERE` clause with the same idea. We will see later in Chapter 5 that some have a better performance than others.

Consider another example using `NOT` and parentheses. Suppose you want to get a list of all movie titles with an `FID` lesser than 7, but not those numbered 4 or 6:

```
mysql> SELECT fid,title from film_list where FID < 7 and not
(FID=4 OR FID=6);

+------+------------------+
| fid  | title            |
+------+------------------+
|    1 | ACADEMY DINOSAUR |
|    2 | ACE GOLDFINGER   |
|    3 | ADAPTATION HOLES |
|    5 | AFRICAN EGG      |
+------+------------------+
4 rows in set (0.06 sec)
```

Again, parentheses' expression lists movies that meet a condition — those that do not have `fid` equal to 4 or 6 and all movies with the `fid` lesser than 7.

The operator's precedence can be a little tricky, and sometimes it takes a lot of time for the DBA to debug a query and identify why the query is not returning the requested values. We show the operator precedences in the following list, from highest priority to the lowest. Operators that are shown together on a line have the same priority.

INTERVAL
BINARY, COLLATE
!
- (unary minus), ~ (unary bit inversion)
^
*, /, DIV, %, MOD
-,+
<<, >>
&
|
= (comparison), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN, MEMBER OF
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
AND, &&
XOR
OR, ||
= (assignment), :=

It is possible to combine these operators in the most diverse ways to get the desired results. For example, suppose we want the movie titles that have a price range between 2 USD and 4 USD and belong to the Documentary or Horror category, and one of the actors has the name Bob:

```
mysql> SELECT title
    -> FROM film_list
    -> WHERE price BETWEEN 2 AND 4
    -> AND (category LIKE 'Documentary' OR category LIKE
'Horror')
    -> AND actors LIKE '%BOB%';

+-------------------+
| title             |
+-------------------+
| ADAPTATION HOLES |
+-------------------+
1 row in set (0.08 sec)
```

Finally, before we move to *sorting*, it is possible to execute queries that do not attend the requisites, and in this case they will return empty:

```
mysql> SELECT title FROM film_list
    -> WHERE price BETWEEN 2 AND 4
    -> AND (category LIKE 'Documentary' OR category LIKE Horror)
    -> AND actors LIKE '%GRIPPA%';

Empty set (0.04 sec)
```

ORDER BY Clauses

We've so far discussed how to choose the columns, and which rows are returned as part of the query result, but not how to control how the result is displayed. In a relational database, the rows in a table form a set; there is no intrinsic order between the rows, and so we have to ask MySQL to sort the results if we want them in a particular order. This section explains how to use the `ORDER BY` clause to do this. Sorting does not affect *what* is returned; it only affects *what order* the results are returned.

NOTE

MySQL, in particular, InnoDB table, has a special index called *clustered index* that stores the data. When you define a PRIMARY KEY on a table, InnoDB uses it as the clustered index. Suppose you are executing queries based on the PRIMARY KEY. In that case, the rows will return ordered in ascending order by the PRIMARY KEY. However, we always recommending using the ORDER BY if you want to enforce a particular order.

Suppose you want to return a list of the first ten customers in the sakila database, sorted alphabetically by name. Here's what you'd type:

```
mysql> SELECT name FROM customer_list
    -> ORDER BY name
    -> LIMIT 10;
```

```
+--------------------+
| name               |
+--------------------+
| AARON SELBY        |
| ADAM GOOCH         |
| ADRIAN CLARY       |
| AGNES BISHOP       |
| ALAN KAHN          |
| ALBERT CROUSE      |
| ALBERTO HENNING    |
| ALEX GRESHAM       |
| ALEXANDER FENNELL  |
| ALFRED CASILLAS    |
+--------------------+
10 rows in set (0.01 sec)
```

The ORDER BY clause indicates that sorting is required, followed by the column that should be used as the sort key. In this example, we're sorting by alphabetically-ascending name. The default sort is case-insensitive and in ascending order, and MySQL automatically sorts alphabetically because the columns are character strings. The way strings are sorted is determined by the character set and collation order that are being used. We discuss these in "Collation and Character Sets". For most of this book, we assume that you're using the default settings.

Let’s see a second example. This time, let’s sort the output from the `address` table by ascending the `last_update` column:

```
mysql> SELECT address, last_update FROM address
    -> *ORDER BY last_update LIMIT 5;

+-----------------------------+---------------------+
| address                     | last_update         |
+-----------------------------+---------------------+
| 1168 Najafabad Parkway      | 2014-09-25 22:29:59 |
| 1031 Daugavpils Parkway     | 2014-09-25 22:29:59 |
| 1924 Shimonoseki Drive      | 2014-09-25 22:29:59 |
| 757 Rustenburg Avenue       | 2014-09-25 22:30:01 |
| 1892 Nabereznyje Telny Lane | 2014-09-25 22:30:02 |
+-----------------------------+---------------------+
5 rows in set (0.00 sec)
```

As we can see, it is possible to sort different types of columns. Moreover, we can compound the sorting with two or more columns. For example, let’s say you want to sort alphabetically the addresses but for each district.

```
mysql> SELECT address, district FROM address
    -> ORDER BY district, address;

+----------------------------------------+----------------------+
| address                                | district             |
+----------------------------------------+----------------------+
| 1368 Maracabo Boulevard                |                      |
| 18 Duisburg Boulevard                  |                      |
| 962 Tama Loop                          |                      |
| 535 Ahmadnagar Manor                   | Abu Dhabi            |
| 669 Firozabad Loop                     | Abu Dhabi            |
| 1078 Stara Zagora Drive                | Aceh                 |
| 663 Baha Blanca Parkway                | Adana                |
| 842 Salzburg Lane                      | Adana                |
| 614 Pak Kret Street                    | Addis Abeba          |
| 751 Lima Loop                          | Aden                 |
| 1157 Nyeri Loop                        | Adygea               |
| 387 Mwene-Ditu Drive                   | Ahal                 |
| 775 ostka Drive                        | al-Daqahliya         |
...
| 1416 San Juan Bautista Tuxtepec Avenue | Zufar                |
| 138 Caracas Boulevard                  | Zulia                |
+------------------------------------------+----------------------+
603 rows in set (0.00 sec)
```

You can also sort in descending order, and you can control this behavior for each sort key. Suppose you want to sort the address by descending alphabetical order and the districts in descending order. You type this:

```
mysql> SELECT address,district FROM address
    -> ORDER BY district ASC, address DESC
    -> LIMIT 10;

+-------------------------+-------------+
| address                 | district    |
+-------------------------+-------------+
| 962 Tama Loop           |             |
| 18 Duisburg Boulevard   |             |
| 1368 Maracabo Boulevard |             |
| 669 Firozabad Loop      | Abu Dhabi   |
| 535 Ahmadnagar Manor    | Abu Dhabi   |
| 1078 Stara Zagora Drive | Aceh        |
| 842 Salzburg Lane       | Adana       |
| 663 Baha Blanca Parkway | Adana       |
| 614 Pak Kret Street     | Addis Abeba |
| 751 Lima Loop           | Aden        |
+-------------------------+-------------+
10 rows in set (0.01 sec)
```

If a collision of values occurs and you don’t specify another sort key, the sort order is undefined. This may not be important for you; you may not care about the order in which two customers with the identical name “John A. Smith” appear.

The LIMIT Clause

As you may have noted, a few queries previously mentioned use the `LIMIT` clause. The `LIMIT` clause is a useful, nonstandard SQL statement that allows you to control which rows are output. Its basic form allows you to limit the number of rows returned from a `SELECT` statement, which is useful when you want to limit the amount of data communicated over a network or output to the screen. You might use it, for example, to get a sample of the data from the table as we have been doing. Here’s an example:

```
mysql> SELECT name FROM customer_list LIMIT 10;

+-------------------+
| name              |
+-------------------+
| VERA MCCOY        |
| MARIO CHEATHAM    |
| JUDY GRAY         |
| JUNE CARROLL      |
| ANTHONY SCHWAB    |
| CLAUDE HERZOG     |
| MARTIN BALES      |
| BOBBY BOUDREAU    |
| WILLIE MARKHAM    |
| JORDAN ARCHULETA |
+-------------------+
```

The `LIMIT` clause can have two arguments. With two arguments, the first argument specifies the first row’s offset to return, and the second specifies the maximum number of rows to return. The first argument is known as *offset*. Suppose you want five rows, but you want to remove the first five rows, which means the result will start at the sixth row. You do this by starting from after the fifth answer:

```
mysql> SELECT name FROM customer_list LIMIT 5, 5;

+-------------------+
| name              |
+-------------------+
| CLAUDE HERZOG     |
| MARTIN BALES      |
| BOBBY BOUDREAU    |
| WILLIE MARKHAM    |
| JORDAN ARCHULETA |
+-------------------+
5 rows in set (0.00 sec)
```

The output is rows 6 to 10 from the `SELECT` query.

There's an alternative syntax that you might see for the `LIMIT` keyword: instead of writing `LIMIT 10,5`, you can write `LIMIT 10 OFFSET 5`. The `OFFSET` syntax discards the N values specified in it.

This is an example with no `offset`:

```
mysql> SELECT id, name FROM customer_list
    -> ORDER BY id LIMIT 10;

+----+------------------+
| ID | name             |
+----+------------------+
|  1 | MARY SMITH       |
|  2 | PATRICIA JOHNSON |
|  3 | LINDA WILLIAMS   |
|  4 | BARBARA JONES    |
|  5 | ELIZABETH BROWN  |
|  6 | JENNIFER DAVIS   |
|  7 | MARIA MILLER     |
|  8 | SUSAN WILSON     |
|  9 | MARGARET MOORE   |
| 10 | DOROTHY TAYLOR   |
+----+------------------+
10 rows in set (0.00 sec)
```

Now using an `offset` of 5, we will discard the first five ids:

```
mysql> SELECT id, name FROM customer_list
    -> ORDER BY id LIMIT 10 OFFSET 5;

+----+----------------+
| ID | name           |
+----+----------------+
|  6 | JENNIFER DAVIS |
|  7 | MARIA MILLER   |
|  8 | SUSAN WILSON   |
|  9 | MARGARET MOORE |
| 10 | DOROTHY TAYLOR |
| 11 | LISA ANDERSON  |
| 12 | NANCY THOMAS   |
| 13 | KAREN JACKSON  |
| 14 | BETTY WHITE    |
| 15 | HELEN HARRIS   |
+----+----------------+
10 rows in set (0.01 sec)
```

Joining Two Tables

We've so far worked with just one table in our `SELECT` queries. However, in the ER model, we saw that a relational database is all about working with the relationships between tables to answer information needs. Indeed, as we've explored the tables in the `sakila` database, it's become obvious that by using these relationships, we can answer more interesting queries. For example, it'd be useful to know the countries of each city. This section shows you how to answer these queries by joining two tables. We'll return to this issue as part of a longer, more advanced discussion of joins in Chapter 5.

We use only one join syntax in this chapter. There are several more, and each gives you a different way to bring together data from two or more tables. The syntax we use here is the `INNER JOIN`, which is the most used in daily activities. Consider an example, and then we'll explain more about how it works:

```
mysql> SELECT city, country FROM city INNER JOIN country
    -> ON city.country_id = country.country_id
    -> WHERE country.country_id < 5
    -> ORDER BY country, city;

+----------+----------------+
| city     | country        |
+----------+----------------+
| Kabul    | Afghanistan    |
| Batna    | Algeria        |
| Bchar    | Algeria        |
| Skikda   | Algeria        |
| Tafuna   | American Samoa |
| Benguela | Angola         |
| Namibe   | Angola         |
+----------+----------------+
7 rows in set (0.00 sec)
```

The output shows the cities and their country. You can see for the first time which city belongs to which country.

How does the `INNER JOIN` work? The statement has two parts: first, two table names separated by the `INNER JOIN` keywords; second, the `ON` keyword that indicates which column (or columns) holds the relationship between the two tables. In our first example, the two tables to be joined are `city` and `country`, expressed as `city INNER JOIN country` (for the basic `INNER JOIN`, it doesn't matter what order you list the tables in, and so using `country INNER JOIN city` would have the same effect). In the example, the `ON` clause is where we tell MySQL the columns that hold the relationship between the tables; you should recall this from our design and our previous discussion in Chapter 2.

If the join condition uses the equal operator (=) and the column names in both tables used for matching are the same, you can use the USING clause instead:

```
mysql> SELECT city, country FROM city
    -> INNER JOIN country using (country_id)
    -> WHERE country.country_id < 5
    -> ORDER BY country, city;

+----------+----------------+
| city     | country        |
+----------+----------------+
| Kabul    | Afghanistan    |
| Batna    | Algeria        |
| Bchar    | Algeria        |
| Skikda   | Algeria        |
| Tafuna   | American Samoa |
| Benguela | Angola         |
| Namibe   | Angola         |
+----------+----------------+
7 rows in set (0.01 sec)
```

The following Venn diagram in Figure 3-1 illustrates the inner join:

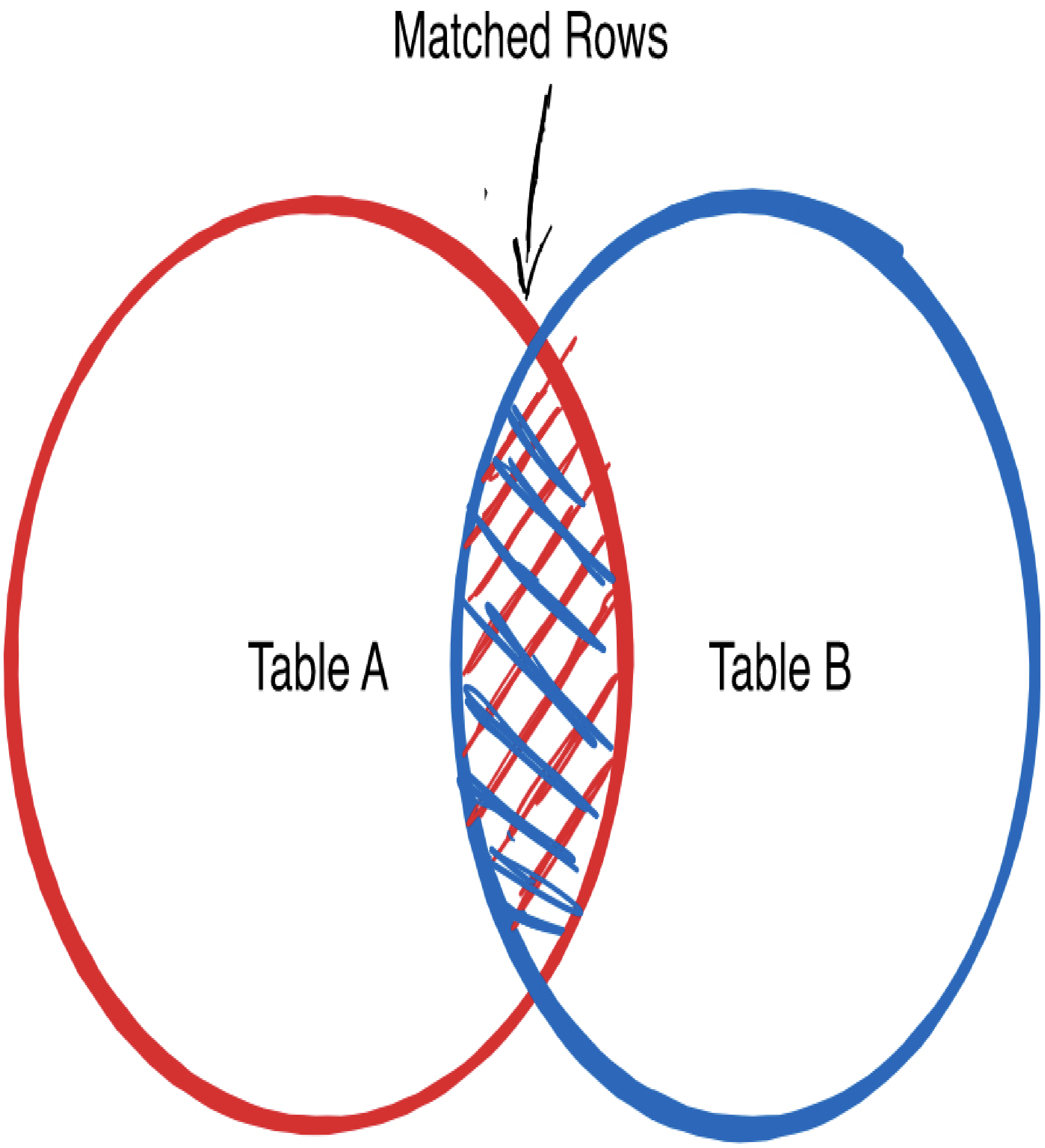


Figure 3-1. The Venn diagram representation of the INNER JOIN

As we saw in the previous example, all operators are supported when using `INNER JOIN`. For example, we used the `WHERE` condition and the `LIMIT` clause.

Before we leave `SELECT`, we'll give you a taste of one of the functions you can use to aggregate values. Suppose you want to count how many cities

Italy has in our database. You can do this by counting the number of rows using the `COUNT()` function. Here's how it works:

```
mysql> SELECT count(1) FROM city INNER JOIN country
    -> ON city.country_id = country.country_id
    -> WHERE country.country_id = 49
    -> ORDER BY country, city;

+----------+
| count(1) |
+----------+
|        7 |
+----------+
1 row in set (0.00 sec)
```

We explain more features of `SELECT` and aggregate functions in Chapter 5.

The INSERT Statement

The `INSERT` statement is used to add new data to tables. This section explains its basic syntax and shows you simple examples that add new rows to the `sakila` database. In Chapter 4, we'll discuss how to load data from existing tables or external data sources.

INSERT Basics

Inserting data typically occurs in two situations: when you bulk-load in a large batch as you create your database, and when you add data on an ad hoc basis as you use the database. In MySQL, different optimizations are built into the server for each situation. Importantly, different SQL syntaxes are available to make it easy for you to work with the server in both cases. We explain a basic `INSERT` syntax in this section and show you examples of using it for bulk and single record insertion.

Let's start with the basic task of inserting one new row into the `language` table. To do this, you need to understand the table's structure. As we explained in Chapter 2, you can discover this with the `SHOW COLUMNS` statement:

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

```
+-------------+------------------+------+-----+------------------
-+---------------------------------------------+
| Field       | Type             | Null | Key | Default
| Extra                                        |
+-------------+------------------+------+-----+------------------
-+---------------------------------------------+
| language_id | tinyint unsigned | NO   | PRI | NULL
| auto_increment                               |
| name        | char(20)         | NO   |     | NULL
|                                              |
| last_update | timestamp        | NO   |     | CURRENT_TIMESTAMP
| DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
+-------------+------------------+------+-----+------------------
-+---------------------------------------------+
3 rows in set (0.00 sec)
```

This tells you that the *language_id* column is auto-generated, and the *last_update* column is updated every time an UPDATE operation happens. You'll learn more about the `AUTO_INCREMENT` shortcut to automatically assign the next available identifier in Chapter 4.

Our new row is for a new language, "Portuguese". How can we insert the row? There are two ways to do it. The most common is to let MySQL fill the default values for the `language_id`. It will be like this:

```
mysql> INSERT INTO language VALUES (NULL, 'Portuguese', NOW());
```

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

And if we execute a SELECT on the table:

```
mysql> SELECT * FROM language;
```

```
+-------------+------------+---------------------+
| language_id | name       | last_update         |
+-------------+------------+---------------------+
|           1 | English    | 2006-02-15 05:02:19 |
|           2 | Italian    | 2006-02-15 05:02:19 |
|           3 | Japanese   | 2006-02-15 05:02:19 |
|           4 | Mandarin   | 2006-02-15 05:02:19 |
|           5 | French     | 2006-02-15 05:02:19 |
|           6 | German     | 2006-02-15 05:02:19 |
|           7 | Portuguese | 2020-09-26 09:11:36 |
+-------------+------------+---------------------+
7 rows in set (0.00 sec)
```

We can see MySQL inserted the row. Note that we used the function `NOW()` in the `last_update` column. The `NOW()` function returns the current date and time of the MySQL server.

The second option is inserting the value of the `language_id` manually. Now that we already have seven languages, we should use 8 for the next value of the `language_id`. We can check with this SQL instruction:

```
mysql> SELECT MAX(language_id) FROM language;
```

```
+------------------+
| max(language_id) |
+------------------+
|                7 |
+------------------+
1 row in set (0.00 sec)
```

The `MAX()` function tells you the maximum value for the column supplied as a parameter. This is cleaner than `SELECT artist_id FROM artist`, which prints out all rows and requires you to inspect the rows to find the maximum value; adding an `ORDER BY` makes it easier. Using `MAX()` is also much more straightforward than `SELECT language_id FROM language ORDER BY language_id DESC LIMIT 1`, which again returns the correct answer.

We’re now ready to insert the row. In this INSERT, we are going to insert the `last_update` manually too. Here’s the needed command:

```
mysql> INSERT INTO language VALUES (8, 'Russian', '2020-09-26
10:35:00');
```

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

A new row is created — MySQL reports that one row has been affected — and the value. Let's check again:

```
mysql> SELECT * FROM language;
```

```
+-------------+------------+---------------------+
| language_id | name       | last_update         |
+-------------+------------+---------------------+
|           1 | English    | 2006-02-15 05:02:19 |
|           2 | Italian    | 2006-02-15 05:02:19 |
|           3 | Japanese   | 2006-02-15 05:02:19 |
|           4 | Mandarin   | 2006-02-15 05:02:19 |
|           5 | French     | 2006-02-15 05:02:19 |
|           6 | German     | 2006-02-15 05:02:19 |
|           7 | Portuguese | 2020-09-26 09:11:36 |
|           8 | Russian    | 2020-09-26 10:35:00 |
+-------------+------------+---------------------+
8 rows in set (0.00 sec)
```

It is also possible to insert multiple values at once:

```
mysql> INSERT INTO language VALUES (NULL, 'Spanish', NOW()),
    -> (NULL, 'Hebrew', NOW());
```

```
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

The single row `INSERT` style detects primary key duplicates: if it finds a duplicate, it'll stop as soon as it finds a duplicate key. For example, suppose we try to insert the same language again:

```
mysql> INSERT INTO language VALUES (8, 'Arabic', '2020-09-26
10:35:00');
```

```
ERROR 1062 (23000): Duplicate entry '8' for key
'language.PRIMARY'
```

The `INSERT` operation stops on the first duplicate key. You can add an `IGNORE` clause to prevent the error if you want, but note that the row is not going to be inserted:

```
mysql> INSERT IGNORE INTO language VALUES (8, 'Arabic', '2020-09-
26 10:35:00');

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

However, in most cases, you want to know about possible problems (after all, primary keys are supposed to be unique), so this `IGNORE` syntax is rarely used.

You’ll notice that MySQL reports the results of bulk insertion differently from single insertion. From our initial bulk insertion, it reports:

```
mysql> INSERT INTO language VALUES (null, 'Spanish', NOW()),
    -> *(NULL, 'Hebrew', NOW());

Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

The first line tells you how many rows were inserted, while the first entry in the final line tells you how many rows (or records) were actually processed. If you use `INSERT IGNORE` and try to insert a duplicate record — for which the primary key matches that of an existing row — then MySQL will quietly skip inserting it and report it as a duplicate in the second entry on the final line:

```
Query OK, 0 rows affected, 2 warnings (0.00 sec)
Records: 2  Duplicates: 2  Warnings: 2
```

We discuss causes of warnings — shown as the third entry on the final line — in Chapter 4.

Alternative Syntaxes

There are several alternatives to the `VALUES` syntax we've shown you so far. This section shows you these and explains the advantages and drawbacks of each. If you're happy with the basic syntax we've described so far, and want to move on to a new topic, feel free to skip ahead to "The DELETE Statement".

There are three disadvantages of the `VALUES` syntax we've shown you: You need to remember the order of the columns. You need to provide a value for each column. It's closely tied to the underlying table structure: if you change the table's structure, you need to change the `INSERT` statements, and the function of the `INSERT` statement isn't obvious unless you have the table structure at hand. However, the three advantages of the approach are that it works for both single and bulk inserts, you get an error message if you forget to supply values for all columns, and you don't have to type in column names. Fortunately, we can avoid the disadvantages by varying the syntax.

Suppose you know that the `actor` table has four columns, and you recall their names, but you forget their order. You can insert using the following approach:

```
mysql> INSERT INTO actor (actor_id, first_name, last_name,
last_update)
    -> VALUES (NULL, 'Vinicius', 'Grippa', NOW());

Query OK, 1 row affected (0.03 sec)
```

The column names are included in parentheses after the table name, and the values stored in those columns are listed in parentheses after the `VALUES` keyword. So, in this example, a new row is created and the value 201 is stored as the `actor_id` (remember `actor_id` has the `auto_increment` property), `Vinicius` is stored as the `first_name`, `Grippa` is stored as the `last_name`, and the `last_update` is updated with the current timestamp. This syntax's advantages are that it's readable and flexible (addressing the third disadvantage we described) and order-independent (addressing the first disadvantage). The burden is that you need to know the column names and type them in.

This new syntax can also address the second disadvantage of the simpler approach — that is, it can allow you to insert values for only some columns. To understand how this might be useful, let's explore the `city` table:

```
mysql> DESC city;

+-------------+----------------------+------+-----+--------------
-----+-----------------------------+
| Field       | Type                 | Null | Key | Default
| Extra                       |
+-------------+----------------------+------+-----+--------------
-----+-----------------------------+
| city_id     | smallint(5) unsigned | NO   | PRI | NULL
| auto_increment              |
| city        | varchar(50)          | NO   |     | NULL
|                             |
| country_id  | smallint(5) unsigned | NO   | MUL | NULL
|                             |
| last_update | timestamp            | NO   |     |
CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
+-------------+----------------------+------+-----+--------------
-----+-----------------------------+
4 rows in set (0.00 sec)
```

Notice that the `last_update` column has a default value of `CURRENT_TIMESTAMP`. This means that if you don't insert a value for the `last_update` column, it'll insert the current date and time by default. This is just what we want: when we record a store, we don't want to bother checking the date and time and typing it in. Here's how you insert an incomplete played entry:

```
mysql> INSERT INTO city (city, country_id) VALUES ('Bebedouro',
19);

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

We didn't set the `city_id` column, so MySQL defaults it to the next available value (`auto_increment` property), and `last_update` stores the current date and time. You can check this with a query:

```
mysql> SELECT * FROM city where city like 'Bebedouro';

+---------+-----------+------------+---------------------+
| city_id | city      | country_id | last_update         |
+---------+-----------+------------+---------------------+
|     601 | Bebedouro |         19 | 2021-02-27 21:34:08 |
+---------+-----------+------------+---------------------+
1 row in set (0.01 sec)
```

You can also use this approach for bulk insertion as follows:

```
mysql> INSERT INTO city (city,country_id) VALUES
    -> ('Sao Carlos',19),
    -> ('Araraquara',19),
    -> ('Ribeirao Preto',19);

Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

This approach's disadvantages are that you can accidentally omit values for columns, and you need to remember and type column names. MySQL will set the omitted columns to the default values.

All columns in a MySQL table have a default value of `NULL` unless another default value is explicitly assigned when the table is created or modified.

You can set defaults and still use the original `INSERT` syntax with MySQL 5.7 or later by using the `DEFAULT` keyword. Here's an example that adds a `country` row using `DEFAULT`:

```
mysql> INSERT INTO country VALUES (NULL, 'Uruguay', DEFAULT);

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

The keyword DEFAULT tells MySQL to use the default value for that column, and so the current date and time are inserted in our example. This approach's advantages are that you can use the bulk-insert feature with default values, and you can never accidentally omit a column.

There's another alternative INSERT syntax. In this approach, you list the column name and value together, giving the advantage that you don't have to mentally map the list of values to the earlier list of columns. Here's an example that adds a new row to the country table:

```
mysql> INSERT INTO country SET country_id=NULL,
    -> country='Bahamas', last_update=NOW();

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

The syntax requires you to list a table name, the keyword SET, and then column-equals-value pairs, separated by commas. Columns that aren't supplied are set to their default values. Again, the disadvantages are that you can accidentally omit values for columns and that you need to remember and type in column names. A significant additional disadvantage is that you can't use this method for bulk insertion.

You can also insert using values returned from a query. We discuss this in Chapter 7.

The DELETE Statement

The DELETE statement is used to remove one or more rows from a table. We explain single-table deletes here, and discuss multi-table deletes — which remove data from two or more tables through one statement — in Chapter 7.

DELETE Basics

The simplest use of DELETE is to remove all rows in a table. Suppose you want to empty your rental table. You do this with:

```
mysql> DELETE FROM rental;

Query OK, 16044 rows affected (2.41 sec)
```

The DELETE syntax doesn’t include column names since it’s used to remove whole rows and not just values from a row. To reset or modify a value in a row, you use the UPDATE statement, described later in this chapter in “The UPDATE Statement”. The DELETE statement doesn’t remove the table itself. For example, having deleted all rows in the rental table, you can still query the table:

```
mysql> SELECT * FROM rental;

Empty set (0.00 sec)
```

Of course, you can also continue to explore its structure using DESCRIBE or SHOW CREATE TABLE, and insert new rows using INSERT. To remove a table, you use the DROP statement described in Chapter 4.

Note that if our table has a relationship with another table, the delete will fail because of the foreign key constraint:

```
mysql> DELETE FROM language;

ERROR 1451 (23000): Cannot delete or update a parent row: a
foreign key constraint fails (`sakila`.`film`, CONSTRAINT
`fk_film_language` FOREIGN KEY (`language_id`) REFERENCES
`language` (`language_id`) ON UPDATE CASCADE)
```

Using WHERE, ORDER BY, and LIMIT

If you’ve deleted rows in the previous section, reload your sakila database now. You need the rows in the rental table restored for the examples in this section.

To remove one or more rows, but not all rows in a table, use a WHERE clause. This works in the same way as it does for SELECT. For example, suppose you want to remove all rows from the `rental` table with `rentail_id` less than 10. You do this with:

```
mysql> DELETE FROM rental WHERE rental_id < 10;

Query OK, 9 rows affected (0.01 sec)
```

The result is that the nine rental rows that match the criteria are removed.

Suppose we want to remove all the payments from a customer called Mary Smith. First, let’s perform a select with the `customer` and `payment` tables using INNER JOIN (“Joining Two Tables”):

```
mysql> SELECT first_name, last_name, customer.customer_id,
    -> amount, payment_date FROM payment INNER JOIN customer
    -> ON customer.customer_id=payment.customer_id
    -> WHERE first_name like 'Mary'
    -> AND last_name like 'Smith';

+------------+-----------+-------------+--------+----------------
-----+
| first_name | last_name | customer_id | amount | payment_date
|
+------------+-----------+-------------+--------+----------------
-----+
| MARY       | SMITH     |           1 |   2.99 | 2005-05-25
11:30:37 |
| MARY       | SMITH     |           1 |   0.99 | 2005-05-28
10:35:23 |
| MARY       | SMITH     |           1 |   5.99 | 2005-06-15
00:54:12 |
| MARY       | SMITH     |           1 |   0.99 | 2005-06-15
18:02:53 |
...
| MARY       | SMITH     |           1 |   1.99 | 2005-08-22
01:27:57 |
| MARY       | SMITH     |           1 |   2.99 | 2005-08-22
19:41:37 |
| MARY       | SMITH     |           1 |   5.99 | 2005-08-22
20:03:46 |
+------------+-----------+-------------+--------+----------------
-----+
32 rows in set (0.00 sec)
```

Next, to remove the rows from Mary Smith which has the `customer_id` equal one, the following delete is performed:

```
mysql> DELETE FROM artist WHERE artist_id = 3;

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

Then do the same thing for the `album`, `track`, and `played` tables:

```
mysql> DELETE FROM payment where customer_id=1;

Query OK, 32 rows affected (0.01 sec)
```

You can use the `ORDER BY` and `LIMIT` clauses with `DELETE`. You usually do this when you want to limit the number of rows deleted.

```
mysql> DELETE FROM payment ORDER BY customer_id LIMIT 10000;

Query OK, 10000 rows affected (0.22 sec)
```

TIP

We highly recommend using `delete` and `update` operations for a small set of rows due to performance issues. This value usually varies depending on the hardware, but a good value is around 20000-40000 rows per batch.

Removing All Rows with TRUNCATE

If you want to remove all rows in a table, there's a faster method than removing them with `DELETE`. Using the `TRUNCATE TABLE` statement, MySQL takes the shortcut of dropping the table, removing the table structures, and then re-creating them. When there are many rows in a table, this is much faster.

NOTE

As a curiosity, there is a bug in MySQL 5.6 version that can stall MySQL when performing `TRUNCATE` operation when MySQL is configured with a large InnoDB buffer pool (200Gb or more). The bug is reported here:

Concurrent `TRUNCATE` operations cause stalls

If you want to remove the data in the `payment` table, you can execute this:

```
mysql> TRUNCATE TABLE payment;

Query OK, 0 rows affected (0.07 sec)
```

Notice that the number of rows affected is shown as zero: to quickly delete all the data in the table, MySQL doesn't count the number of rows that are deleted, so the number shown does not reflect the actual number of rows deleted.

The `TRUNCATE TABLE` statement differs from `DELETE` in a lot of ways, but it is worth mentioning a few:

- Truncate operations drop and re-create the table, which is much faster than deleting rows one by one, particularly for large tables.
- Truncate operations cause an implicit commit, so we cannot rollback.
- We cannot perform truncation operations if the session holds an active table lock.

Table types, transactions, and locking are discussed in Chapter 5. None of these limitations affect most applications in practice, and you can use `TRUNCATE TABLE` to speed up your processing. Of course, it's not common to delete whole tables during regular operation. An exception is temporary tables used to store query results for a particular user session temporarily and can be deleted without losing the original data.

The UPDATE Statement

The `UPDATE` statement is used to change data. In this section, we show you how to update one or more rows in a single table. Multitable updates are discussed in "Updates".

If you've deleted rows from your `sakila` database, reload it by following the instructions in "Entity Relationship Modeling Examples".

Examples

The simplest use of the `UPDATE` statement is to change all rows in a table. Imagine the situation where we need to update the `amount` column of the `payment` table by 10% for all payments. To change all rows, you need to execute:

```
mysql> UPDATE payment SET amount=amount*1.1;

Query OK, 16025 rows affected, 16025 warnings (0.41 sec)
Rows matched: 16049  Changed: 16025  Warnings: 16025
```

Note that we forgot to update the `last_update` status. To make it coherent with the database model we expect, you can fix running the following statement:

```
mysql> UPDATE payment SET last_update='2021-02-28 17:53:00';

Query OK, 16049 rows affected (0.27 sec)
Rows matched: 16049  Changed: 16049  Warnings: 0
```

TIP

You can use the `NOW()` function to update the `last_update` column with the current timestamp of the execution. For example:

```
mysql> UPDATE payment SET last_update=NOW();
```

The second row reported by an `UPDATE` statement shows the overall effect of the statement. In our example, you see:

```
Rows matched: 16049  Changed: 16049  Warnings: 0
```

The first column reports the number of rows that were retrieved as answers by the statement; in this case, since there's no `WHERE` or `LIMIT` clause, all rows in the table match the query. The second column reports how many rows needed to be changed, and this is always equal to or less than the number of rows that match. If you repeat the statement, you'll see a different result:

```
mysql> UPDATE payment SET last_update='2021-02-28 17:53:00';

Query OK, 0 rows affected (0.07 sec)
Rows matched: 16049  Changed: 0  Warnings: 0
```

This time, since the date is already set to `2021-02-28 17:53:00` and there is no `WHERE` condition, all the rows still match the statement but none are changed. Note also the number of rows changed is always equal to the number of rows affected, as reported on the first line of the output.

Using WHERE, ORDER BY, and LIMIT

Often, you don't want to change all rows in a table. Instead, you want to update one or more rows that match a condition. As with `SELECT` and `DELETE`, the `WHERE` clause is used for the task. In addition, in the same way as with `DELETE`, you can use `ORDER BY` and `LIMIT` together to control how many rows are updated from an ordered list.

Let's try an example that modifies one row in a table. Suppose that in the `actor` table, the actress Penelope Guiness changed her last name. To update it in the database, you need to execute:

```
mysql> UPDATE actor SET last_name= UPPER('cruz')
    -> WHERE first_name LIKE 'PENELOPE'
    -> AND last_name like 'GUINESS';
```

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

As expected, MySQL matched one row, and MySQL changed one row.

To control how many updates occur, you can use the combination of `ORDER BY` and `LIMIT`. As with `DELETE`, you would do this because you either want to perform the operation in small chunks or modify only some rows.

```
mysql> UPDATE payment SET last_update=NOW() LIMIT 10;

Query OK, 10 rows affected (0.01 sec)
Rows matched: 10  Changed: 10  Warnings: 0
```

You can see that 10 rows were matched and were changed.

The previous query also illustrates an important aspect of updates. As you've seen, updates have two phases: a matching phase — where rows are found that match the `WHERE` clause — and a modification phase, where the rows that need changing are updated.

Exploring Databases and Tables with SHOW and mysqlshow

We've already explained how you can use the `SHOW` command to obtain information on the structure of a database, its tables, and the table columns. In this section, we'll review the most common types of `SHOW` statement with brief examples using the `sakila` database. The `mysqlshow` command-line program performs the same function as several `SHOW` command variants, but without needing to start the MySQL client.

The `SHOW DATABASES` statement lists the databases you can access. If you've followed our sample database installation steps in "Entity Relationship Modeling Examples" and deployed the bank model in "Creating a Bank Database ER Model", your output should be as follows:

```
mysql> SHOW DATABASES;

+--------------------+
| Database           |
+--------------------+
| information_schema |
| bank_model         |
| employees          |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
8 rows in set (0.01 sec)
```

These are the databases that you can access with the `USE` command; as we explain in "Using the Sakila Database", you can't see databases for which you have no access privileges unless you have the global `SHOW DATABASES` privilege. You can get the same effect from the command line using the `mysqlshow` program:

```
# mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306
```

You can add a `LIKE` clause to `SHOW DATABASES`. This statement is useful only if you have many databases and want a shortlist as output. For example, to see databases beginning with `s`, run:

```
mysql> SHOW DATABASES LIKE 's%';

+---------------+
| Database (s%) |
+---------------+
| sakila        |
| sys           |
+---------------+
2 rows in set (0.00 sec)
```

The `LIKE` statement's syntax is identical to that in its use in `SELECT`.

To see the statement used to create a database, you can use the `SHOW CREATE DATABASE` statement. For example, to see how you created `sakila`, type:

```
mysql> SHOW CREATE DATABASE sakila;

+----------+------------------------------------------------------
------
------------------------------------------------------------------
------+
| Database | Create Database
|
+----------+------------------------------------------------------
------
------------------------------------------------------------------
------+
| sakila   | CREATE DATABASE `sakila` /*!40100 DEFAULT CHARACTER
SET
utf8mb4 COLLATE utf8mb4_0900_ai_ci */ /*!80016 DEFAULT
ENCRYPTION='N'
*/ |
+----------+------------------------------------------------------
------
------------------------------------------------------------------
------+
1 row in set (0.00 sec)
```

This is perhaps the least exciting `SHOW` statement; it only displays the statement:

```
CREATE DATABASE `sakila` /*!40100 DEFAULT CHARACTER SET utf8mb4
COLLATE utf8mb4_0900_ai_ci */ /*!80016 DEFAULT ENCRYPTION='N' */
```

Some additional keywords are enclosed between the comment symbols `□0□!` and `□1□`:

```
40100 DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci
80016 DEFAULT ENCRYPTION='N'
```

These instructions contain MySQL-specific extensions to standard SQL that are unlikely to be understood by other database programs. A database server other than MySQL would ignore this comment text, and so the syntax is usable by both MySQL and other database server software. The optional number `40100` indicates the minimum version of MySQL that can process this particular instruction — in this case, version 4.01.00; older versions of MySQL ignore such instructions. You'll learn about creating databases in Chapter 4.

The `SHOW TABLES` statement lists the tables in a database. To check the tables in `sakila`, type:

```
mysql> SHOW TABLES FROM sakila;

+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| address                    |
| category                   |
| city                       |
| country                    |
| customer                   |
| customer_list              |
| film                       |
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| staff_list                 |
| store                      |
+----------------------------+
23 rows in set (0.01 sec)
```

If you've already selected the `sakila` database with the `USE sakila` command, you can use the shortcut:

```
mysql> SHOW TABLES;
```

You can get a similar result by specifying the database name to the `mysqlshow` program:

```
$ mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306 sakila
```

As with `SHOW DATABASES`, you can't see tables that you don't have privileges for it. This means you can't see tables in a database you can't access, even if you have the `SHOW DATABASES` global privilege.

The `SHOW COLUMNS` statement lists the columns in a table. For example, to check the columns of `country`, type:

```
mysql> SHOW COLUMNS FROM country;
```

```
+-------------+-------------------+------+-----+-----------------
--+-----------------------------------------------+
| Field       | Type              | Null | Key | Default
| Extra                                         |
+-------------+-------------------+------+-----+-----------------
--+-----------------------------------------------+
| country_id  | smallint unsigned | NO   | PRI | NULL
| auto_increment                                |
| country     | varchar(50)       | NO   |     | NULL
|                                               |
| last_update | timestamp         | NO   |     |
CURRENT_TIMESTAMP | DEFAULT_GENERATED on update CURRENT_TIMESTAMP
|
+-------------+-------------------+------+-----+-----------------
--+-----------------------------------------------+
3 rows in set (0.00 sec)
```

The output reports all column names, their types, and sizes, whether they can be `NULL`, whether they are part of a key, their default value, and any extra information. Types, keys, `NULL` values, and defaults are discussed further in Chapter 4. If you haven't already chosen the `sakila` database with the `USE` command, then you can add the database name before the table name, as in `sakila.country`. Unlike the previous `SHOW` statements, you can always see all column names if you have access to a table; it doesn’t matter that you don’t have certain privileges for all columns. You can get a similar result by using `mysqlshow` with the database and table name:

```
$ mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306 sakila country
```

And you will receive:

```
Database: sakila  Table: country
+-------------+-------------------+--------------------+------+--
---+-------------------+-----------------------------------------
------+---------------------------------+---------+
| Field       | Type              | Collation          | Null |
Key | Default           | Extra
| Privileges                      | Comment |
+-------------+-------------------+--------------------+------+--
---+-------------------+-----------------------------------------
------+---------------------------------+---------+
| country_id  | smallint unsigned |                    | NO   |
PRI |                   | auto_increment
| select,insert,update,references |         |
| country     | varchar(50)       | utf8mb4_0900_ai_ci | NO   |
|                   |
| select,insert,update,references |         |
| last_update | timestamp         |                    | NO   |
| CURRENT_TIMESTAMP | DEFAULT_GENERATED on update
CURRENT_TIMESTAMP | select,insert,update,references |         |
+-------------+-------------------+--------------------+------+--
---+-------------------+-----------------------------------------
------+---------------------------------+---------+
```

You can see the statement used to create a particular table using the `SHOW CREATE TABLE` statement; creating tables is a subject of Chapter 4. Some users prefer this output to that of `SHOW COLUMNS`, since it has the familiar format of a `CREATE TABLE` statement. Here’s an example for the `country` table:

```
mysql> SHOW CREATE TABLE country\G

*************************** 1. row ***************************
       Table: country
```

```
Create Table: CREATE TABLE `country` (
  `country_id` smallint unsigned NOT NULL AUTO_INCREMENT,
  `country` varchar(50) NOT NULL,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON
UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`country_id`)
) ENGINE=InnoDB AUTO_INCREMENT=110 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

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; detail on how to load the database is presented in "Entity Relationship Modeling Examples". If you've followed those instructions, you'll already have the database available and know how to restore the database 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 Chapter 5 and Chapter 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 `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 command:

```
mysql> USE lucy;

Database changed
```

As discussed previously in Chapter 3, 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 the database, you can start creating tables, indexes, and other structures using the steps discussed in "Creating Tables".

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 create a database that already exists:

```
mysql> CREATE DATABASE lucy;

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

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. To make your SQL machine-independent, we recommend that you consistently use lowercase names for databases (and for tables, columns, aliases, and indexes). This behavior is controlled by the `lower_case_table_names` parameter. If set to 0, table names are stored as specified and comparisons are case-sensitive. If set to 1, table names are stored in lowercase on disk and comparisons are not case-sensitive. If set to 2, table names are stored as given but compared in lowercase. On Windows the default value is 1. On macOS, the default value 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. There’s a way around this problem: you can enclose the reserved word with the backtick symbol (`) on either side, but it’s more trouble remembering to do so than it’s worth. In addition, you can’t use selected characters in the names: specifically, you can’t use 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 syntax; check
the manual
that corresponds to your MySQL server version for the right
syntax to use
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;`.

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 our examples in this section, we'll assume that the database `sakila` hasn't been created. If you want to follow 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, 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 talk about 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` statement 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 add 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 section.

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`* 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 the 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 later in this section.

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.PRIMARY'
```

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 later in this section.

Column names have fewer restrictions than database and table names. What's more, they're not dependent on the operating system: 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 the semicolon or the dash sign), you'll need to enclose the name with a backtick symbol (`` ` ``) on either side. 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 the columns, as well as other database objects is something of a personal preference when starting anew, or a matter of following standards when working on an existing codebase. We recommend avoiding repeating yourself. Column name `actor_first_name` is going to look redundant when table name precedes it (e.g. in a complex join query): `actor.actor_first_name` or `actor.first_name`. Usually, one exception is done to that: the ubiquitous `id` column name should either not be used or have the table name prepended for clarity. You may use `sakila`, `world`, or `employee` example databases to get an inspiration. Another good practice is to use the underscore character to separate words; you could use underscores or dashes (but remember to escape dashes and other special symbols with a backtick), or omit the word-separating formatting altogether. However, "CamelCase" is harder to read. As with database and table names, the longest column name is 64 characters in length.

Collation and Character Sets

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. When you’re comparing or sorting strings, how MySQL evaluates the result depends on the character set and collation used. Character sets 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.

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 it’s worth mentioning. Mismatched charsets and collations may result in unexpected situations including loss of data and incorrect query results. You may, however, skip this section, as well as some later discussion, and come back when you want to learn about this specifically. That won’t affect other material in this book.

In our previous string-comparison examples, we ignored the collation and character-set 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 character set 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. Outputs below come 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 for 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 collation
| Maxlen |
+----------+---------------------------------+-------------------
--+--------+
| armscii8 | ARMSCII-8 Armenian              |
armscii8_general_ci |      1 |
| ascii    | US ASCII                        | ascii_general_ci
|      1 |
| big5     | Big5 Traditional Chinese        | big5_chinese_ci
|      2 |
| binary   | Binary pseudo charset           | binary
|      1 |
| cp1250   | Windows Central European        | cp1250_general_ci
|      1 |
| cp1251   | Windows Cyrillic                | cp1251_general_ci
|      1 |
...
| ujis     | EUC-JP Japanese                 | ujis_japanese_ci
|      3 |
| utf16    | UTF-16 Unicode                  | utf16_general_ci
|      4 |
| utf16le  | UTF-16LE Unicode                | utf16le_general_ci
|      4 |
| utf32    | UTF-32 Unicode                  | utf32_general_ci
|      4 |
| utf8     | UTF-8 Unicode                   | utf8_general_ci
|      3 |
| utf8mb4  | UTF-8 Unicode                   | utf8mb4_0900_ai_ci
|      4 |
+----------+---------------------------------+-------------------
--+--------+
41 rows in set (0.00 sec)
```

For example, the `latin1` character set is actually the Windows code page 1252 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 one byte. By comparison, if you use the default `utf8mb4` character set, each character would take up to four bytes of storage. Sometimes, it makes sense to change that default. For example, there’s no reason to store *base64-encoded* data (which, by definition, is ASCII) in `utf8mb4`. With a 128 character wide column, at million rows you’re looking at approximately 350MiB of overhead on charset alone in the worst case.

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

```
mysql> SHOW COLLATION;

+--------------------+----------+-----+---------+----------+----
-----+---------------+
| Collation          | Charset  | Id  | Default | Compiled |
Sortlen | Pad_attribute |
+--------------------+----------+-----+---------+----------+----
-----+---------------+
| armscii8_bin       | armscii8 |  64 |         | Yes      |
1 | PAD SPACE     |
| armscii8_general_ci | armscii8 |  32 | Yes     | Yes      |
1 | PAD SPACE     |
| ascii_bin          | ascii    |  65 |         | Yes      |
1 | PAD SPACE     |
| ascii_general_ci   | ascii    |  11 | Yes     | Yes      |
1 | PAD SPACE     |
...
| utf8mb4_0900_ai_ci  | utf8mb4  | 255 | Yes     | Yes      |
0 | NO PAD        |
| utf8mb4_0900_as_ci  | utf8mb4  | 305 |         | Yes      |
0 | NO PAD        |
| utf8mb4_0900_as_cs  | utf8mb4  | 278 |         | Yes      |
0 | NO PAD        |
| utf8mb4_0900_bin    | utf8mb4  | 309 |         | Yes      |
1 | NO PAD        |
...
| utf8_unicode_ci     | utf8     | 192 |         | Yes      |
8 | PAD SPACE     |
| utf8_vietnamese_ci  | utf8     | 215 |         | Yes      |
8 | PAD SPACE     |
+--------------------+----------+-----+---------+----------+----
-----+---------------+
272 rows in set (0.02 sec)
```

NOTE

Number of character sets and collations available depends on how the MySQL server was built and packaged. Examples we show come from a default MySQL 8.0 installation, and 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/charsets/ |
...
| 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 utf8mb4
    -> 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 your 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 detail of how to do that here. We will show how collations affect string types in "String types".

Other Features

This section briefly describes other features of the MySQL `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 sakila database, unlike a 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_INCREMENT,
    -> 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 0 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 few of which are present in this more full statement. Many of these are advanced and aren't discussed in this book, but you can find more information in the MySQL manual under the heading "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 later in this chapter 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, `sakila` database has a foreign key constraint on a `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, they 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 the table creation time, as well as at a later time. We will not be covering partitioning in this book.

You can check the `CREATE TABLE` statement for 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_TIMESTAMP
        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 occur through using 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=utf8mb4` 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 widely-used datatypes, 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 re-reading this chapter later, and consulting with 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 number. First, the two most popular integer types.

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

The most commonly used numeric type. 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 four 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 an 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. Contrary to 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 same for the purpose of data storage.

`ZEROFILL` is an additional argument, which is used to left-pad the values with zeros up to the length, specified by the *`width`*. 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, `LPAD()` function can be used, or numbers can be stored formatted in `CHAR` column.

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

In the world of growing data sizes, having tables with count of rows in the billions is getting 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 -9223372036854775808 to 9223372036854775807. Unsigned `BIGINT` can store numbers from 0 to 18446744073709551615. Column of this type will require eight 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 extremely careful when dealing with extremely large numbers. First, unsigned big integers larger than 9223372036854775807 should only be used with bit functions. Second, if a result of a arithmetical operation is larger than 9223372036854775807, unexpected results might be observed.

For example:

```
mysql> CREATE TABLE test_bigint (id BIGINT UNSIGNED);

Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO test_bigint VALUES (18446744073709551615);

Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO test_bigint VALUES (18446744073709551615-
1);

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

```
mysql> INSERT INTO test_bigint VALUES
(18446744073709551616*100);

ERROR 1690 (22003): BIGINT value
is out of range in '(18446744073709551616 * 100)'
```

Even though 1844674407370955161600 is less than 18446744073709551615, since signed BIGINT is used for multiplication internally, the out of range error is observed.

Data type SERIAL 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 UNSIGNED INT runs out of range much quicker than you'd expect, and in the worst possible time.

We will only touch other integer types briefly. However, keep in mind that although it's possible to store every integer as BIGINT, that's wasteful in terms of storage space. Moreover, since as we discussed the *width* parameter doesn't constrain the range of values, different integer types can be used for that purpose.

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

As expected from its name, this type holds a small integer, with range from -32768 to 32767 signed, and from 0 to 65535 unsigned. It takes two bytes of storage.

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

Even smaller integer, and the smallest numeric data type. Range of this type is -128 to 127 signed and 0 to 255 unsigned. It only takes one byte of storage.

BOOL[(width)]

Short for `BOOLEAN`, and a synonym for `TINYINT(1)`. Usually, boolean types only accept two values: true or false. However, since `BOOL` in MySQL is an integer type, you can store values from -128 to 127 there. 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.

```
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),(127);

Query OK, 4 rows affected (0.02 sec)

Records: 4  Duplicates: 0  Warnings: 0

mysql> SELECT i, IF(i,'true','false') FROM test_bool;

+------+----------------------+

| 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]`

Another type of integer. Takes 3 bytes of storage space. Stores values in the signed range of -8388608 to 8388607, and unsigned range of 0 to 16777215.

`BIT[(M)]`

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

Fixed-point types

Both `DECIMAL` and `NUMERIC` data types in MySQL are the same. Thus we will only describe `DECIMAL` here, and everything will apply to `NUMERIC`. The main difference between fixed-point types (or type, in MySQL case) 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 other 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.

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

A commonly used numeric type. 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. The maximum value of *`width`* is 255. For example, a column declared as `price`

`DECIMAL(6,2)` should be used to store values in the range –9999.99 to 9999.99. `price DECIMAL(10,4)` would allow values like 123456.1234.

Prior to MySQL version 5.7, if you tried to store a value that's 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 5.7, however, special *SQL mode* is set, which prohibits this, and other unsafe behaviors: `STRICT_TRANS_TABLES`. Using old behavior is possible, but could result in a data loss.

SQL modes are special settings that control behavior of MySQL when it comes to queries. For example, as you can see above, they can restrict "unsafe" behavior. Other modes might 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 is usually required for compatibility with legacy applications across MySQL releases.

The *`width`* is optional, and a value of 10 is assumed when it is omitted. The number of *`decimals`* is optional and, when omitted, a value of 0 is assumed; the maximum value of *`decimals`* should be two less than the value of *`width`*. The maximum value of *`width`* is 65, and *`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` column are stored using a binary format. This format uses four bytes for every nine digits.

Floating-point types

In “Fixed-point types”, we discussed the fixed-point `DECIMAL` type. 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 it to a per-month amount, it’s $4,166 plus 66 and 2/3rds 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 2/3rds 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 2/3rds or pi with a large number of decimal places, allowing accurate approximate representations of exact quantities. You can later use `ROUND()` function to restore 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 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 a rounding with a desired precision. For example, your business might require precision to five decimal places. In this case, you could restore the original:

```
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 value of floating-point data types.

Here are the details of the `DOUBLE` and `FLOAT` 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 type stores small, four-byte, single-precision floating-point values; usually, you use it without providing any parameters. When *`precision`* is between 0 and 24, the default behavior occurs. When *`precision`* is between 25 and 53, the type behaves as for `DOUBLE`. The *`width`* 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 has one optional syntax: it allows an optional number of *`decimals`* and an optional display *`width`*. Without parameters, the type stores normal, eight-byte, double-precision floating point values; usually, you use it without providing any parameters. The *`width`* 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. In MySQL, the string data types are CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT, ENUM, and SET.

[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.

CHAR and VARCHAR types are very similar, but have 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 than or larger than 255 bytes. Note that this is different from string length in characters, as certain character 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 stores strings of arbitrary length (up to the width definied), shorter strings stored will require less storage than CHAR of similar width.

Another difference betwee CHAR and VARCHAR is their handling of trailing spaces. VARCHAR retains trailing spaces up to the specified column length, and will truncate the excess, producing a warning. As will be shown later, CHAR values are right-padded to the column length, 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 VARCHAR(2)
UNIQUE);

Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO test_varchar_trailing VALUES ('a'), (\'a
');
```

```
Query OK, 2 rows affected (0.01 sec)

Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT d, LEGNTH(d) FROM test_varchar_trailing;

+------+-----------+

| d    | LENGTH(d) |

+------+-----------+

| a    |         1 |

| a    |         2 |

+------+-----------+

2 rows in set (0.00 sec)
```

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

```
mysql> INSERT INTO test_varchar_trailing VALUES (\'a  ');

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`*, and so the value stored changes from “a " to “a “. Since we already have a row with such value, a duplicate entry error is reported. This behavior for `VARCHAR` and `TEXT` can be controlled by changing the column collation. Some collations have `PAD SPACE` attribute, meaning that upon retreival they are padded to the *`width`* with spaces. This doesn’t affect storage, but does affect uniqueness checks as well as how `GROUP BY` and

DISTINCT operators work, which we'll discuss in Chapter 5. 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 VALUES ('a');

Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO test_varchar_pad_collation VALUES (\'a ');

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

The NO PAD collations is a new addition of MySQL 8. In prior releases of MySQL, which you may still often see, 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.

Both CHAR and VARCHAR data types disallow storage of values longer than *width*, unless strict SQL mode is disabled. 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 an unaccounted data loss.

Sorting and comparison of VARCHAR, CHAR, and TEXT types happens according to the collation of the character set assigned. You can see that it is possible to specify character set, as well as collation for each individual string type column. It’s also possible to specify `binary` character set, which effectively converts `VARCHAR` into `VARBINARY`. Don’t mistake `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 VALUES ('ä'),
('z');

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)
```

`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 such charset. It’s important to note that MySQL versions 5.7 and 8.0 disagree on what is `utf8` exactly: former using it as an alias for `utf8mb3`, and 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 we discussed in the `VARCHAR` section above, values in `CHAR` columns are always stored at the specified length. Single letter stored in a `CHAR(255)` column will take 255 bytes (in `latin1` charset), and will be padded with spaces. The padding is removed when reading the data, unless

PAD_CHAR_TO_FULL_LENGTH SQL mode is enabled. Worth mentioning again that it means that strings stored in CHAR will lose all of their trailing spaces.

Note that earlier, width of a CHAR column was often associated with bytes. That’s not always the case now, and it’s definitely not the case by default. Multi-byte character sets, such as default utf8mb4 can result in a 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, by default, InnoDB will store CHAR(255) as it would VARCHAR. Let’s see a small 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 ('Plain text',
'Plain text');

Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO test_char_length VALUES ('的開源軟體',
'Plain text');

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

```
mysql> SELECT LENGTH(utf8char), LENGTH(asciichar) FROM
test_char_length;

+------------------+-------------------+
| 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 when 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 collaction, and sorting them is dependent on the numeric values of the bytes in values stored. Instead of character strings, byte strings are stored. Remember that in `VARCHAR` we described the `binary` charset and `BINARY` attribute. Only the `binary` charset "converts" `VARCHAR` or `CHAR` into a respective `BINARY` form. `BINARY` attribute to a charset will not change the fact that character strings are stored. Unlike `VARCHAR` and `CHAR`, *`width`* here is exactly the number of bytes. When *`width`* is omitted for `BINARY`, it defaults to 1.

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

It is important to keep the concept of binary string in mind when working with both of these data types. Even though they’ll accept strings, they aren’t a synonym 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 see 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_data;
```

```
*************************** 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_binary_data;

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

Note how MySQL command-line client actually shows values of binary types in a hex format. We believe that this is much better than 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 CHAR) d2t
    -> FROM test_binary_data;

+------------------+-----------+
| d1t              | d2t       |
+------------------+-----------+
| something        | something |
+-----------------+----------+
1 row in set (0.00 sec)
```

You can also notice 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 many data as you like, and similarly of TEXT for VARCHAR. The BLOB and TEXT types itself can store up to 65,535 bytes or characters respectively. As usual, note that multi-byte charsets do exist. *width* attribute is optional, and when it is specified, MySQL actually will change BLOB or TEXT data type to whatever is the smallest type capable of holding that amount of data. 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 `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 is based on the numeric values of actual bytes stored. For `TEXT`, you may specify 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 you must take a prefix of the value when using it in an index (this is discussed in the next section).

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 a `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 the trailing spaces.

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

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]`

Identical to `BLOB` and `TEXT`, respectively, except that a maximum of 16,777,215 bytes or characters can be stored. Types `LONG` and `LONG VARCHAR` map to `MEDIUMTEXT` data type for compatibility.

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

Identical to `BLOB` and `TEXT`, respectively, except that a maximum of four gigabytes of data can be stored. Note that this is a hard limit even in case of `LONGTEXT`, and thus number of characters in multi-byte 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 64MiB.

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

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 enumerated column can contain `NULL` (stored as `NULL`), the empty string □0□ (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 is 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 'fruit_name' at

row 1
```

Similarly, a list of several allowed values isn’t accepted either:

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

ERROR 1265 (01000): Data truncated for column 'fruit_name' at

row 1
```

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

```
mysql> SELECT * FROM fruits_enum;

+------------+

| fruit_name |

+------------+

| Apple      |

+------------+

1 row in set (0.00 sec)
```

Earlier, MySQL produced a warning instead of error, and stored empty string in place of a wrong value. That behavior can be used by unsetting the default strict SQL mode. It’s also possible 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]`

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 'fruit_name' at

row 1


mysql> INSERT INTO fruits_set VALUES ('Apple,Orange');


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 one byte to store the name length. If the average city name is 10 characters long, this means that using a variable length field will take up 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.

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 timezones. We will try to explain details, but it’s worth re-reading this section and documentation later when you’ll need to actually work with timezones.

`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, and 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 the 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 equivalent is 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` type 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 `0000-00-00` is stored. However, only MySQL versions up to and including 5.6 allow that by default. Both 5.7 and 8.0 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 a MySQL server, 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);
```

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: '2020/02/0'
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: '2020/02/31'
for column 'mydate' at row 1

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

ERROR 1292 (22007): Incorrect date value: '2020/02/100'
for column 'mydate' at row 1
```

Once INSERTs are executed, 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. Sometimes you may need to preserve the actual input and manually process it later. That is possible by unsetting aforementioned SQL modes. Doing so, you could 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 also 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`. Useful for storing 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 from 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*, and *seconds*, using the following formats:

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

The *`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 `TIME` definition doesn't specify the *`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` type column with *`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, 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 incorrect value, a warning is generated and the value is set to zero. You can use the `SHOW WARNINGS` command to reports the details of the warning generated by the previous SQL statement. Our recommendation is to stick to the default STRICT SQL mode. Unlike DATE type, there's seemingly no benefit in allowing incorrect TIME entries, apart from easier error management on application side and maintaiting legacy behaviors.

Let's try all these out in practice:

```
mysql> CREATE TABLE test_time(id SMALLINT, mytime TIME);

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

```
mysql> INSERT INTO test_time VALUES(1, "2 13:25:59");

Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO test_time VALUES(2, "35 13:25:59");

ERROR 1292 (22007): Incorrect time value: '35 13:25:59'
for column 'mytime' at row 1

mysql> INSERT INTO test_time VALUES(3, "900.32");

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 next 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` type 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 `DATETIME` type, but there are few differences. Both types accept a time zone modifier to the input value in the most recent MySQL version. Both types will store and present the data equally to the 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 time zones.

In earlier MySQL versions, preceding 5.6, only `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 types support the behaviors, and any number of columns can do so.

The value stored always matches the template `YYYY-MM-DD HH:MM:SS[.fraction][time zone offset]`, but the value 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 `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 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 detail. You control them 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 want 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 to be set to the current time in each new row or 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. The four-digit version stores values in the range 1901 to 2155, as well as the *zero year*, 0000. Illegal values are converted to the zero date. You can input year values as either strings (such as `'2005'`) or integers (such as `2005`). The `YEAR` type requires one byte of storage space. In the earlier versions of MySQL, it was possible to specify the `digits` parameter. Depending on whether `2` or `4` is passed as the optional `digits` parameter. 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 for 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 sa,e automatic update features as `TIMESTAMP` has. Unless `NOT NULL` attribute is specified for a `DATETIME` column, a NULL value is the default, otherwise the default is 0. Unlike `TIMESTAMP`, `DATETIME` values aren't converted to UTC time zone for storage.

Other types

Currently, as of MySQL 8, spatial and JSON data types fall under this broad category. Using both 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 writing a book on its own.

JSON data type allows a native storage of valid JSON documents. Before MySQL version 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. Moreover, there's no storage optimization performed, and all JSON is just stored in its text form. With native JSON, it's stored in binary format. If we were to summarize in one sentence: use 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 some times multiple `KEY` clauses. The reasons why you need a primary key and secondary keys are 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 the all of the actual table data is stored in an index structure. That is different to 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 that its rows will be sorted according to the clustered index ordering, and actually physically storead 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 results in a generally improved query performance, though can be detrimental to writes. InnoDB does not allow you to choose between clustered and non-clustered tables, thus 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 even that is not available, a hidden clustered index is created, based on ID values assigned by InnoDB to each row.

Given that InnoDB is a 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 the “Alternative Storage Engines”.

When primary key is defined, as we mentioned, 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, B+tree variant), with the exception of indexes on spatial data types, which use R-tree. Other storage engines might implement different index types, however, when table’s storage engine is not specified, you can assume that all indexes are B-tree.

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 which 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 the InnoDB 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 very 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—on other columns or combinations of columns—are extremely fast and in order to avoid sequential scans. For example, take the table `actor` again. Apart from having a primary key on `actor_id`, it also has a secondary key on `last_name`, to improve searching by 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; 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—and 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_INCREMENT,
    -> 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, email));

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 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` to check whether the index is being used. The reason it can be used is because the `first_name` column is the first 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 within that it clusters people with identical first names ordered 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 three characters of `first_name` are indexed, then the first two characters of `last_name`, and then 10 characters from `email`. This is a significant saving 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. That is also mandatory for long types like `TEXT`.

To finish this section, we need to also discuss some peculiarities regarding secondary keys in InnoDB. Remember that all the table data is stored in the leafs 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 the `idx_actor_last_name` for quick filterting, 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 results in significantly bloated secondary keys.

This can also be seen in the `EXPLAIN` output, note the `Extra: Using index` in the first output:

```
mysql> EXPLAIN SELECT actor_id, last_name FROM actor WHERE
last_name = 'Smith'\G

*************************** 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_name =
'Smith'\G

*************************** 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 `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_INCREMENT,
    -> 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',
'Kaidanovsky');

Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO actor VALUES (NULL, 'Anatoly', 'Solonitsyn');

Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO actor VALUES (NULL, 'Nikolai', 'Grinko');

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.

Let’s consider how the new feature works. You can see that the `actor_id` column is declared as an integer with the clauses `NOT NULL AUTO_INCREMENT`. The `AUTO_INCREMENT` keyword 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` 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 (most 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 that is it used on cannot have a `DEFAULT` value.
- There can be only one `AUTO_INCREMENT` column per table.

MySQL supports different storage engines; we’ll learn more about these in “Alternative Storage Engines” in Chapter 7. When using the non-default MyISAM table type, you can use the `AUTO_INCREMENT` feature on keys that comprise multiple columns. In effect, you could have multiple independent counters within a single `AUTO_INCREMENT` column. However, it’s 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 to 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 transaction is rolled back, an identifier would be used up anyway. Consider an example; let's create the table `count` that contains an auto-incrementing field `counter`:

```
mysql> CREATE TABLE count (counter INT AUTO_INCREMENT KEY);

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 numbers. 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 a 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 TO
last_updated_time;

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

This particular example utilizes *online DDL* feature of MySQL. What actually happens behind the scenes is that MySQL only modifies metadata, and doesn't need to actually re-write the table in any way. You can notice that by lack of affected rows. Not all of the *DDL* statements can be performed online, and that's not going to be the case with a lot of other changes. 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 statements like that requires special internal mechanisms, including special locking. You probably wouldn't like tables changing while your queries are running! Special locks are called Metadata Locks in MySQL, and we give a detailed overview of how they work in "Metadata Locks".

Note that `ALTER` and other statements that execute through online DDL still require metadata locks to be obtained. In that sense, they 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 momentarily may wreak havoc. We recommend that you read carefully about the metadata locking in our book and in the 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 that it's worth cautioning you. 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 | Default
|...
+-------------------+-------------------+------+-----+------------
-------+...
| language_id       | tinyint unsigned | NO   | PRI | NULL
|...
| name              | char(20)          | NO   |     | NULL
|...
```

```
| last_updated_time | timestamp         | NO   |     |
CURRENT_TIMESTAMP |...
+-------------------+-------------------+------+-----+------------
-------+...
...+-----------------------------------------------+
...| Extra                                         |
...+-----------------------------------------------+
...| auto_increment                                |
...|                                               |
...| DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
...+-----------------------------------------------+
3 rows in set (0.01 sec)
```

In the previos example we used the `ALTER TABLE` statement with `RENAME COLUMN` keyword. That is a MySQL 8 feature. We could alternatively use `ALTER TABLE` with the `CHANGE` keyword for compatibility.

```
mysql> ALTER TABLE language CHANGE last_update last_updated_time
TIMESTAMP
    -> NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP;

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 not change 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 with 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) DEFAULT 'n/a';

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) DEFAULT
'n/a';

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 | Default
|...
+------------------+------------------+------+-----+------------
-------+...
| language_id      | tinyint unsigned | NO   | PRI | NULL
|...
| name             | char(20)         | YES  |     | n/a
|...
| last_updated_time | timestamp       | NO   |     |
CURRENT_TIMESTAMP |...
| native_name      | char(20)         | YES  |     | NULL
|...
+------------------+------------------+------+-----+------------
-------+...
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) FIRST;

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

mysql> SHOW COLUMNS FROM language;

+------------------+------------------+------+-----+------------
-------+...
| Field            | Type             | Null | Key | Default
|...
+------------------+------------------+------+-----+------------
-------+...
| native_name      | char(20)         | YES  |     | NULL
|...
| language_id      | tinyint unsigned | NO   | PRI | NULL
```

```
|...
| name              | char(20)         | YES  |     | n/a
|...
| last_updated_time | timestamp        | NO   |     |
CURRENT_TIMESTAMP |...
+-------------------+------------------+------+-----+------------
-------+...
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) AFTER name;

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

mysql> SHOW COLUMNS FROM language;

+-------------------+------------------+------+-----+------------
-------+...
| Field             | Type             | Null | Key | Default
|...
+-------------------+------------------+------+-----+------------
-------+...
| language_id       | tinyint unsigned | NO   | PRI | NULL
|...
| name              | char(20)         | YES  |     | n/a
|...
| native_name       | char(20)         | YES  |     | NULL
|...
| last_updated_time | timestamp        | NO   |     |
CURRENT_TIMESTAMP |...
+-------------------+------------------+------+-----+------------
-------+...
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 `formed` 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 index 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 later in "Deleting Structures". Be careful when dropping columns; you discard both the data and the structure of your table. 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. We described `INSERT` statements in "The INSERT Statement" in Chapter 3.

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), MODIFY
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 6 records were change. If you haven't noticed yet, note that we didn't see any records affected when altering tables previously. The difference is that this time, we're not performing an online DDL, because changing any column's type will always result in a table being rebuilt. We recommend reading through the documentation section titled "Online DDL Operations" 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. Modifying indexes does not affect the data stored in the 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 this query, 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_INCREMENT,
  `name` char(255) DEFAULT NULL,
  `last_updated_time` timestamp NOT NULL
    DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  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 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:

- 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 manual under the “ALTER DATABASE” and “ALTER TABLE” headings. 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 is not possible to alter is a name of a particular database. However, if you’re using 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.language;

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'; database
doesn't exist
```

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)
```

Again, you can investigate the warning indicates 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 separating table names with commas:

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

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

You can see three warnings because none of these tables existed.

Chapter 5. Advanced Querying

Over the previous two chapters, you’ve completed an introduction to the basic features of querying and modifying databases with SQL. You should now be able to create, modify, and remove database structures, as well as work with data as you read, insert, delete, and update entries. Over this and the next two chapters, we’ll look at more advanced concepts and then will proceed to more administrative and operations-oriented content. You can skim these chapters and return to read them thoroughly when you’re comfortable using MySQL.

This chapter teaches you more about querying, giving you skills to answer complex information needs. You’ll learn how to do the following:

- Use nicknames, or *aliases*, in queries to save typing and allow a table to be used more than once in a query
- Aggregate data into groups so you can discover sums, averages, and counts
- Join tables in different ways
- Use nested queries
- Save query results in variables so they can be reused in other queries
- Understand why MySQL supports several table types

Aliases

Aliases are nicknames. They give you a shorthand way of expressing a column, table, or function name, allowing you to:

- Write shorter queries

- Express your queries more clearly
- Use one table in two or more ways in a single query
- Access data more easily from programs
- Use special types of nested queries; these are the subject of "Nested Queries", discussed later in this chapter

Column Aliases

Column aliases are useful for improving the expression of your queries, reducing the number of characters you need to type, and making it easier to work with programming languages such as Python or PHP. Consider a simple, not-very-useful example:

```
mysql> SELECT first_name AS 'First Name', last_name AS 'Last
Name'
    -> FROM actor LIMIT 5;

+------------+--------------+
| First Name | Last Name    |
+------------+--------------+
| PENELOPE   | GUINESS      |
| NICK       | WAHLBERG     |
| ED         | CHASE        |
| JENNIFER   | DAVIS        |
| JOHNNY     | LOLLOBRIGIDA |
+------------+--------------+
5 rows in set (0.00 sec)
```

The column `first_name` is aliased as `First Name`, and column `last_name` as `Last Name` . You can see that in the output, the usual column headings, `first_name` and `last_name`, are replaced by the aliases `First Name` and `Last Name`. The advantage is that the aliases might be more meaningful to users. In this case, at the very least, they are more human-readable. Other than that, it's not very useful, but it does illustrate the idea that for a column, you add the keyword AS and then a string that represents what you’d like the column to be known as. Specifying the AS keyword is not required but makes things much clearer.

NOTE

We’ll be using the `LIMIT` clause extensively throughout this chapter, otherwise almost every output will be unwieldy and long. Sometimes, we’ll mention that explicitly, sometimes not. You can experiment on your own by removing `LIMIT` from the queries we give. More information about the `LIMIT` clause can be found in “The LIMIT Clause”.

Now let’s see column aliases doing something useful. Here’s an example that uses a MySQL function and an `ORDER BY` clause:

```
mysql> SELECT CONCAT(first_name, ' ', last_name, ' played in ',
title) AS movie
    -> FROM actor JOIN film_actor USING (actor_id)
    -> JOIN film USING (film_id)
    -> ORDER BY movie LIMIT 20;
```

```
+---------------------------------------------+
| movie                                       |
+---------------------------------------------+
| ADAM GRANT played in ANNIE IDENTITY         |
| ADAM GRANT played in BALLROOM MOCKINGBIRD   |
...
| ADAM GRANT played in TWISTED PIRATES        |
| ADAM GRANT played in WANDA CHAMBER          |
| ADAM HOPPER played in BLINDNESS GUN         |
| ADAM HOPPER played in BLOOD ARGONAUTS       |
+---------------------------------------------+
20 rows in set (0.03 sec)
```

The MySQL function `CONCAT ( )` *concatenates* together the strings that are parameters — in this case, the `first_name`, a constant string with a space, the `last_name`, the constant string `played in`, and the `title` to give output such as `ZERO CAGE played in CANYON STOCK`. We’ve added an alias to the function, `AS movie`, so that we can refer to it easily as `movie` throughout the query. You can see that we do this in the

ORDER BY clause, where we ask MySQL to sort the output by ascending movie value. This is much better than the unaliased alternative, which requires you to write out the CONCAT ( ) function again:

```
mysql> SELECT CONCAT(first_name, " ", last_name, " played in ",
title) AS movie
    -> FROM actor JOIN film_actor USING (actor_id)
    -> JOIN film USING (film_id)
    -> ORDER BY CONCAT(first_name, " ", last_name, " played in ",
title)
    -> LIMIT 20;

+---------------------------------------------+
| movie                                       |
+---------------------------------------------+
| ADAM GRANT played in ANNIE IDENTITY         |
| ADAM GRANT played in BALLROOM MOCKINGBIRD   |
...
| ADAM GRANT played in TWISTED PIRATES        |
| ADAM GRANT played in WANDA CHAMBER          |
| ADAM HOPPER played in BLINDNESS GUN         |
| ADAM HOPPER played in BLOOD ARGONAUTS       |
+---------------------------------------------+
20 rows in set (0.03 sec)
```

The alternative is unwieldy, and worse, you risk mistyping some part of the ORDER BY clause and getting a result different from what you expect. (Note that we've used AS movie on the first line so that the displayed column has the label movie.)

There are restrictions on where you can use column aliases. You can't use them in a WHERE clause, or in the USING and ON clauses that we discuss later in this chapter. This means you can't write a query like this:

```
mysql> SELECT first_name AS name FROM actor WHERE name = 'ZERO
CAGE';

ERROR 1054 (42S22): Unknown column 'name' in 'where clause'
```

You can't do that because MySQL doesn't always know the column values before it executes the WHERE clause. However, you can use column aliases in the ORDER BY clause, and in the GROUP BY and HAVING clauses discussed later in this chapter.

The AS keyword is optional, as we've mentioned. Because of this, the following two queries are equivalent:

```
mysql> SELECT actor_id AS id FROM actor WHERE first_name =
'ZERO';

+----+
| id |
+----+
| 11 |
+----+
1 row in set (0.00 sec)

mysql> SELECT actor_id id FROM actor WHERE first_name = 'ZERO';

+----+
| id |
+----+
| 11 |
+----+
1 row in set (0.00 sec)
```

We recommend using the AS keyword, since it helps to clearly distinguish an aliased column, especially where you're selecting multiple columns from a list of columns separated by commas.

Alias names have a few restrictions. They can be at most 255 characters in length and can contain any character. Aliases don't always need to be quoted, and they follow the same rules as table and column names do, which we described in Chapter 4. If an alias is a single word and doesn't include special symbols—like a dash, or a plus sign, or a space, for example —and is not a keyword—like USE,--then you don't need to put quotes around it. Otherwise, you need to quote that alias, and can use double quotes, single quotes, or backticks. We recommend using lowercase alphanumeric strings for alias names and using a consistent character choice

—such as an underscore—to separate words. Aliases are case-insensitive on all platforms.

Table Aliases

Table aliases are useful for the same reasons as column aliases, but they are also sometimes the only way to express a query. This section shows you how to use table aliases, and "Nested Queries", later in this chapter, shows you other sample queries where table aliases are essential.

Here's a basic table-alias example that shows you how to save some typing:

```
mysql> SELECT ac.actor_id, ac.first_name, ac.last_name, fl.title
FROM
    -> actor AS ac INNER JOIN film_actor AS fla USING (actor_id)
    -> INNER JOIN film AS fl USING (film_id)
    -> WHERE fl.title = 'AFFAIR PREJUDICE';

+----------+------------+-----------+------------------+
| actor_id | first_name | last_name | title            |
+----------+------------+-----------+------------------+
|       41 | JODIE      | DEGENERES | AFFAIR PREJUDICE |
|       81 | SCARLETT   | DAMON     | AFFAIR PREJUDICE |
|       88 | KENNETH    | PESCI     | AFFAIR PREJUDICE |
|      147 | FAY        | WINSLET   | AFFAIR PREJUDICE |
|      162 | OPRAH      | KILMER    | AFFAIR PREJUDICE |
+----------+------------+-----------+------------------+
5 rows in set (0.00 sec)
```

You can see that the `film` and `actor` tables are aliased as `fl` and `ac`, respectively, using the `AS` keyword. This allows you to express column names more compactly, such as `fl.title`. Notice also that you can use table aliases in the `WHERE` clause; unlike column aliases, there are no restrictions on where table aliases can be used in queries. From our example, you can see that we're referring to the table aliases in `SELECT` before they have been defined in `FROM`. There's, however, a catch with table aliases: if an alias has been used for a table, it's impossible to refer to that table without using its new alias. For example, the following statement will error out, as it would if we'd mention `film` in `SELECT`, or in `ORDER BY`:

```
mysql> SELECT ac.actor_id, ac.first_name, ac.last_name, fl.title
FROM
    -> actor AS ac INNER JOIN film_actor AS fla USING (actor_id)
    -> INNER JOIN film AS fl USING (film_id)
    -> WHERE film.title = 'AFFAIR PREJUDICE';
```

```
ERROR 1054 (42S22): Unknown column 'film.title' in 'where clause'
```

As with column aliases, the `AS` keyword is optional. This means that:

```
actor AS ac INNER JOIN film_actor AS fla
```

is the same as

```
actor ac INNER JOIN film_actor fla
```

Again, we prefer the `AS` style because it's clearer to anyone looking at your queries than the alternative. The restrictions on table-alias-name characters and lengths are the same as column aliases, and our recommendations on choosing them are the same, too.

As discussed in the introduction to this section, table aliases allow you to write queries that you can't otherwise easily express. Consider an example: suppose you want to know whether two or more films in our collection have the same title, and if so, what are those films. Let's think about the basic requirement: you want to know if two movies have the same name. To do get that, you might try a query like this:

```
mysql> SELECT * FROM film WHERE title = title;
```

But that doesn't make sense: every film has the same title as itself, and so the query just produces all films as output:

```
+---------+------------------...
| film_id | title            ...
```

```
+---------+------------------...
|       1 | ACADEMY DINOSAUR ...
|       2 | ACE GOLDFINGER   ...
|       3 | ADAPTATION HOLES ...
...
|    1000 | ZORRO ARK        ...
+---------+------------------...
1000 rows in set (0.01 sec)
```

What you really want is to know whether two different films from the film table have the same name. But how can you do that in a single query? The answer is to give the table two different aliases; you then check to see whether one row in the first aliased table matches a row in the second:

```
mysql> SELECT m1.film_id, m2.title
    -> FROM film AS m1, film AS m2
    -> WHERE m1.title = m2.title;
```

```
+---------+-------------------+
| film_id | title             |
+---------+-------------------+
|       1 | ACADEMY DINOSAUR  |
|       2 | ACE GOLDFINGER    |
|       3 | ADAPTATION HOLES  |
...
|     999 | ZOOLANDER FICTION |
|    1000 | ZORRO ARK         |
+---------+-------------------+
1000 rows in set (0.02 sec)
```

But it still doesn’t work! We get all 1000 movies as answers. The reason is that a film still matches itself because it occurs in both aliased tables.

To get the query to work, we need to make sure a movie from one aliased table doesn’t match itself in the other aliased table. The way to do that is to specify that the movies in each table shouldn’t have the same id:

```
mysql> SELECT m1.film_id, m2.title
    -> FROM film AS m1, film AS m2
    -> WHERE m1.title = m2.title
    -> AND m1.film_id <> m2.film_id;
```

```
Empty set (0.00 sec)
```

You can now see that there aren’t two films in the database with the same name. The additional `AND m1.film_id != m2.film_id` stops answers from being reported where the movie id is the same in both tables.

Table aliases are also useful in nested queries that use the `EXISTS` and `ON` clauses. We show you examples later in this chapter when we introduce nested techniques.

Aggregating Data

Aggregate functions allow you to discover the properties of a group of rows. You use them for purposes such as discovering how many rows there are in a table, how many rows in a table share a property (such as having the same name or date of birth), finding averages (such as the average temperature in November), or finding the maximum or minimum values of rows that meet some condition (such as finding the coldest day in August).

This section explains the `GROUP BY` and `HAVING` clauses, the two most commonly used SQL statements for aggregation. But first it explains the `DISTINCT` clause, which is used to report unique results for the output of a query. When neither the `DISTINCT` nor the `GROUP BY` clause is specified, the returned raw data can still be processed using the aggregate functions that we describe in this section.

The DISTINCT Clause

To begin our discussion on aggregate functions, we’ll focus on the `DISTINCT` clause. This isn’t really an aggregate function, but more of a post-processing filter that allows you to remove duplicates. We’ve added it into this section because, like aggregate functions, it’s concerned with picking examples from the output of a query, rather than processing individual rows.

An example is the best way to understand `DISTINCT`. Consider this query:

```
mysql> SELECT DISTINCT first_name
    -> FROM actor JOIN film_actor USING (actor_id);

+-------------+
| first_name  |
+-------------+
| PENELOPE    |
| NICK        |
...
| GREGORY     |
| JOHN        |
| BELA        |
| THORA       |
+-------------+
128 rows in set (0.00 sec)
```

The query finds all first names of all the actors listed in our database that have participated in a film and reports one example of each name. If you remove the `DISTINCT` clause, you get one row of output for each role in every film we have in our database, or 5462 rows. That's a lot of output, so we're limiting to five rows, but you can spot the difference immediately with names being repeated:

```
mysql> SELECT first_name
    -> FROM actor JOIN film_actor USING (actor_id)
    -> LIMIT 5;

+------------+
| first_name |
+------------+
| PENELOPE   |
| PENELOPE   |
| PENELOPE   |
| PENELOPE   |
| PENELOPE   |
+------------+
5 rows in set (0.00 sec)
```

So, the `DISTINCT` clause helps get a summary.

The `DISTINCT` clause applies to the query output and removes rows that have identical values in the columns selected for output in the query. If you rephrase the previous query to output both `first_name` and `last_name` (but otherwise don't change the `JOIN` clause and still use `DISTINCT`), you'll get 199 rows in the output (that's why we use last names):

```
mysql> SELECT DISTINCT first_name, last_name
    -> FROM actor JOIN film_actor USING (actor_id);
```

```
+-------------+--------------+
| first_name  | last_name    |
+-------------+--------------+
| PENELOPE    | GUINESS      |
| NICK        | WAHLBERG     |
...
| JULIA       | FAWCETT      |
| THORA       | TEMPLE       |
+-------------+--------------+
199 rows in set (0.00 sec)
```

Unfortunately, people's names even when last names are added still make for a bad unique key. There are 200 rows in the `actor` table in `sakila` database, and we're missing one of them. You should remember this issue, as using `DISTINCT` indiscriminately may result in wrong query results.

To remove duplicates, MySQL needs to sort the output. If indexes are available that are in the same order as required for the sort — or the data itself is in an order that's useful — this process has very little overhead. However, for large tables and without an easy way of accessing the data in the right order, sorting can be very slow. You should use `DISTINCT` (and other aggregate functions) with caution on large data sets. If you do use it, you can check its behavior using the `EXPLAIN` statement discussed in Chapter 7.

The GROUP BY Clause

The `GROUP BY` clause groups output data for the purpose of aggregation. Particularly, that allows us to use aggregate functions (covered later in "Aggregate functions") on our data, when our projection (that is, contents of the SELECT clause) contains columns other than those within an aggregate function. GROUP BY is similar to ORDER BY in that it takes a list of columns as an argument. However, these clauses are evaluated at different times and are only similar in how they look, not how they operate.

Let’s take a look at a few GROUP BY examples that will demonstrate what it can be used for. In its most basic form, when we list every column we SELECT in GROUP BY, we end up with a DISTINCT equivalent. We’ve already established that a first name is not a unique identifier for an actor.

```
mysql> SELECT first_name FROM actor
    -> WHERE first_name IN ('GENE', 'MERYL');

+------------+
| first_name |
+------------+
| GENE       |
| GENE       |
| MERYL      |
| GENE       |
| MERYL      |
+------------+
5 rows in set (0.00 sec)
```

We can tell MySQL to group the output by a given column. In this case, we only have one, so let’s do that:

```
mysql> SELECT first_name FROM actor
    -> WHERE first_name IN ('GENE', 'MERYL')
    -> GROUP BY first_name;

+------------+
| first_name |
+------------+
| GENE       |
| MERYL      |
+------------+
2 rows in set (0.00 sec)
```

You can see that the original five rows were “folded”, or, more accurately, grouped into just two resulting rows. Not very helpful, as DISTINCT could do the same. It’s worth mentioning, however, that this is not always going to be the case. `DISTINCT` and `GROUP BY` are evaluated and executed at the very different stages of query execution, so you should not confuse them, even if sometimes the effects are similar.

By the SQL standard, every column projected in the `SELECT` clause, which is not part of an aggregate function, should be listed in the `GROUP BY` clause. The only time this rule may be violated is when the resulting groups have only one row each. If you think about it, that’s logical: when you select first name and last name from the actor table, and group only by first name, how should the database behave? It cannot output a few rows with the same first name, as that goes against the grouping rules, but there may be more last names for a given name than one. For a long time, MySQL extended the standard by allowing to `GROUP BY` based on fewer columns than defined in `SELECT`. What did it do with the extra columns? Well, it output some value in a non-deterministic way. For example, when you grouped the first name, but didn’t the last name, from two rows `GENE, WILLIS` and `GENE, HOPKINS` you could either get first or second. That’s a non-standard and dangerous behavior. Imagine that for a year, you got the last name as if it was ordered by the alphabetic order, and came to rely on that. Then the table was re-organized and the order changed. Your authors firmly believe that SQL standard is correct to limit such behaviors.

We may also note here that while every column in `SELECT` must either be used in `GROUP BY` or in an aggregate function, you can `GROUP BY` columns that are not part of the `SELECT`. You’ll see some examples of that later.

Now let’s construct a more useful example. An actor usually takes part in many films throughout their career. We may want to find out just how many films a particular actor played in, or do a calculation for each actor we know of and get an actor rating by productivity. To start, we can use the techniques we’ve learned so far and perform an `INNER JOIN` between `actor` and `film_actor` tables. We don’t need the `film` table as we’re not looking for any details. We can then order the output by actor’s name making it easier to count what we want.

```
mysql> SELECT first_name, last_name, film_id
    -> FROM actor INNER JOIN film_actor USING (actor_id)
    -> ORDER BY first_name, last_name LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| ADAM       | GRANT     |      26 |
| ADAM       | GRANT     |      52 |
| ADAM       | GRANT     |     233 |
| ADAM       | GRANT     |     317 |
| ADAM       | GRANT     |     359 |
| ADAM       | GRANT     |     362 |
| ADAM       | GRANT     |     385 |
| ADAM       | GRANT     |     399 |
| ADAM       | GRANT     |     450 |
| ADAM       | GRANT     |     532 |
| ADAM       | GRANT     |     560 |
| ADAM       | GRANT     |     574 |
| ADAM       | GRANT     |     638 |
| ADAM       | GRANT     |     773 |
| ADAM       | GRANT     |     833 |
| ADAM       | GRANT     |     874 |
| ADAM       | GRANT     |     918 |
| ADAM       | GRANT     |     956 |
| ADAM       | HOPPER    |      81 |
| ADAM       | HOPPER    |      82 |
+------------+-----------+---------+
20 rows in set (0.01 sec)
```

By running down the list, it’s easy to count off how many films we’ve got for each actor, or at least for Adam Grant. Without a `LIMIT`, however, the query would return 5462 distinct rows and calculating our counts would take a lot of time. The `GROUP BY` clause can help automate this process by grouping the movies by actor; we can then use the `COUNT()` function to count off the number of films in each group. Finally, we can use `ORDER BY` and `LIMIT` to get the top ten actors by the number of films they appeared in. Here’s the query that does what we want:

```
mysql> SELECT first_name, last_name, COUNT(film_id) AS num_films
FROM
    -> actor INNER JOIN film_actor USING (actor_id)
    -> GROUP BY first_name, last_name
    -> ORDER BY num_films DESC LIMIT 5;

+------------+-------------+-----------+
| first_name | last_name   | num_films |
+------------+-------------+-----------+
| SUSAN      | DAVIS       |        54 |
| GINA       | DEGENERES   |        42 |
| WALTER     | TORN        |        41 |
| MARY       | KEITEL      |        40 |
| MATTHEW    | CARREY      |        39 |
| SANDRA     | KILMER      |        37 |
| SCARLETT   | DAMON       |        36 |
| VAL        | BOLGER      |        35 |
| ANGELA     | WITHERSPOON |        35 |
| UMA        | WOOD        |        35 |
+------------+-------------+-----------+
10 rows in set (0.01 sec)
```

You can see that the output we've asked for is `first_name, last_name, COUNT(film_id) as num_films`, and this tells us exactly what we wanted to know. We group our data by `first_name` and `last_name` columns, running the `COUNT()` aggregate function in the process. For each "bucket" of rows we got in the previous query, we now get only a single row, albeit giving the information we want. Notice how we've combined `GROUP BY` and `ORDER BY` to get the ordering we wanted, that is by the number of films from more to fewer. `GROUP BY` doesn't guarantee ordering, only grouping. Finally, we `LIMIT` the output to ten rows representing our most productive actors, otherwise we'd get 199 rows of output.

Let's consider the query further. We'll start with the `GROUP BY` clause. This tells us how to put rows together into groups: in this example, we're telling MySQL that the way to group rows is by `first_name, last_name`. The result is that rows for actors with the same name form a cluster (or a bucket) — that is, each distinct name becomes one group. Once the rows are grouped, they're treated in the rest of the query as if they're one row. So, for example, when we write `SELECT first_name, last_name`, we get just one row for each group. This is exactly the same as `DISTINCT`, as we've already discussed. The `COUNT()` function tells us about the properties of the group. More specifically, it tells us the number of rows that form each group; you can count any column in a group, and you'll get the same answer, so `COUNT(film_id)` is almost always the same as `COUNT(*)` or `COUNT(first_name)`. See "Aggregate functions" for more details on why *almost*. We could also just do `COUNT(1)`, which in some databases is a useful optimization technique, although not in MySQL. Of course, you can use a column alias for the `COUNT()` column.

Let's try another example. Suppose you want to know how many different actors played in each movie, along with the film name and its category, and get five films with the largest crew. Here's the query:

```
mysql> SELECT title, name AS category_name, COUNT(*) AS cnt
    -> FROM film INNER JOIN film_actor USING (film_id)
    -> INNER JOIN film_category USING (film_id)
    -> INNER JOIN category USING (category_id)
    -> GROUP BY film_id, category_id
    -> ORDER BY cnt DESC LIMIT 5;
```

```
+------------------+---------------+-----+
| title            | category_name | cnt |
+------------------+---------------+-----+
| LAMBS CINCINATTI | Games         |  15 |
| CRAZY HOME       | Comedy        |  13 |
| CHITTY LOCK      | Drama         |  13 |
| RANDOM GO        | Sci-Fi        |  13 |
| DRACULA CRYSTAL  | Classics      |  13 |
+------------------+---------------+-----+
5 rows in set (0.03 sec)
```

Before we discuss what's new, think about the general function of the query. We join four tables together using `INNER JOIN`: `film`, `film_actor`, `film_category`, and `category` using the identifier columns. Forgetting the aggregation for a moment, the output of this query is one row per a combination of movie and actor.

The GROUP BY clause puts the rows together into clusters. In this query, we want the films grouped together with their category. So, the GROUP BY clause uses film_id and category_id to do that. You can use the film_id from any of the three tables; film.film_id, film_actor.film_id, or film_category.film_id are the same for this purpose. It doesn't matter since the INNER JOIN makes sure they match anyway. The same applies to category_id.

Even though it's required to list every non-aggregated column in GROUP BY, you can GROUP BY on columns outside of the SELECT.

As in the previous example query, we're using the COUNT() function to tell us how many rows are in each group. For example, you can see that COUNT(*) tells us that there are 15 actors in the *LAMBS CINCINATTI* game show. Again, it doesn't matter what column or columns you count in the query: for example, COUNT(*) has the same effect as COUNT(film.film_id) or COUNT(category.name).

We're then ordering the output by the COUNT(*) column aliased cnt in a descending order and pick the first five rows. Note how there are multiple rows with cnt equal to thirteen. In fact, there are even more of those—six total—in the database, making this ordering a bit unfair, as movies having the same number of actors will be sorted randomly. You may add another column to the ORDER BY like title, to make sorting more predictable.

Let's try another example. Sakila is not only about movies and actors: it's a video rental place database, after all. We have, among other things, the customer information, as well as data on what films they rented. Say we want to know five customers that rent movies from the same category most. For example, we might want to adjust our ads based on whether a person likes different categories or sticks to a single one most of the time. We need to carefully think about our grouping: we don't need to group by movie, as that'd just give us the number of times a customer rented it. The resulting query is quite complex, although it's still a variation on INNER JOIN and GROUP BY.

```
mysql> SELECT email, name AS category_name, COUNT(category_id) AS
cnt
    -> FROM customer cs INNER JOIN rental USING (customer_id)
    -> INNER JOIN inventory USING (inventory_id)
    -> INNER JOIN film_category USING (film_id)
    -> INNER JOIN category cat USING (category_id)
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC LIMIT 5;
```

```
+----------------------------------+---------------+-----+
| email                            | category_name | cnt |
+----------------------------------+---------------+-----+
| WESLEY.BULL@sakilacustomer.org   | Games         |   9 |
| ALMA.AUSTIN@sakilacustomer.org   | Animation     |   8 |
| KARL.SEAL@sakilacustomer.org     | Animation     |   8 |
| LYDIA.BURKE@sakilacustomer.org   | Documentary   |   8 |
| NATHAN.RUNYON@sakilacustomer.org | Animation     |   7 |
+----------------------------------+---------------+-----+
5 rows in set (0.08 sec)
```

You can see that some customers do enjoy renting films from the same category multiple times. What we don't know is if any of them rented the same movie multiple times, or if those were all different movies within a category. The `GROUP BY` clause hides the details. Again, we use `COUNT(*)` to do the counting of rows in the groups, and you can see the `INNER JOIN` spread over lines 2 to 5 in the query.

The interesting thing about this query is that we didn't specify column names for `GROUP BY` or `ORDER BY` clauses explicitly. Instead, we used the columns' position numbers (counted from 1) as they appear in the `SELECT` clause. This technique saves on typing, but can be problematic if you later decide to add another column in the `SELECT`, which would break the ordering.

We should talk about the danger of `GROUP BY`, as we've previously mentioned for `DISTINCT`. Consider the following query:

```
mysql> SELECT COUNT(*) FROM actor GROUP BY first_name, last_name;
```

```
+----------+
| COUNT(*) |
+----------+
|        1 |
|        1 |
...
|        1 |
|        1 |
+----------+
199 rows in set (0.00 sec)
```

Looks simple enough, and it produces the number of times a combination of a given first name and last name was found in the `actor` table. You could even assume that it just outputs 199 rows of digit `1`. However, if we do a `COUNT(*)` on the `actor` table we get 200 rows. What's the catch? Apparently, two actors have the same first name and last name. These things happen, and you have to be mindful of them. When you group based on columns which do not form a unique identifier, you may accidentally group together unrelated rows, resulting in misleading data. To find the duplicates, we can modify a query that we constructed in "Table Aliases" to look for films with the same name:

```
mysql> SELECT a1.actor_id, a1.first_name, a1.last_name
    -> FROM actor AS a1, actor AS a2
    -> WHERE a1.first_name = a2.first_name
    -> AND a1.last_name = a2.last_name
    -> AND a1.actor_id <> a2.actor_id;
```

```
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+------------+-----------+
|      101 | SUSAN      | DAVIS     |
|      110 | SUSAN      | DAVIS     |
+----------+------------+-----------+
2 rows in set (0.00 sec)
```

Before we end this section, let's again touch on how MySQL extends the SQL standard around the `GROUP BY` clause. Before MySQL 5.7, it was possible by default to specify an incomplete column list in the `GROUP BY`, and, as we've explained, that resulted in a random rows output within groups for non-grouped dependent columns. For reasons of supporting legacy sofware, both MySQL 5.7 and 8.0 continue providing this behavior, though it has to be explicitly enabled. The behavior is controlled by the `ONLY_FULL_GROUP_BY` SQL mode, which is set by default. If you find yourself in a situation where you need to port a program relying on the legacy `GROUP BY` behavior, we recommend that you do not resort to changing the SQL mode. There are generally two ways to handle this problem. The first is to understand whether the query logic requires incomplete grouping at all — that is hardly the case. The second is to support the random data behavior for non-grouped columns by using either an aggregate function like `MIN()` or `MAX()`, or by using the special `ANY_VALUE()` aggregate function, which, unsurprisingly, just produces a random value from within a group.

Aggregate functions

We've seen examples of how the `COUNT()` function can be used to tell how many rows are in a group. Here we will cover some other functions commonly used to explore the properties of aggregated rows. But first, we should expand a bit on `COUNT()` as it's frequently used.

`COUNT()`

> Returns the number of rows *or* the number of values in a column. Remember we mentioned that `COUNT(*)` is *almost* always the equivalent of `COUNT(<column>)`. The problem is `NULL`. `COUNT(*)` will do a count of rows returned, regardless of whether the column in those rows is `NULL` or not. However, when you do a `COUNT(<column>)`, only non-NULL values will be counted. In our `sakila` database, a customer's email may be NULL, and we can observe the impact.

```
mysql> SELECT COUNT(*) FROM customer;

+----------+
| count(*) |
+----------+
|      599 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(email) FROM customer;

+--------------+
| count(email) |
+--------------+
|          598 |
+--------------+
1 row in set (0.00 sec)
```

We should also add that `COUNT()` can be run with an internal `DISTINCT` clause like this — `COUNT(DISTINCT <column>)` and will return a number of distinct values instead of all values.

AVG()

Returns the average (mean) of the values in the specified column for all rows in a group. For example, you could use it to find the average cost of a house in a city, when the houses are grouped by city:

```
SELECT AVG(cost) FROM house_prices GROUP BY city;
```

MAX()

Returns the maximum value from rows in a group. For example, you could use it to find the warmest day in a month, when the rows are grouped by month.

`MIN()`

Returns the minimum value from rows in a group. For example, you could use it to find the youngest student in a class, when the rows are grouped by class.

`STD()` or `STDDEV()` or `STDDEV_POP()`

Returns the standard deviation of values from rows in a group. For example, you could use it to understand the spread of test scores, when rows are grouped by university course. All three of these are synonyms. `STD()` is a MySQL extension, `STDDEV()` is added for compatibility with Oracle, and `STDDEV_POP()` is an SQL standard function.

`SUM()`

Returns the sum of values from rows in a group. For example, you could use it to compute the dollar amount of sales in a given month, when rows are grouped by month.

There are other functions available for use with `GROUP BY`; they're less frequently used than the ones we've introduced. You can find more details on them in the MySQL manual under the heading "Aggregate Function Descriptions."

The HAVING Clause

You're now familiar with the `GROUP BY` clause, which allows you to sort and cluster data. You should now be able to use it to find out about counts, averages, minimums, and maximums. This section shows how you can use the `HAVING` clause to add additional control to the aggregation of rows in a `GROUP BY` operation.

Suppose you want to know how many popular actors there are in our database. You've decided to define an actor as popular if they've taken part in at least forty movies. In the previous section, we tried an almost identical query but without the popularity limitation. We also grouped the actors by first and last name, losing one record, so we'll add grouping on the `actor_id`, which we know to be unique. Here's the new query, with an additional `HAVING` clause that adds the constraint:

```
mysql> SELECT first_name, last_name, COUNT(film_id)
    -> FROM actor INNER JOIN film_actor USING (actor_id)
    -> GROUP BY actor_id, first_name, last_name
    -> HAVING COUNT(film_id) > 40
    -> ORDER BY COUNT(film_id) DESC;
```

```
+------------+-----------+----------------+
| first_name | last_name | COUNT(film_id) |
+------------+-----------+----------------+
| GINA       | DEGENERES |             42 |
| WALTER     | TORN      |             41 |
+------------+-----------+----------------+
2 rows in set (0.01 sec)
```

You can see there are only two actors that meet the new criteria.

The `HAVING` clause must contain an expression or column that's listed in the `SELECT` clause. In this example, we've used `HAVING COUNT(film_id) >= 40`, and you can see that `COUNT(film_id)` is part of the `SELECT` clause. Typically, the expression in the `HAVING` clause uses an aggregate function such as `COUNT()`, `SUM()`, `MIN()`, or `MAX()`. If you find yourself wanting to write a `HAVING` clause that uses a column or expression that isn't in the `SELECT` clause, chances are you should be using a `WHERE` clause instead. The `HAVING` clause is only for deciding how to form each group or cluster, not for choosing rows in the output. We'll show you an example later that illustrates when not to use `HAVING`.

Let's try another example. Suppose you want a list of top five movies that were rented more than 30 times, together with the number of times they were rented ordered by popularity in reverse. Here's the query you'd use:

```
mysql> SELECT title, COUNT(rental_id) AS num_rented FROM
    -> film INNER JOIN inventory USING (film_id)
    -> INNER JOIN rental USING (inventory_id)
    -> GROUP BY title
    -> HAVING num_rented > 30
    -> ORDER BY num_rented DESC LIMIT 5;
```

```
+--------------------+------------+
| title              | num_rented |
+--------------------+------------+
| BUCKET BROTHERHOOD |         34 |
| ROCKETEER MOTHER   |         33 |
| FORWARD TEMPLE     |         32 |
| GRIT CLOCKWORK     |         32 |
| JUGGLER HARDLY     |         32 |
+--------------------+------------+
5 rows in set (0.04 sec)
```

You can again see that the expression `COUNT()` is used in both the `SELECT` and `HAVING` clauses. This time, though, we aliased the `COUNT(rental_id)` to `num_rented`, and used the alias in both `HAVING` and `ORDER BY` clauses.

Now let's consider an example where you shouldn't use `HAVING`. You want to know how many films a particular actor played in. Here's the query you shouldn't use:

```
mysql> SELECT first_name, last_name, COUNT(film_id) AS film_cnt
FROM
    -> actor INNER JOIN film_actor USING (actor_id)
    -> GROUP BY actor_id, first_name, last_name
    -> HAVING first_name = 'EMILY' AND last_name = 'DEE';
```

```
+------------+-----------+----------+
| first_name | last_name | film_cnt |
+------------+-----------+----------+
| EMILY      | DEE       |       14 |
+------------+-----------+----------+
1 row in set (0.02 sec)
```

It gets the right answer, but in the wrong — and, for large amounts of data, a much slower — way. It's not the correct way to write the query because the `HAVING` clause isn't being used to decide what rows should form each group, but is instead being incorrectly used to filter the answers to display. For this query, we should really use a `WHERE` clause as follows:

```
mysql> SELECT first_name, last_name, COUNT(film_id) AS film_cnt
FROM
    -> actor INNER JOIN film_actor USING (actor_id)
    -> WHERE first_name = 'EMILY' AND last_name = 'DEE'
    -> GROUP BY actor_id, first_name, last_name;

+------------+-----------+----------+
| first_name | last_name | film_cnt |
+------------+-----------+----------+
| EMILY      | DEE       |       14 |
+------------+-----------+----------+
1 row in set (0.00 sec)
```

This correct query forms the groups, and then picks which groups to display based on the `WHERE` clause.

Advanced Joins

So far in the book, we've used the `INNER JOIN` clause to bring together rows from two or more tables. We'll explain the inner join in more detail in this section, contrasting it with the other join types we discuss: the union, left and right joins, and natural joins. At the conclusion of this section, you'll be able to answer difficult information needs and be familiar with the correct choice of join for the task.

The Inner Join

The `INNER JOIN` clause matches rows between two tables based on the criteria you provide in the `USING` clause. For example, you're very familiar now with an inner join of the `actor` and `film_actor` tables:

```
mysql> SELECT first_name, last_name, film_id FROM
    -> actor INNER JOIN film_actor USING (actor_id)
    -> LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| PENELOPE   | GUINESS   |       1 |
| PENELOPE   | GUINESS   |      23 |
...
| PENELOPE   | GUINESS   |     980 |
| NICK       | WAHLBERG  |       3 |
+------------+-----------+---------+
20 rows in set (0.00 sec)
```

Let’s review the key features of an `INNER JOIN`:

- Two tables (or results of a previous join) are listed on either side of the `INNER JOIN` keyphrase.
- The `USING` clause defines one or more columns that are in both tables or results, and used to join or match rows.
- Rows that don’t match aren’t returned. For example, if you have a row in the `actor` table that doesn’t have any matching films in the `film_actor` table, it won’t be included in the output.

You can actually write inner-join queries with the `WHERE` clause without using the `INNER JOIN` keyphrase. Here’s a rewritten version of the previous query that produces the same result:

```
mysql> SELECT first_name, last_name, film_id
    -> FROM actor, film_actor
    -> WHERE actor.actor_id = film_actor.actor_id
    -> LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| PENELOPE   | GUINESS   |       1 |
| PENELOPE   | GUINESS   |      23 |
...
| PENELOPE   | GUINESS   |     980 |
| NICK       | WAHLBERG  |       3 |
+------------+-----------+---------+
20 rows in set (0.00 sec)
```

You can see that we didn’t spell out the inner join: we’re selecting from the `actor` and `film_actor` tables the rows where the identifiers match between the tables.

You can modify the `INNER JOIN` syntax to express the join criteria in a way that’s similar to using a `WHERE` clause. This is useful if the names of the identifiers don’t match between the tables, although that’s not the case in this example. Here’s the previous query, rewritten in this style:

```
mysql> SELECT first_name, last_name, film_id FROM
    -> actor INNER JOIN film_actor
    -> ON actor.actor_id = film_actor.actor_id
    -> LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| PENELOPE   | GUINESS   |       1 |
| PENELOPE   | GUINESS   |      23 |
...
| PENELOPE   | GUINESS   |     980 |
| NICK       | WAHLBERG  |       3 |
+------------+-----------+---------+
20 rows in set (0.00 sec)
```

You can see that the `ON` clause replaces the `USING` clause, and that the columns that follow are fully specified to include the table and column names. If the columns were named differently and uniquely between two tables, you may have skipped the table name there. There’s no real advantage or disadvantage in using `ON` or a `WHERE` clause; it’s just a matter of taste. Typically, these days, you’ll find most SQL professionals use the `INNER JOIN` with `ON` clause in preference to `WHERE`, but it’s not universal.

Before we move on, let’s consider what purpose the `WHERE`, `ON`, and `USING` clauses serve. If you omit the `WHERE` clause from the query we showed you, you get a very different result. Here’s the query, and the first few lines of output:

```
mysql> SELECT first_name, last_name, film_id
    -> FROM actor, film_actor LIMIT 20;

+------------+-------------+---------+
| first_name | last_name   | film_id |
+------------+-------------+---------+
| THORA      | TEMPLE      |       1 |
| JULIA      | FAWCETT     |       1 |
...
| DEBBIE     | AKROYD      |       1 |
| MATTHEW    | CARREY      |       1 |
+------------+-------------+---------+
20 rows in set (0.00 sec)
```

The output is nonsensical: what's happened is that each row from the `actor` table has been output alongside each row from the `film_actor` table, for all possible combinations. Since there are 200 actors and 5462 records in `film_actor` table, there are 200 × 5462 = 1092400 rows of output, and we know that only 5462 of those combinations actually make sense (there are only 5462 records for actors that played in films). We can see the number of rows we'd get without a `LIMIT`:

```
mysql> SELECT COUNT(*) FROM actor, film_actor;

+----------+
| COUNT(*) |
+----------+
|  1092400 |
+----------+
1 row in set (0.00 sec)
```

This type of query, without a clause that matches rows, is known as a *Cartesian product*. Incidentally, you also get the Cartesian product if you perform an inner join without specifying a column with a `USING` or `ON` clause, as in the query:

```
SELECT first_name, last_name, film_id
FROM actor INNER JOIN film_actor;
```

Later in “The Natural Join”, we’ll introduce the natural join, which is an inner join on identically named columns. While the natural join doesn’t use explicitly specified columns, it still produces an inner join, rather than a Cartesian product.

The keyphrase `INNER JOIN` can be replaced with `JOIN` or `STRAIGHT JOIN`; they all do the same thing. However, `STRAIGHT JOIN` forces MySQL to always read the table on the left before it reads the table on the right. We’ll have a look at how MySQL processes queries behind the scenes in Chapter 7. The keyphrase `JOIN` is the one you’ll see most commonly used: it’s a standard shorthand for `INNER JOIN` used by many other database systems besides MySQL, and we will use it in most of our inner-join examples.

The Union

The `UNION` statement isn’t really a join operator. Rather, it allows you to combine the output of more than one `SELECT` statement to give a consolidated result set. It’s useful in cases where you want to produce a single list from more than one source, or you want to create lists from a single source that are difficult to express in a single query.

Let’s look at an example. If you wanted to output all actor *and* movie *and* customer names in the `sakila` database, you could do this with a `UNION` statement. It’s a contrived example, but you might want to do this just to list all of the text fragments, rather than to meaningfully present the relationships between the data. There’s text in the `actor.first_name`, `film.title`, and `customer.first_name` columns in the `actor`, `film`, and `customer` tables, respectively. Here’s how to display it:

```
mysql> SELECT first_name FROM actor
    -> UNION
    -> SELECT first_name FROM customer
    -> UNION
    -> SELECT title FROM film;
```

```
+-----------------------------+
| first_name                  |
+-----------------------------+
| PENELOPE                    |
| NICK                        |
| ED                          |
...
| ZHIVAGO CORE                |
| ZOOLANDER FICTION           |
| ZORRO ARK                   |
+-----------------------------+
1647 rows in set (0.00 sec)
```

We’ve only shown a few of the 1647 rows. The `UNION` statement outputs all results from all queries together, under a heading appropriate to the first query.

A slightly less contrived example is to create a list of five top rented and five least rented movies in our database. You can do this easily with the `UNION` operator:

```
mysql> (SELECT title, COUNT(rental_id) AS num_rented
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> GROUP BY title ORDER BY num_rented DESC LIMIT 5)
    -> UNION
    -> (SELECT title, COUNT(rental_id) AS num_rented
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> GROUP BY title ORDER BY num_rented ASC LIMIT 5);
```

```
+--------------------+------------+
| title              | num_rented |
+--------------------+------------+
| BUCKET BROTHERHOOD |         34 |
| ROCKETEER MOTHER   |         33 |
| FORWARD TEMPLE     |         32 |
| GRIT CLOCKWORK     |         32 |
| JUGGLER HARDLY     |         32 |
| TRAIN BUNCH        |          4 |
| HARDLY ROBBERS     |          4 |
| MIXED DOORS        |          4 |
| BUNCH MINDS        |          5 |
| BRAVEHEART HUMAN   |          5 |
+--------------------+------------+
10 rows in set (0.04 sec)
```

The first query uses `ORDER BY` with the `DESC` (descending) modifier and a `LIMIT 5` clause to find the top five movies rented. The second query uses `ORDER BY` with the `ASC` (ascending) modifier and a `LIMIT 5` clause to find the five movies least rented. The `UNION` combines the result sets. Because there are multiple titles with the same `num_rented` value, ordering of titles with the same value in `num_rented` is not guaranteed to be determined. You may see different titles listed for `num_rented` of 32 and 5 on your end.

The `UNION` operator has several limitations:

- The output is labeled with the names of the columns or expressions from the first query. Use column aliases to change this behavior.
- The queries should output the same number of columns. If you try using different numbers of columns, MySQL will report an error.
- All matching columns should have the same type. So, for example, if the first column output from the first query is a date, the first column output from any other query must be a date.
- The results returned are unique, as if you'd applied a `DISTINCT` to the overall result set. To see this in action, let's try a pretty simple example. Remember we had issues with actor's names — first name is a bad unique identifier. If we select two actors with the same first name, and `UNION` the two queries, we will end up with just one row. The implicit `DISTINCT` operation hides the duplicate (for `UNION`) rows:

  ```
  mysql> SELECT first_name FROM actor WHERE actor_id = 88
      -> UNION
      -> SELECT first_name FROM actor WHERE actor_id = 169;
  ```

```
+------------+
| first_name |
+------------+
| KENNETH    |
+------------+
1 row in set (0.01 sec)
```

If you want to show any duplicates, replace `UNION` with `UNION ALL`:

```
mysql> SELECT first_name FROM actor WHERE actor_id = 88
    -> UNION ALL
    -> SELECT first_name FROM actor WHERE actor_id = 169;

+------------+
| first_name |
+------------+
| KENNETH    |
| KENNETH    |
+------------+
2 rows in set (0.00 sec)
```

Here, the first name `KENNETH` appears twice.

The implicit `DISTINCT` that `UNION` performs has a non-zero cost from the performance side of things. Whenever you use `UNION`, see whether `UNION ALL` fits logically, and if it can improve query performance.

- If you want to apply `LIMIT` or `ORDER BY` to an individual query that is part of a `UNION` statement, enclose that query in parentheses (as shown in the previous example). It's useful to use parentheses anyway to keep the query easy to understand.

The UNION operation simply concatenates the results of the component queries with no attention to order, so there's not much point in using ORDER BY within one of the subqueries. The only time that it makes sense to order a subquery in a UNION operation is when you want to select a subset of results. In our example, we've ordered the movies by the number of times they were rented, and then selected only the top five (in the first subquery) and the bottom five (in the second subquery).

For efficiency, MySQL will actually ignore an ORDER BY clause within a subquery if it's used without LIMIT. Let's look at some examples to see exactly how this works.

First, let's run a simple query to list the rental information for a particular movie, along with the time the rent happened. We've enclosed the query in parentheses for consistency with our other examples — the parentheses don't actually have any effect here — and haven't used an ORDER BY or LIMIT clause:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998);

+--------------+---------------------+-------------------
--+
| title        | rental_date         | return_date
|
+--------------+---------------------+-------------------
--+
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21
00:19:20 |
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12
09:47:57 |
| ZHIVAGO CORE | 2005-07-27 14:53:55 | 2005-07-31
```

```
19:48:55 |
| ZHIVAGO CORE | 2005-08-20 17:18:48 | 2005-08-26
15:31:48 |
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07
00:49:20 |
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26
09:48:54 |
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16
03:43:02 |
| ZHIVAGO CORE | 2005-08-02 02:05:04 | 2005-08-10
21:58:04 |
| ZHIVAGO CORE | 2006-02-14 15:16:03 | NULL
|
+--------------+---------------------+-------------------
--+
9 rows in set (0.00 sec)
```

The query returns all the times the movie was rented, in no particular order (see the fourth and fifth entries).

Now, let’s add an `ORDER BY` clause to this query:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC);
```

```
+--------------+---------------------+-------------------
--+
| title        | rental_date         | return_date
|
+--------------+---------------------+-------------------
--+
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07
```

```
00:49:20 |
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21
00:19:20 |
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26
09:48:54 |
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12
09:47:57 |
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16
03:43:02 |
| ZHIVAGO CORE | 2005-07-27 14:53:55 | 2005-07-31
19:48:55 |
| ZHIVAGO CORE | 2005-08-02 02:05:04 | 2005-08-10
21:58:04 |
| ZHIVAGO CORE | 2005-08-20 17:18:48 | 2005-08-26
15:31:48 |
| ZHIVAGO CORE | 2006-02-14 15:16:03 | NULL
|
+--------------+---------------------+-------------------
--+
9 rows in set (0.00 sec)
```

As expected, we get all the times the movie was rented, in the order of the rent date.

Adding a `LIMIT` clause to the previous query selects the first five rents, in chronological order — no surprises here:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC LIMIT 5);

+--------------+---------------------+-------------------
--+
| title        | rental_date         | return_date
|
+--------------+---------------------+-------------------
--+
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07
00:49:20 |
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21
00:19:20 |
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26
09:48:54 |
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12
09:47:57 |
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16
03:43:02 |
+--------------+---------------------+-------------------
--+
5 rows in set (0.01 sec)
```

Now, let's see what happens when we perform a `UNION` operation. In this example, we're using two subqueries, each with an `ORDER BY` clause. We've used a `LIMIT` clause for the second subquery, but not for the first:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC)
    -> UNION ALL
    -> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC LIMIT 5);
```

```
+--------------+---------------------+-------------------
--+
| title        | rental_date         | return_date
|
+--------------+---------------------+-------------------
--+
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21
00:19:20 |
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12
09:47:57 |
| ZHIVAGO CORE | 2005-07-27 14:53:55 | 2005-07-31
19:48:55 |
| ZHIVAGO CORE | 2005-08-20 17:18:48 | 2005-08-26
15:31:48 |
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07
00:49:20 |
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26
09:48:54 |
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16
03:43:02 |
| ZHIVAGO CORE | 2005-08-02 02:05:04 | 2005-08-10
21:58:04 |
| ZHIVAGO CORE | 2006-02-14 15:16:03 | NULL
|
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07
00:49:20 |
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21
00:19:20 |
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26
09:48:54 |
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12
09:47:57 |
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16
03:43:02 |
+--------------+---------------------+-------------------
--+
14 rows in set (0.01 sec)
```

As expected, the first subquery returns all the times the movie was rented (the first 9 rows of this output), and the second subquery returns the first 5 rentals (the last 5 rows of this output). Notice how the first 9 rows are not in order (see the fourth and fifth rows), even though the first subquery does have a `ORDER BY` clause. Since we're performing a `UNION` operation, the MySQL server has decided that there's no point sorting the result of the subquery. The second subquery includes a `LIMIT` operation, so the results of that subquery are sorted.

The output of a `UNION` operation isn't guaranteed to be ordered, even if the subqueries are ordered, so if you want the final output to be ordered, you should add an `ORDER BY` clause at the end of the whole query. Note that it can be in another order from the subqueries. See the following:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC)
    -> UNION ALL
    -> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC LIMIT 5)
    -> ORDER BY rental_date DESC;
```

```
+--------------+---------------------+-------------------
--+
| title        | rental_date         | return_date
|
+--------------+---------------------+-------------------
--+
| ZHIVAGO CORE | 2006-02-14 15:16:03 | NULL
|
| ZHIVAGO CORE | 2005-08-20 17:18:48 | 2005-08-26
15:31:48 |
| ZHIVAGO CORE | 2005-08-02 02:05:04 | 2005-08-10
21:58:04 |
| ZHIVAGO CORE | 2005-07-27 14:53:55 | 2005-07-31
19:48:55 |
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16
03:43:02 |
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16
03:43:02 |
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12
09:47:57 |
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12
09:47:57 |
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26
09:48:54 |
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26
09:48:54 |
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21
00:19:20 |
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21
00:19:20 |
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07
00:49:20 |
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07
00:49:20 |
+--------------+--------------------+------------------
--+
14 rows in set (0.00 sec)
```

Here’s another example of sorting the final results, including a limit on the number of returned results:

```
mysql> (SELECT first_name, last_name FROM actor WHERE
actor_id < 5)
    -> UNION
    -> (SELECT first_name, last_name FROM actor WHERE
actor_id > 190)
    -> ORDER BY first_name LIMIT 4;

+------------+-----------+
| first_name | last_name |
+------------+-----------+
| BELA       | WALKEN    |
| BURT       | TEMPLE    |
| ED         | CHASE     |
| GREGORY    | GOODING   |
+------------+-----------+
4 rows in set (0.00 sec)
```

The `UNION` operation is somewhat unwieldy, and there are generally alternative ways of getting the same result. For example, the previous query could have been written more simply like this:

```
mysql> SELECT first_name, last_name FROM actor
    -> WHERE actor_id < 5 OR actor_id > 190
    -> ORDER BY first_name LIMIT 4;

+------------+-----------+
| first_name | last_name |
+------------+-----------+
| BELA       | WALKEN    |
| BURT       | TEMPLE    |
| ED         | CHASE     |
| GREGORY    | GOODING   |
+------------+-----------+
4 rows in set (0.00 sec)
```

The Left and Right Joins

The joins we've discussed so far output only rows that match between tables. For example, when you join the `film` and `rental` tables through the `inventory` table, you see only the films that were rented. Therefore, rows for films that haven't been rented are ignored. This makes sense in many cases, but it isn't the only way to join data. This section explains other options you have.

Suppose you did want a comprehensive list of all films and the number of times they've been rented. Unlike the example earlier in this chapter, included in the list you want to see a zero next to movies that haven't been rented. You can do this with a left join, a different type of join that's driven by one of the two tables participating in the join. A left join works like this: each row in the left table — the one that's doing the driving — is processed and output, with the matching data from the second table if it exists and `NULL` values if there is no matching data in the second table. We'll show you how to write this type of query later in this section, but we'll start with a simpler example.

Here's a simple `LEFT JOIN` example. You want to list all movies, and next to each movie you want to show when it was rented. If a movie has never been rented, you want to see that. If it's been rented many times, you want to see that too. Here's the query:

```
mysql> SELECT title, rental_date
    -> FROM film LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id);
```

```
+------------------------------+---------------------+
| title                        | rental_date         |
+------------------------------+---------------------+
| ACADEMY DINOSAUR             | 2005-07-08 19:03:15 |
| ACADEMY DINOSAUR             | 2005-08-02 20:13:10 |
| ACADEMY DINOSAUR             | 2005-08-21 21:27:43 |
...
| WAKE JAWS                    | NULL                |
| WALLS ARTIST                 | NULL                |
...
| ZORRO ARK                    | 2005-07-31 07:32:21 |
| ZORRO ARK                    | 2005-08-19 03:49:28 |
+------------------------------+---------------------+
16087 rows in set (0.06 sec)
```

You can see what happens: movies that have been rented have dates and times, and those that haven't don't (the `rental_date` value is `NULL`). Let's also discuss the fact that we `LEFT JOIN` twice in this example. First, we join `film` and `inventory`, and we want to make sure that even if a movie is not in our inventory (and thus cannot be rented by definition), we still output it. Then we join the `rental` table with the dataset resulting from prior join. We use the `LEFT JOIN` again, as we may have films that are not in our inventory, and those won't have any row in `rental` table, too. However, we may also have films listed in our inventory that just haven't been rented. That's why we need to `LEFT JOIN` both tables here.

The order of the tables in the `LEFT JOIN` is important. If you reverse the order in the previous query, you get very different output:

```
mysql> SELECT title, rental_date
    -> FROM rental LEFT JOIN inventory USING (inventory_id)
    -> LEFT JOIN film USING (film_id)
    -> ORDER BY rental_date DESC;
```

```
+------------------------------+---------------------+
| title                        | rental_date         |
+------------------------------+---------------------+
...
| LOVE SUICIDES                | 2005-05-24 23:04:41 |
| GRADUATE LORD                | 2005-05-24 23:03:39 |
| FREAKY POCUS                 | 2005-05-24 22:54:33 |
| BLANKET BEVERLY              | 2005-05-24 22:53:30 |
+----------------------------+--------------------+
16044 rows in set (0.06 sec)
```

In this, the query is driven by the `rental` table, so all rows from it are matched against the `inventory`, and then against `film`. Since all rows in `rental` table by definition are based on the `inventory`, which is linked to `film`, we have no `NULL` values in the output. There can be no rental for a film that doesn't exist. We adjusted the query with `ORDER BY rental_date DESC` to show that we really didn't get any NULL values (these would have been last).

By now you can see that left joins are useful when we're sure that our *left* table has some important data, but not sure whether *right* table even has any. We want to get the rows from the left one with or without corresponding rows from the right one. Let's try to apply this to a query we wrote in "The GROUP BY Clause", which showed customers renting a lot from the same category. Here's the query:

```
mysql> SELECT email, name AS category_name,
COUNT(cat.category_id) AS cnt
    -> FROM customer cs INNER JOIN rental USING (customer_id)
    -> INNER JOIN inventory USING (inventory_id)
    -> INNER JOIN film_category USING (film_id)
    -> INNER JOIN category cat USING (category_id)
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC LIMIT 5;
```

```
+----------------------------------+---------------+-----+
| email                            | category_name | cnt |
+----------------------------------+---------------+-----+
| WESLEY.BULL@sakilacustomer.org   | Games         |   9 |
| ALMA.AUSTIN@sakilacustomer.org   | Animation     |   8 |
| KARL.SEAL@sakilacustomer.org     | Animation     |   8 |
| LYDIA.BURKE@sakilacustomer.org   | Documentary   |   8 |
| NATHAN.RUNYON@sakilacustomer.org | Animation     |   7 |
+----------------------------------+---------------+-----+
5 rows in set (0.06 sec)
```

What if we want now to see whether a customer we found this way is looking at anything but her favorite category. Well, it turns out that it's actually pretty difficult!

Let’s consider this task. We need to start with a `category` table, as that will have all the categories we may have for our films. We then need to start constructing a whole chain of `LEFT JOIN`. `category` left joins to `film_category` as we may have categories having no films. Then we left join an `inventory`, as some movies we know about may not be in our catalog. We then left join the `rental`, as we may not have rented some of the films in a category. Finally, we need to left join our `customer` table. Even though there can be no associated customer record without a rent, omitting the left join will cause MySQL to discard rows for categories that end up with no customer records.

Now, after this whole long explanation, can we finally go ahead and filter by email and get our data? No! Unfortunately, by adding a WHERE condition on the table which is not *left* in our left join relationship, we break the idea of this join. See what happens:

```
mysql> SELECT COUNT(*) FROM category;

+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT email, name AS category_name, COUNT(category_id) AS
cnt
    -> FROM category cat LEFT JOIN film_category USING
(category_id)
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> JOIN customer cs ON rental.customer_id = cs.customer_id
    -> WHERE cs.email = 'WESLEY.BULL@sakilacustomer.org'
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC;

+--------------------------------+---------------+-----+
| email                          | category_name | cnt |
+--------------------------------+---------------+-----+
| WESLEY.BULL@sakilacustomer.org | Games          |   9 |
| WESLEY.BULL@sakilacustomer.org | Foreign        |   6 |
...
| WESLEY.BULL@sakilacustomer.org | Comedy         |   1 |
| WESLEY.BULL@sakilacustomer.org | Sports         |   1 |
+--------------------------------+----------------+-----+
14 rows in set (0.00 sec)
```

We got 14 categories for our customer, while there are 16 in total. In fact, MySQL will optimize away all the left joins in this query, as it understands they are meaningless when put like this. There's no easy way to answer the question we have with just joins, and we'll need to add some more knowledge. We'll get back to this example in "Nested Queries in JOINs".

The query that we've written is still useful. While by default `sakila` does not have a film category that has no film rented, if we were to expand our database slightly, we can see the effectiveness of left joins.

```
mysql> INSERT INTO category(name) VALUES (Thriller);

Query OK, 1 row affected (0.01 sec)

mysql> SELECT cat.name, COUNT(rental_id) cnt
    -> FROM category cat LEFT JOIN film_category USING
(category_id)
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> LEFT JOIN customer cs ON rental.customer_id =
cs.customer_id
    -> GROUP BY 1
    -> ORDER BY 2 DESC;


+---------------+------+
| category_name | cnt  |
+---------------+------+
| Sports        | 1179 |
| Animation     | 1166 |
...
| Music         |  830 |
| Thriller      |    0 |
+---------------+------+
17 rows in set (0.07 sec)
```

If we were to use regular `INNER JOIN` (or just `JOIN`, its synonym), we'd not get information for "Thriller" category, and might've had different counts for other categories. As the category is our leftmost table, it drives the process of query, and every row from that table is present in the output.

We've shown you that it matters what comes before and after the `LEFT JOIN` statement. Whatever is on the left drives the process, hence the name "left join." If you really don't want to reorganize your query so it matches that template, you can use `RIGHT JOIN`. It's exactly the same, except whatever is on the right drives the process. Earlier we showed the importantce of order of the tables in the `LEFT JOIN` using two queries for film rental info. Let's rewrite the second of them (which showed incorrect data) using a right join:

```
mysql> SELECT title, rental_date
    -> FROM rental RIGHT JOIN inventory USING (inventory_id)
    -> RIGHT JOIN film USING (film_id)
    -> ORDER BY rental_date DESC;
```

```
...
| SUICIDES SILENCE             | NULL                 |
| TADPOLE PARK                 | NULL                 |
| TREASURE COMMAND             | NULL                 |
| VILLAIN DESPERATE            | NULL                 |
| VOLUME HOUSE                 | NULL                 |
| WAKE JAWS                    | NULL                 |
| WALLS ARTIST                 | NULL                 |
+------------------------------+----------------------+
16087 rows in set (0.06 sec)
```

We got the same amount of rows and see the `NULL` values are the same as the "correct" query gave us. The right join is useful sometimes because it allows you to write a query more naturally, expressing it in a way that's more intuitive. However, you won't often see it used, and we'd recommend avoiding it where possible.

Both the `LEFT JOIN` and `RIGHT JOIN` can use either the `USING` or `ON` clauses discussed for the `INNER JOIN` earlier in this chapter in "The Inner

Join”. You should use one or the other: without them, you’ll get the Cartesian product also discussed in “The Inner Join”.

There’s an extra `OUTER` keyword that you can optionally use in left and right joins, to make them read as `LEFT OUTER JOIN` and `RIGHT OUTER JOIN`. It’s just an alternative syntax that doesn’t do anything different, and you won’t often see it used. We stick to the basic versions in this book.

The Natural Join

We’re not big fans of the natural join that we’re about to describe in this section. It’s in here only for completeness and because you’ll see it used sometimes in SQL statements you’ll encounter. Our advice is to avoid using it where possible.

A natural join is, well, supposed to be magically natural. This means that you tell MySQL what tables you want to join, and it figures out how to do it and gives you an `INNER JOIN` result set. Here’s an example for the `actor_info` and `film_actor` tables:

```
mysql> SELECT first_name, last_name, film_id
    -> FROM actor_info NATURAL JOIN film_actor
    -> LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| PENELOPE   | GUINESS   |       1 |
| PENELOPE   | GUINESS   |      23 |
...
| PENELOPE   | GUINESS   |     980 |
| NICK       | WAHLBERG  |       3 |
+------------+-----------+---------+
20 rows in set (0.28 sec)
```

In reality, it’s not quite magical: all MySQL does is look for columns with the same names and, behind the scenes, adds these silently into an inner join with join conditions written into the where clause. So, the above query is actually translated into something like this:

```
mysql> SELECT first_name, last_name, film_id FROM
    -> actor_info JOIN film_actor
    -> WHERE (actor_info.actor_id = film_actor.actor_id)
    -> LIMIT 20;
```

If identifier columns don't share the same name, natural joins won't work. Also, more dangerously, if columns that do share the same names aren't identifiers, they'll get thrown into the behind-the-scenes `USING` clause anyway. You can very easily see this in the `sakila` database. In fact, that's why we resorted to showing the above example with `actor_info`, which isn't even a table: it's a view. Let's see what would have happened if we used regular `actor` and `film_actor` tables.

```
mysql> SELECT first_name, last_name, film_id FROM actor NATURAL
JOIN film_actor;

Empty set (0.01 sec)
```

But how? The problem is: `NATURAL JOIN` really does take **ALL** of the columns into consideration. With the `sakila` database, that's a huge roadblock, as every table has a `last_update` column. If you were to run an `EXPLAIN` statement on the above query, and then execute `SHOW WARNINGS`, you'd see that the resulting query is meaningless. See below:

```
mysql> SHOW WARNINGS\G

*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `sakila`.`customer`.`email` AS
`email`,
`sakila`.`rental`.`rental_date` AS `rental_date`
from `sakila`.`customer` join `sakila`.`rental`
where ((`sakila`.`rental`.`last_update` =
`sakila`.`customer`.`last_update`)
and (`sakila`.`rental`.`customer_id` =
`sakila`.`customer`.`customer_id`))
1 row in set (0.00 sec)
```

You’ll sometimes see the natural join mixed with left and right joins. The following are valid join syntaxes: `NATURAL LEFT JOIN`, `NATURAL LEFT OUTER JOIN`, `NATURAL RIGHT JOIN`, and `NATURAL RIGHT OUTER JOIN`. The former two are left joins without `ON` or `USING` clauses, and the latter two are right joins. Again, avoid writing them when you can, but you should understand what they mean if you see them used.

Constant expressions in joins

In all of the examples of the joins we gave you so far, we always used column identifiers to define the join condition. When you’re using the `USING` clause, that’s the only possible way to go. When you’re defininig the join conditions in `WHERE`, that’s also the only thing that will work. However, when you’re using the `ON` clause, you can actually add constant expressions.

Let’s consider an example with our films and actors again, listing all films for a particular actor.

```
mysql> SELECT first_name, last_name, title
    -> FROM actor JOIN film_actor USING (actor_id)
    -> JOIN film USING (film_id)
    -> WHERE actor_id = 11;

+------------+-----------+--------------------+
| first_name | last_name | title              |
+------------+-----------+--------------------+
| ZERO       | CAGE      | CANYON STOCK       |
| ZERO       | CAGE      | DANCES NONE        |
...
| ZERO       | CAGE      | WEST LION          |
| ZERO       | CAGE      | WORKER TARZAN      |
+------------+-----------+--------------------+
25 rows in set (0.00 sec)
```

What we can do, however, is move the `actor_id` clause to the join like this:

```
mysql> SELECT first_name, last_name, title
    -> FROM actor JOIN film_actor
    ->   ON actor.actor_id = film_actor.actor_id
    ->   AND actor.actor_id = 11
    -> JOIN film USING (film_id);

+------------+-----------+--------------------+
| first_name | last_name | title              |
+------------+-----------+--------------------+
| ZERO       | CAGE      | CANYON STOCK       |
| ZERO       | CAGE      | DANCES NONE        |
...
| ZERO       | CAGE      | WEST LION          |
| ZERO       | CAGE      | WORKER TARZAN      |
+------------+-----------+--------------------+
25 rows in set (0.00 sec)
```

Well, that's neat, of course, but why? Is this any more expressive than having the proper `WHERE` clause? The answer to both questions is that constant conditions in joins are evaluated and resolved differently than the conditions in `WHERE` clause are. It's easier to show this with an example, but the query above is a poor one. The impact of constant conditions in joins is best shown with a left join.

Remember this query from our left join chapter:

```
mysql> SELECT email, name AS category_name, COUNT(rental_id) AS
cnt
    -> FROM category cat LEFT JOIN film_category USING
(category_id)
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> LEFT JOIN customer cs USING (customer_id)
    -> WHERE cs.email = 'WESLEY.BULL@sakilacustomer.org'
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC;

+----------------------------------+---------------+-----+
| email                            | category_name | cnt |
+--------------------------------+---------------+-----+
| WESLEY.BULL@sakilacustomer.org | Games         |   9 |
| WESLEY.BULL@sakilacustomer.org | Foreign       |   6 |
...
| WESLEY.BULL@sakilacustomer.org | Comedy        |   1 |
| WESLEY.BULL@sakilacustomer.org | Sports        |   1 |
+--------------------------------+---------------+-----+
14 rows in set (0.01 sec)
```

If we go ahead and move the `cs.email` clause to the `LEFT JOIN customer cs` part, we'll see completely different results:

```
mysql> SELECT email, name AS category_name, COUNT(rental_id) AS
cnt
    -> FROM category cat LEFT JOIN film_category USING
(category_id)
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> LEFT JOIN customer cs ON rental.customer_id =
cs.customer_id
    -> AND cs.email = 'WESLEY.BULL@sakilacustomer.org'
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC;
```

```
+--------------------------------+-------------+------+
| email                          | name        | cnt  |
+--------------------------------+-------------+------+
| NULL                           | Sports      | 1178 |
| NULL                           | Animation   | 1164 |
...
| NULL                           | Travel      |  834 |
| NULL                           | Music       |  829 |
| WESLEY.BULL@sakilacustomer.org | Games       |    9 |
| WESLEY.BULL@sakilacustomer.org | Foreign     |    6 |
...
| WESLEY.BULL@sakilacustomer.org | Comedy      |    1 |
| NULL                           | Thriller    |    0 |
+--------------------------------+-------------+------+
31 rows in set (0.07 sec)
```

That's interesting! Instead of getting only Wesley's rental counts per category, we also get rental counts for everyone else broken down by category. That even includes our new and so far empty "Thriller" category. Let's try to understand what happens here.

The `WHERE` clause contents are applied logically after the joins were resolved and executed. We tell MySQL we only need rows from whatever we join where the `cs.email` column equals `'WESLEY.BULL@sakilacustomer.org'`. In reality, MySQL is smart enough to optimize this situation and will actually start the plan execution as if regular inner joins were used. When we have the `cs.email` condition within the `LEFT JOIN customer` clause, we tell MySQL that we want to add columns from `customer` table to our resultset so far (which includes category, inventory, and rental tables), but only when the certain value of `email` column is met. Since this is a `LEFT JOIN`, we get `NULL` in every column of `customer` in rows which didn't match.

It's important to mind this behavior.

Nested Queries

Nested queries — supported by MySQL since version 4.1 — are the most difficult to learn. However, they provide a powerful, useful, and concise way of expressing difficult information needs in short SQL statements. This section explains them, beginning with simple examples and leading to the more complex features of the `EXISTS` and `IN` statements. At the conclusion of this section, you'll have completed everything this book contains about querying data, and you should be comfortable understanding almost any SQL query you encounter.

Nested Query Basics

You know how to find names of all actors who played in a particular movie using an `INNER JOIN`:

```
mysql> SELECT first_name, last_name FROM
    -> actor JOIN film_actor USING (actor_id)
    -> JOIN film USING (film_id)
    -> WHERE title = 'ZHIVAGO CORE';
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| UMA        | WOOD      |
| NICK       | STALLONE  |
| GARY       | PENN      |
| SALMA      | NOLTE     |
| KENNETH    | HOFFMAN   |
| WILLIAM    | HACKMAN   |
+------------+-----------+
6 rows in set (0.00 sec)
```

But there’s another way, using a *nested query*:

```
mysql> SELECT first_name, last_name FROM
    -> actor JOIN film_actor USING (actor_id)
    -> WHERE film_id = (SELECT film_id FROM film
    -> WHERE title = 'ZHIVAGO CORE');
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| UMA        | WOOD      |
| NICK       | STALLONE  |
| GARY       | PENN      |
| SALMA      | NOLTE     |
| KENNETH    | HOFFMAN   |
| WILLIAM    | HACKMAN   |
+------------+-----------+
6 rows in set (0.00 sec)
```

It’s called a nested query because one query is inside another. The *inner query*, or *subquery* — the one that is nested — is written in parentheses, and you can see that it determines the `film_id` for the film with the title `ZHIVAGO CORE`. The parentheses are required for inner queries. The *outer query* is the one that’s listed first and isn’t parenthesized here: you can see that it finds the `first_name` and `last_name` of the the actors from a JOIN with `film_actor` with an `film_id` that matches the result of the subquery. So, overall, the inner query finds the `film_id`, and the outer query uses it to find actor’s names. Whenever nested queries are used, it’s possible to rewrite them as a few separate queries, and we’ll break down the example above, as it may help understand what is going on:

```
mysql> SELECT film_id FROM film WHERE title = 'ZHIVAGO CORE';

+---------+
| film_id |
+---------+
|     998 |
+---------+
1 row in set (0.03 sec)

mysql> SELECT first_name, last_name
    -> FROM actor JOIN film_actor USING (actor_id)
    -> WHERE film_id = 998;

+------------+-----------+
| first_name | last_name |
+------------+-----------+
| UMA        | WOOD      |
| NICK       | STALLONE  |
| GARY       | PENN      |
| SALMA      | NOLTE     |
| KENNETH    | HOFFMAN   |
| WILLIAM    | HACKMAN   |
+------------+-----------+
6 rows in set (0.00 sec)
```

So, which approach is preferable: nested or not nested? The answer isn’t easy. In terms of performance, the answer is usually not: nested queries are hard to optimize, and so they’re almost always slower to run than the unnested alternative.

Does this mean you should avoid nesting? The answer is no: sometimes it’s your only choice if you want to write a single query, and sometimes nested queries can answer information needs that can’t be easily solved otherwise. What’s more, nested queries are expressive. Once you’re comfortable with the idea, they’re a very readable way to show how a query is evaluated. In fact, many SQL designers advocate teaching nested queries before the join-based alternatives we’ve shown you in the past few chapters. We’ll show you examples of where nesting is readable and powerful throughout this section.

Before we begin to cover the keywords that can be used in nested queries, let’s visit an example that can’t be done easily in a single query — at least, not without MySQL’s non-standard, although ubiquitous, LIMIT clause! Suppose you want to know what movie a customer rented most recently. To do this, following the methods we’ve learned previously, you could find the date and time of the most recently stored row in the rental table:

```
mysql> SELECT MAX(rental_date) FROM rental
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org';
```

```
+---------------------+
| MAX(rental_date)    |
+---------------------+
| 2005-08-23 15:46:33 |
+---------------------+
1 row in set (0.01 sec)
```

You can then use the output as input to another query to find the film title:

```
mysql> SELECT title FROM film
    -> JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org'
    -> AND rental_date = '2005-08-23 15:46:33';
```

```
+-------------+
| title       |
+-------------+
| KARATE MOON |
+-------------+
1 row in set (0.00 sec)
```

In “User Variables”, later in this chapter, we’ll show how you can use variables to avoid having to type in the value in the second query.

With a nested query, you can do both steps in one shot:

```
mysql> SELECT title FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE rental_date = (SELECT MAX(rental_date) FROM rental
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org');
```

```
+-------------+
| title       |
+-------------+
| KARATE MOON |
+-------------+
1 row in set (0.01 sec)
```

You can see the nested query combines the two previous queries. Rather than using the constant date and time value discovered from a previous query, it executes the query directly as a subquery. This is the simplest type of nested query, one that returns a *scalar operand* — that is, a single value.

The previous example used the equality operator, the equals sign, `=`. You can use all types of comparison operators: `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), and `!=` (not equals) or `<>` (not equals).

The ANY, SOME, ALL, IN, and NOT IN Clauses

Before we start to show some more advanced features of nested queries, we need to switch to a new database in our examples. Unfortunately, our `sakila` database is a little too well normalized to effectively demonstrate the full power of nested querying. So, let's add a new database to give us something to play with.

The database we'll install is the `employees` sample database. You can find instructions for installation in the MySQL documentation *https://dev.mysql.com/doc/employee/en/employees-installation.html* or in the database's github repo over at *https://github.com/datacharmer/test_db* Either clone the repository using `git`, or download the latest release (1.0.7 at the time of writing *https://github.com/datacharmer/test_db/releases/tag/v1.0.7)*. Once you have the necessary files ready, we need to run two commands.

First command creates necessary structures and loads the data.

```
$ mysql -uroot -p < employees.sql
INFO
CREATING DATABASE STRUCTURE
INFO
storage engine: InnoDB
INFO
LOADING departments
INFO
LOADING employees
INFO
LOADING dept_emp
INFO
LOADING dept_manager
INFO
LOADING titles
INFO
LOADING salaries
data_load_time_diff
00:00:28
```

Second command verifies the installation.

```
$ mysql -uroot -p < test_employees_md5.sql
INFO
TESTING INSTALLATION
table_name      expected_records        expected_crc
departments     9       d1af5e170d2d1591d776d5638d71fc5f
dept_emp        331603  ccf6fe516f990bdaa49713fc478701b7
dept_manager    24      8720e2f0853ac9096b689c14664f847e
employees       300024  4ec56ab5ba37218d187cf6ab09ce1aa1
salaries        2844047 fd220654e95aea1b169624ffe3fca934
titles  443308  bfa016c472df68e70a03facafa1bc0a8
table_name      found_records           found_crc
departments     9       d1af5e170d2d1591d776d5638d71fc5f
dept_emp        331603  ccf6fe516f990bdaa49713fc478701b7
dept_manager    24      8720e2f0853ac9096b689c14664f847e
employees       300024  4ec56ab5ba37218d187cf6ab09ce1aa1
salaries        2844047 fd220654e95aea1b169624ffe3fca934
titles  443308  bfa016c472df68e70a03facafa1bc0a8
table_name      records_match   crc_match
```

```
departments     OK      ok
dept_emp        OK      ok
dept_manager    OK      ok
employees       OK      ok
salaries        OK      ok
titles  OK      ok
computation_time
00:00:25
summary result
CRC     OK
count   OK
```

Once this is done, you can proceed to work through the examples we'll be providing next.

To connect to the new database, either run `mysql` from the command line like this (or specify `employees` as a target for your MySQL client of choice):

```
$ mysql employees
```

Or execute the following in a `mysql` prompt to change the default database:

```
mysql> use employees
```

Now you're ready to move forward.

Using ANY and IN

Now that you've created the sample tables, you can try an example using ANY. Suppose you're looking to find assistant engineers who've been working longer than the least experienced manager. You can express this information need as follows:

```
mysql> SELECT emp_no, first_name, last_name, hire_date
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Assistant Engineer'
    -> AND hire_date < ANY (SELECT hire_date FROM
    -> employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager');
```

```
+--------+----------------+-----------------+------------+
| emp_no | first_name     | last_name       | hire_date  |
+--------+----------------+-----------------+------------+
|  10009 | Sumant         | Peac            | 1985-02-18 |
|  10066 | Kwee           | Schusler        | 1986-02-26 |
...
...
| 499958 | Srinidhi       | Theuretzbacher  | 1989-12-17 |
| 499974 | Shuichi        | Piazza          | 1989-09-16 |
+--------+----------------+-----------------+------------+
10747 rows in set (0.20 sec)
```

Turns out, there are a lot of people falling under this criteria! The subquery finds the dates on which managers were hired:

```
mysql> SELECT hire_date FROM
    -> employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager';
```

```
+------------+
| hire_date  |
+------------+
| 1985-01-01 |
| 1986-04-12 |
...
| 1991-08-17 |
| 1989-07-10 |
+------------+
24 rows in set (0.10 sec)
```

The outer query goes through each employee with the title `Associate Engineer`, returning the engineer if their hire date is lower (older) than any of the values in the set returned by the subquery. So, for example, Sumant Peac is output because 1985-02-18 is older than at least one value in the set. For example, the second hire date returned for managers is 1986-04-12. The `ANY` keyword means just that: it's true if the column or expression preceding it is true for *any* of the values in the set returned by the subquery. Used in this way, `ANY` has the alias `SOME`, which was included so that some queries can be read more clearly as English expressions; it doesn't do anything different and you'll rarely see it used.

The ANY keyword gives you more power in expressing nested queries. Indeed, the previous query is the first nested query in this section with a *column subquery* — that is, the results returned by the subquery are one or more values from a column, instead of a single scalar value as in the previous section. With this, you can now compare a column value from an outer query to a set of values returned from a subquery.

Consider another example using ANY. Suppose you want to know the managers who also have some other title. You can do this with the following nested query:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no = ANY (SELECT emp_no FROM employees
    -> JOIN titles USING (emp_no) WHERE
    -> title <> 'Manager');
```

```
+--------+------------+-------------+
| emp_no | first_name | last_name   |
+--------+------------+-------------+
| 110022 | Margareta  | Markovitch  |
| 110039 | Vishwani   | Minakawa    |
...
| 111877 | Xiaobin    | Spinelli    |
| 111939 | Yuchang    | Weedman     |
+--------+------------+-------------+
24 rows in set (0.11 sec)
```

The = ANY causes the outer query to return a manager when the emp_no is equal to any of the engineer employee numbers returned by the subquery. The = ANY keyphrase has the alias IN, which you'll see commonly used in nested queries. Using IN, the previous example can be rewritten:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no IN (SELECT emp_no FROM employees
    -> JOIN titles USING (emp_no) WHERE
    -> title <> 'Manager');
```

```
+--------+------------+--------------+
| emp_no | first_name | last_name    |
+--------+------------+--------------+
| 110022 | Margareta  | Markovitch   |
| 110039 | Vishwani   | Minakawa     |
...
| 111877 | Xiaobin    | Spinelli     |
| 111939 | Yuchang    | Weedman      |
+--------+------------+--------------+
24 rows in set (0.11 sec)
```

Of course, for this particular example, you could also have used a join query. Note that we have to use `DISTINCT` here, because otherwise we get 30 rows returned. Some people hold more than one non-engineer title.

```
mysql> SELECT DISTINCT emp_no, first_name, last_name
    -> FROM employees JOIN titles mgr USING (emp_no)
    -> JOIN titles nonmgr USING (emp_no)
    -> WHERE mgr.title = 'Manager'
    -> AND nonmgr.title <> 'Manager';
```

```
+--------+------------+--------------+
| emp_no | first_name | last_name    |
+--------+------------+--------------+
| 110022 | Margareta  | Markovitch   |
| 110039 | Vishwani   | Minakawa     |
...
| 111877 | Xiaobin    | Spinelli     |
| 111939 | Yuchang    | Weedman      |
+--------+------------+--------------+
24 rows in set (0.11 sec)
```

Again, nested queries are expressive but typically slow in MySQL, so use a join where you can.

Using ALL

Suppose you want to find assistant engineers who are more experienced than all of the managers — that is, more experienced than the most experienced manager. You can do this with the `ALL` keyword in place of `ANY`:

```
mysql> SELECT emp_no, first_name, last_name, hire_date
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Assistant Engineer'
    -> AND hire_date < ALL (SELECT hire_date FROM
    -> employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager');

Empty set (0.18 sec)
```

You can see that there are no answers. We can inspect the data further to check what is the oldest hire date of a manager, and of an assistant engineer:

```
mysql> (SELECT 'Assistant Engineer' AS title,
    -> MIN(hire_date) AS mhd FROM employees
    -> JOIN titles USING (emp_no)
    -> WHERE title = 'Assistant Engineer')
    -> UNION
    -> (SELECT 'Manager' title, MIN(hire_date) mhd FROM employees
    -> JOIN titles USING (emp_no)
    -> WHERE title = 'Manager');

+--------------------+------------+
| title              | mhd        |
+--------------------+------------+
| Assistant Engineer | 1985-02-01 |
| Manager            | 1985-01-01 |
+--------------------+------------+
2 rows in set (0.26 sec)
```

Looking at the data, we see that the first manager was hired on January first, 1985, and the first assistant engineer only on February first, same year. While the `ANY` keyword returns values that satisfy at least one condition (Boolean OR), the `ALL` keyword returns values when all the conditions are satisfied (Boolean AND).

We can use the alias `NOT IN` in place of `<> ANY` or `!= ANY`. Let’s find all the managers who aren’t senior staff:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager' AND emp_no NOT IN
    -> (SELECT emp_no FROM titles
    -> WHERE title = 'Senior Staff');

+--------+------------+-------------+
| emp_no | first_name | last_name   |
+--------+------------+-------------+
| 110183 | Shirish    | Ossenbruggen |
| 110303 | Krassimir  | Wegerle     |
...
| 111400 | Arie       | Staelin     |
| 111692 | Tonny      | Butterworth |
+--------+------------+-------------+
15 rows in set (0.09 sec)
```

As an exercise, try writing the above query using the ANY syntax and as a join query.

The ALL keyword has a few tricks and traps:

- If it's false for any value, it's false. Suppose that table a contains a row with the value 14. Suppose table b contains the values 16, 1, and NULL. If you check whether the value in a is greater than ALL values in b, you'll get false, since 14 isn't greater than 16. It doesn't matter that the other values are 1 and NULL.
- If it isn't false for any value, it isn't true unless it's true for all values. Suppose that table a again contains 14, and suppose b contains 1 and NULL. If you check whether the value in a is greater than ALL values in b, you'll get UNKNOWN (neither true or false) because it can't be determined whether NULL is greater than or less than 14.
- If the table in the subquery is empty, the result is always true. Hence, if a contains 14 and b is empty, you'll get true when you check if the value in a is greater than ALL values in b.

When using the ALL keyword, be very careful with tables that can have NULL values in columns; consider disallowing NULL values in such cases. Also, be careful with empty tables.

Writing row subqueries

In the previous examples, the subquery returned a single, scalar value (such as an `actor_id`) or a set of values from one column (such as all of the `emp_no` values). This section describes another type of subquery, the row subquery that works with multiple columns from multiple rows.

Suppose you’re interested in whether a manager had another position within the same calendar year. To answer this need, you must match both employee number and the title assignment date, or, more precisely, year. You can write this as a join:

```
mysql> SELECT mgr.emp_no, YEAR(mgr.from_date) AS fd
    -> FROM titles AS mgr, titles AS other
    -> WHERE mgr.emp_no = other.emp_no
    -> AND mgr.title = 'Manager'
    -> AND mgr.title <> other.title
    -> AND YEAR(mgr.from_date) = YEAR(other.from_date);

+--------+------+
| emp_no | fd   |
+--------+------+
| 110765 | 1989 |
| 111784 | 1988 |
+--------+------+
2 rows in set (0.11 sec)
```

But you can also write it as a nested query:

```
mysql> SELECT emp_no, YEAR(from_date) AS fd
    -> FROM titles WHERE title = 'Manager' AND
    -> (emp_no, YEAR(from_date)) IN
    -> (SELECT emp_no, YEAR(from_date)
    -> FROM titles WHERE title <> 'Manager');

+--------+------+
| emp_no | fd   |
+--------+------+
| 110765 | 1989 |
| 111784 | 1988 |
+--------+------+
2 rows in set (0.12 sec)
```

You can see there's a different syntax being used in this nested query: a list of two column names in parentheses follows the `WHERE` statement, and the inner query returns two columns. We'll explain this syntax next.

The row subquery syntax allows you to compare multiple values per row. The expression `(emp_no, YEAR(from_date))` means two values per row are compared to the output of the subquery. You can see following the `IN` keyword that the subquery returns two values, `emp_no` and `YEAR(from_date)`. So, the fragment:

```
(emp_no, YEAR(from_date)) IN (SELECT emp_no, YEAR(from_date)
FROM titles WHERE title <> 'Manager')
```

matches manager numbers and starting years to non managers numbers and starting years, and returns a true value when a match is found. The result is that if a matching pair is found, the overall query outputs a result. This is a typical row subquery: it finds rows that exist in two tables.

To explain the syntax further, let's consider another example. Suppose you want to see if a particular employee is a senior staff member. You can do this with the following query:

```
mysql> SELECT first_name, last_name
    -> FROM employees, titles
    -> WHERE (employees.emp_no, first_name, last_name, title) =
    -> (titles.emp_no, 'Marjo', 'Giarratana', 'Senior Staff');

+------------+------------+
| first_name | last_name  |
+------------+------------+
| Marjo      | Giarratana |
+------------+------------+
1 row in set (0.09 sec)
```

It's not a nested query, but it shows you how the new row subquery syntax works. You can see that the query matches the list of columns before the equals sign, `(employees.emp_no, first_name, last_name, title)`, to the list of columns and values after the equals sign, `(titles.emp_no, 'Marjo', 'Giarratana', 'Senior Staff')`. So, when the `emp_no` values match, the employee’s full name is `Marjo Giarratana`, and the title is *Senior Staff*, we get output from the query. We don’t recommend writing queries like this—use a regular `WHERE` clause with multiple `AND` conditions instead—but it does illustrate exactly what’s going on. For an exercise, try writing this query using a join.

Row subqueries require that the number, order, and type of values in the columns match. So, for example, our previous example matches an `INT` to an `INT`, and two character strings to two character strings.

The EXISTS and NOT EXISTS Clauses

You’ve now seen three types of subquery: scalar subqueries, column subqueries, and row subqueries. In this section, you’ll learn about a fourth type, the *correlated subquery*, where a table used in the outer query is referenced in the subquery. Correlated subqueries are often used with the `IN` statement we’ve already discussed, and almost always used with the `EXISTS` and `NOT EXISTS` clauses that are the focus of this section.

EXISTS and NOT EXISTS basics

Before we start on our discussion of correlated subqueries, let’s investigate what the `EXISTS` clause does. We’ll need a simple but strange example to introduce the clause, since we’re not discussing correlated subqueries just yet. So, here goes: suppose you want to find a count of all films in the database, but only if the database is active (which you’ve defined to mean only if at least one movie from any branch has been rented). Here’s the query that does it. Do not forget to connect to `sakila` database again (hint: use the `use <db>` command).

```
mysql> SELECT COUNT(*) FROM film
    -> WHERE EXISTS (SELECT * FROM rental);

+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.01 sec)
```

The subquery returns all rows from the `rental` table. However, what's important is that it returns at least one row; it doesn't matter what's in the row, how many rows there are, or whether the row contains only `NULL` values. So, you can think of the subquery as being true or false, and in this case it's true because it produces some output. When the subquery is true, the outer query that uses the `EXISTS` clause returns a row. The overall result is that all rows in the `film` table are counted because, for each one, the subquery is true.

Let's try a query where the subquery isn't true. Again, let's contrive a query: this time, we'll output the names of all films in the database, but only if a particular film exists. Here's the query:

```
mysql> SELECT title FROM film
    -> WHERE EXISTS (SELECT * FROM film
    -> WHERE title = 'IS THIS A MOVIE?');

Empty set (0.00 sec)
```

Since the subquery isn't true — no rows are returned because `IS THIS A MOVIE?` isn't in our database — no results are returned by the outer query.

The `NOT EXISTS` clause does the opposite. Imagine you want a list of all actors if you don't have a particular movie in the database. Here it is:

```
mysql> SELECT * FROM actor WHERE NOT EXISTS
    -> (SELECT * FROM film WHERE title = 'ZHIVAGO CORE');

Empty set (0.00 sec)
```

This time, the inner query is true but the `NOT EXISTS` clause negates it to give false. Since it's false, the outer query doesn't produce results.

You'll notice that the subquery begins with `SELECT * FROM film`. It doesn't actually matter what you select in an inner query when you're using the `EXISTS` clause, since it's not used by the outer query anyway. You can select one column, everything, or even a constant (as in `SELECT 'cat' from film`), and it'll have the same effect. Traditionally, though, you'll see most SQL authors write `SELECT *` by convention.

Correlated subqueries

So far, it's difficult to imagine what you'd do with the `EXISTS` or `NOT EXISTS` clauses. This section shows you how they're really used, illustrating the most advanced type of nested query that you'll typically see in action.

Let's think about the realistic information you might want to answer from the `sakila` database. Suppose you want a list of all employees who've rented something from our company, or are just a customer. You can do this easily with a join query, which we recommend you try to think about before you continue. You can also do it with the following nested query that uses a *correlated subquery*:

```
mysql> SELECT first_name, last_name FROM staff
    -> WHERE EXISTS (SELECT * FROM customer
    -> WHERE customer.first_name = staff.first_name
    -> AND customer.last_name = staff.last_name);

Empty set (0.01 sec)
```

There's no output because nobody from the staff is also a customer (or that's forbidden, but we'll bend rules). Let's add a customer with the same details as one of the staff members.

```
mysql> INSERT INTO customer(store_id, first_name, last_name,
    -> email, address_id, create_date)
    -> VALUES (1, 'Mike', 'Hillyer',
    -> 'Mike.Hillyer@sakilastaff.com', 3, NOW());

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

Now the query:

```
mysql> SELECT first_name, last_name FROM staff
    -> WHERE EXISTS (SELECT * FROM customer
    -> WHERE customer.first_name = staff.first_name
    -> AND customer.last_name = staff.last_name);

+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Mike       | Hillyer   |
+------------+-----------+
1 row in set (0.00 sec)
```

So, the query works; now, we just need to understand how!

Let's examine the subquery in our previous example. You can see that it lists only the `customer` table in the `FROM` clause, but it uses a column from the `staff` table in the `WHERE` clause. If you run it in isolation, you'll see this isn't allowed:

```
mysql> SELECT * FROM customer WHERE customer.first_name =
staff.first_name;

ERROR 1054 (42S22): Unknown column 'staff.first_name' in 'where
clause'
```

However, it's legal when executed as a subquery because tables listed in the outer query are allowed to be accessed in the subquery. So, in this example, the current value of `staff.first_name` and `staff.last_name` in the outer query is supplied to the subquery as a constant, scalar value and compared to the customer's first and last names. If the customer's name matches the staff member's name, the subquery is true, and so the outer query outputs a row. Consider two cases that illustrate this more clearly:

- When the `first_name` and `last_name` being processed by the outer query are `Jon` and `Stephens`, the subquery is false because `SELECT * FROM customer WHERE first_name = 'Jon' and last_name = 'Stephens';` doesn't return any rows, and so the staff row for Jon Stephens isn't output as an answer.

- When the `first_name` and `last_name` being processed by the outer query are `Mike` and `Hillyer`, the subquery is true because `SELECT * FROM customer WHERE first_name = 'Mike' and last_name = 'Hillyer';` returns at least one row. Overall, the staff row for Mike is returned.

Can you see the power of correlated subqueries? You can use values from the outer query in the inner query to evaluate complex information needs.

We'll now explore another example using `EXISTS`. Let's try to find a count of all films from whom we own at least two copies. To do this with `EXISTS`, we need to think through what the inner and outer queries should do. The inner query should produce a result only when the condition we're checking is true; in this case, it should produce output when the are at least two rows in the inventory for the same film. The outer query should increment the counter whenever the inner query is true. Here's the query:

```
mysql> SELECT COUNT(*) FROM film WHERE EXISTS
    -> (SELECT film_id FROM inventory
    -> WHERE inventory.film_id = film.film_id
    -> GROUP BY film_id HAVING COUNT(*) >= 2);

+----------+
| COUNT(*) |
+----------+
|      958 |
+----------+
1 row in set (0.00 sec)
```

This is yet another query where nesting isn't necessary and a join would suffice, but let's stick with this version for the purpose of explanation. Have a look at the inner query: you can see that the `WHERE` clause ensures that films match by the unique film_id, and only matching rows — for the current film — are considered by the subquery. The `GROUP BY` clause clusters the rows for that film, but only if there are at least two entries in the inventory. Therefore, the inner query only produces output when there are at least two rows for the current film in our inventory. The outer query is straightforward: it can be thought of as incrementing a counter when the subquery produces output.

Here's one more example before we move on and discuss other issues. This example will be in the `employees` database, so switch your client. We've already shown you a query that uses `IN` and finds managers who also had some other position:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no IN (SELECT emp_no FROM employees
    -> JOIN titles USING (emp_no) WHERE
    -> title <> 'Manager');

+--------+------------+-------------+
| emp_no | first_name | last_name   |
+--------+------------+-------------+
| 110022 | Margareta  | Markovitch  |
| 110039 | Vishwani   | Minakawa    |
...
| 111877 | Xiaobin    | Spinelli    |
| 111939 | Yuchang    | Weedman     |
+--------+------------+-------------+
24 rows in set (0.11 sec)
```

Let's rewrite the query to use `EXISTS`. First, think about the subquery: it should produce output when there's a title record for an employee with the same name as a manager.

Second, think about the outer query: it should return the employee's name when the inner query produces output. Here's the rewritten query:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND EXISTS (SELECT emp_no FROM titles
    -> WHERE titles.emp_no = employees.emp_no
    -> AND title <> 'Manager');
```

```
+--------+------------+--------------+
| emp_no | first_name | last_name    |
+--------+------------+--------------+
| 110022 | Margareta  | Markovitch   |
| 110039 | Vishwani   | Minakawa     |
...
| 111877 | Xiaobin    | Spinelli     |
| 111939 | Yuchang    | Weedman      |
+--------+------------+--------------+
24 rows in set (0.09 sec)
```

Again, you can see that the subquery references the `emp_no` column, which comes from the outer query.

Correlated subqueries can be used with any nested query type. Here's the previous `IN` query rewritten with an outer reference:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no IN (SELECT emp_no FROM titles
    -> WHERE titles.emp_no = employees.emp_no
    -> AND title <> 'Manager');
```

```
+--------+------------+--------------+
| emp_no | first_name | last_name    |
+--------+------------+--------------+
| 110022 | Margareta  | Markovitch   |
| 110039 | Vishwani   | Minakawa     |
...
| 111877 | Xiaobin    | Spinelli     |
| 111939 | Yuchang    | Weedman      |
+--------+------------+--------------+
24 rows in set (0.09 sec)
```

The query is more convoluted than it needs to be, but it illustrates the idea. You can see that the `emp_no` in the subquery references the `employees` table from the outer query.

If the query would return a single row, it could also be rewritten to use an equals instead of `IN`:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no = (SELECT emp_no FROM titles
    -> WHERE titles.emp_no = employees.emp_no
    -> AND title <> 'Manager');

ERROR 1242 (21000): Subquery returns more than 1 row
```

This doesn't work because the subquery returns more than one scalar value. Let's narrow down.

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no = (SELECT emp_no FROM titles
    -> WHERE titles.emp_no = employees.emp_no
    -> AND title = 'Senior Engineer');

+--------+------------+-----------+
| emp_no | first_name | last_name |
+--------+------------+-----------+
| 110344 | Rosine     | Cools     |
| 110420 | Oscar      | Ghazalie  |
| 110800 | Sanjoy     | Quadeer   |
+--------+------------+-----------+
3 rows in set (0.10 sec)
```

Works now — there's only one manager and senior engineer title with each name — and so the column subquery operator `IN` isn't necessary. Of course, if titles are duplicated (person switches between positions back and forth), you'd need to use `IN`, `ANY`, or `ALL` instead.

Nested Queries in the FROM Clause

The techniques we've shown all use nested queries in the `WHERE` clause. This section shows you how they can alternatively be used in the `FROM` clause. This is useful when you want to manipulate the source of the data you're using in a query.

In the employees database, the salaries table stores the annual wage alongside the employee id. If you want to find the monthly rate, for example, you can do some math in the query; one option in this class is to do it with a subquery:

```
mysql> SELECT emp_no, monthly_salary FROM
    -> (SELECT emp_no, salary/12 AS monthly_salary FROM salaries)
AS ms
    -> LIMIT 5;

+--------+----------------+
| emp_no | monthly_salary |
+--------+----------------+
|  10001 |      5009.7500 |
|  10001 |      5175.1667 |
|  10001 |      5506.1667 |
|  10001 |      5549.6667 |
|  10001 |      5580.0833 |
+--------+----------------+
5 rows in set (0.00 sec)
```

Focus on what follows the FROM clause: the subquery uses the salaries table and returns two columns. The first column is the emp_no; the second column is aliased as monthly_salary, and is the salary value divided by 12. The outer query is straightforward: it just returns the emp_no and the monthly_salary value created through the subquery. Note that we've added the table alias AS ms for the subquery. When we use a subquery as a table, that is, we use a SELECT FROM operation on it — this "derived table" must have an alias — even if we don't use the alias in our query. MySQL complains if we omit the alias:

```
mysql> SELECT emp_no, monthly_salary FROM
    -> (SELECT emp_no, salary/12 AS monthly_salary FROM salaries)
    -> LIMIT 5;

ERROR 1248 (42000): Every derived table must have its own alias
```

Here's another example, now in the sakila database. We'll find out the average sum a film brings us through rentals, or the average gross, as we'll call it. Let’s begin by thinking through the subquery. It should return the sum of payments that we have for each film. Then, the outer query should average the values to give the answer. Here’s the query:

```
mysql> SELECT AVG(gross) FROM
    -> (SELECT SUM(amount) AS gross
    -> FROM payment JOIN rental USING (rental_id)
    -> JOIN inventory USING (inventory_id)
    -> JOIN film USING (film_id)
    -> GROUP BY film_id) AS gross_amount;
```

```
+------------+
| AVG(gross) |
+------------+
|  70.361754 |
+------------+
1 row in set (0.05 sec)
```

You can see that the inner query joins together `payment`, `rental`, `inventory`, and `film`, and groups the sales together by film so you can get a sum for each film. If you run it in isolation, here’s what happens:

```
mysql> SELECT SUM(amount) AS gross
    -> FROM payment JOIN rental USING (rental_id)
    -> JOIN inventory USING (inventory_id)
    -> JOIN film USING (film_id)
    -> GROUP BY film_id;
```

```
+--------+
| gross  |
+--------+
|  36.77 |
|  52.93 |
|  37.88 |
...
|  14.91 |
|  73.83 |
| 214.69 |
+--------+
958 rows in set (0.08 sec)
```

Now, the outer query takes these sums — which are aliased as `gross` — and averages them to give the final result. This query is the typical way that you apply two aggregate functions to one set of data. You can't apply aggregate functions in cascade, as in `AVG(SUM(amount))`; it won't work:

```
mysql> SELECT AVG(SUM(amount)) AS avg_gross
    -> FROM payment JOIN rental USING (rental_id)
    -> JOIN inventory USING (inventory_id)
    -> JOIN film USING (film_id) GROUP BY film_id;

ERROR 1111 (HY000): Invalid use of group function
```

With subqueries in `FROM` clauses, you can return a scalar value, a set of column values, more than one row, or even a whole table. However, you can't use correlated subqueries, meaning that you can't reference tables or columns from tables that aren't explicitly listed in the subquery. Note also that you must alias the whole subquery using the `AS` keyword and give it a name, even if you don't use that name anywhere in the query.

Nested Queries in JOINs

The last use of nested queries we'll show, but not the least useful, is using them in joins. In this use case, the results of the subquery basically form a new table and can be used in any of the `JOIN` types we have discussed.

For the example of this, let's go back to the query which listed the number of films from each of the categories a particular customer has rented. Remember, we had an issue writing that query using just joins: we didn't get a 0 count for categories from which our customer didn't rent. This was the query:

```
mysql> SELECT cat.name AS category_name, COUNT(cat.category_id)
AS cnt
    -> FROM category AS cat LEFT JOIN film_category USING
(category_id)
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> JOIN customer AS cs ON rental.customer_id = cs.customer_id
    -> WHERE cs.email = 'WESLEY.BULL@sakilacustomer.org'
    -> GROUP BY category_name ORDER BY cnt DESC;
```

```
+-------------+-----+
| name        | cnt |
+-------------+-----+
| Games       |   9 |
| Foreign     |   6 |
...
...
| Comedy      |   1 |
| Sports      |   1 |
+-------------+-----+
14 rows in set (0.00 sec)
```

Now that we know about subqueries and joins, and the subqueries can be used in joins, we can easily finish the task. This is our new query:

```
mysql> SELECT cat.name AS category_name, cnt
    -> FROM category AS cat
    -> LEFT JOIN (SELECT cat.name, COUNT(cat.category_id) AS cnt
    ->    FROM category AS cat
    ->    LEFT JOIN film_category USING (category_id)
    ->    LEFT JOIN inventory USING (film_id)
    ->    LEFT JOIN rental USING (inventory_id)
    ->    JOIN customer cs ON rental.customer_id = cs.customer_id
    ->    WHERE cs.email = 'WESLEY.BULL@sakilacustomer.org'
    ->    GROUP BY cat.name) customer_cat USING (name)
    -> ORDER BY cnt DESC;
```

```
+-------------+------+
| name        | cnt  |
+-------------+------+
| Games       |    9 |
| Foreign     |    6 |
...
| Children    |    1 |
| Sports      |    1 |
| Sci-Fi      | NULL |
| Action      | NULL |
| Thriller    | NULL |
+-------------+------+
17 rows in set (0.01 sec)
```

Finally, we get all the categories displayed, and we get NULL values for those where no rentals were made. Let's review what's going on in our new query. The subquery, which we aliased `customer_cat`, is our previous query sans the `ORDER BY` clause. Thus, we know what it will return: 14 rows for categories Wesley rented something and the number of rentals. What we do next is we use `LEFT JOIN` to concatenate that information to the full list of categories from the `category` table. The `category` table is driving the join, so it'll have every row selected. We join the subquery using the `name` column that matches between the subquery's output and the `category` table's column.

The technique we showed here is a very powerful one, however, as always with subqueries, it comes at a cost. MySQL cannot optimize the whole query as efficiently, when a subquery is present in the join clause.

User Variables

Often you'll want to save values that are returned from queries. You might want to do this so that you can easily use a value in a later query. You might also simply want to save a result for later display. In both cases, user variables solve the problem: they allow you to store a result and use it later.

Let's illustrate user variables with a simple example. The following query finds the title of a film and saves the result in a user variable:

```
mysql> SELECT @film:=title FROM film WHERE film_id = 1;

+------------------+
| @film:=title     |
+------------------+
| ACADEMY DINOSAUR |
+------------------+
1 row in set, 1 warning (0.00 sec)
```

The user variable is named `film`, and it's denoted as a user variable by the `@` character that precedes it. The value is assigned using the `:=` operator.

You can print out the contents of the user variable with the following very short query:

```
mysql> SELECT @film;

+------------------+
| @film            |
+------------------+
| ACADEMY DINOSAUR |
+------------------+
1 row in set (0.00 sec)
```

You could notice a warning, what was that about?

```
mysql> SET @film := (SELECT title FROM film WHERE film_id = 1);
mysql> SHOW WARNINGS\G

*************************** 1. row ***************************
  Level: Warning
   Code: 1287
Message: Setting user variables within expressions is deprecated
and will be removed in a future release. Consider alternatives:
'SET variable=expression, ...', or
'SELECT expression(s) INTO variables(s)'.
1 row in set (0.00 sec)
```

Let’s cover the two alternatives proposed. First, we can still execute a nested query within a SET statement:

```
mysql> SET @film := (SELECT title FROM film WHERE film_id = 1);

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @film;

+------------------+
| @film            |
+------------------+
| ACADEMY DINOSAUR |
+------------------+
1 row in set (0.00 sec)
```

Second, we can do the `SELECT INTO` statement:

```
mysql> SELECT title INTO @film FROM film WHERE film_id = 1;

Query OK, 1 row affected (0.00 sec)

mysql> SELECT @film;

+-------------------+
| @film             |
+-------------------+
| ACADEMY DINOSAUR |
+-------------------+
1 row in set (0.00 sec)
```

You can explicitly set a variable using the `SET` statement without a `SELECT`. Suppose you want to initialize a counter to 0:

```
mysql> SET @counter := 0;

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

The `:=` is optional, and you can write `=` instead, and mix them up. You should separate several assignments with a comma, or put each in a statement of its own:

```
mysql> SET @counter = 0, @age := 23;

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

The alternative syntax for `SET` is `SELECT INTO`. You can initialize a single variable:

```
mysql> SELECT 0 INTO @counter;

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

Or multiple at once:

```
mysql> SELECT 0, 23 INTO @counter, @age;

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

The most common use of user variables is to save a result and use it later. You'll recall the following example from earlier in the chapter, which we used to motivate nested queries (which are certainly a better solution for this problem). We want to find the name of the film that was rented most recently:

```
mysql> SELECT MAX(rental_date) FROM rental
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org';

+---------------------+
| MAX(rental_date)    |
+---------------------+
| 2005-08-23 15:46:33 |
+---------------------+
1 row in set (0.01 sec)

mysql> SELECT title FROM film
    -> JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org'
    -> AND rental_date = '2005-08-23 15:46:33';

+-------------+
| title       |
+-------------+
| KARATE MOON |
+-------------+
1 row in set (0.00 sec)
```

You can use a user variable to save the result for input into the following query. Here's the same query pair rewritten using this approach:

```
mysql> SELECT MAX(rental_date) INTO @recent FROM rental
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org';
```

```
1 row in set (0.01 sec)

mysql> SELECT title FROM film
    -> JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org'
    -> AND rental_date = @recent;

+-------------+
| title       |
+-------------+
| KARATE MOON |
+-------------+
1 row in set (0.00 sec)
```

This can save you cutting and pasting, and it certainly helps you avoid typing errors.

Here are some guidelines on using user variables:

- User variables are unique to a connection: variables that you create can't be seen by anyone else, and two different connections can have two different variables with the same name.
- The variable names can be alphanumeric strings and can also include the period (`.`), underscore (_), and dollar ($) characters.
- Variable names are case-sensitive in MySQL versions earlier than version 5, and case-insensitive from version 5 onward.
- Any variable that isn't initialized has the value `NULL`; you can also manually set a variable to be `NULL`.
- Variables are destroyed when a connection closes.
- You should avoid trying to both assign a value to a variable and use the variable as part of a `SELECT` query. Two reasons for this are that the new value may not be available for use immediately in the same statement, and a variable's type is set when it's first assigned

in a query; trying to use it later as a different type in the same SQL statement can lead to unexpected results.

Let's look at the first issue in more detail using the new variable `@fid`. Since we haven't used this variable before, it's empty. Now, let's show the `film_id` for movies that have an entry in the `inventory` table. Instead of showing it directly, we'll assign the `film_id` to the `@fid` variable. Our query will show the variable twice: once before the assignment operation, once as part of the assignment operation, and once afterwards:

```
mysql> SELECT @fid, @fid:=film.film_id, @fid FROM film,
inventory
    -> WHERE inventory.film_id = @fid;

Empty set, 1 warning (0.16 sec)
```

This returns nothing apart from a deprecation warning; since there's nothing in the variable to start with, the `WHERE` clause tries to look for empty `inventory.film_id` values. If we modify the query to use `film.film_id` as part of the `WHERE` clause, things work as expected:

```
mysql> SELECT @fid, @fid:=film.film_id, @fid FROM film,
inventory
    -> WHERE inventory.film_id = film.film_id LIMIT 20;

+------+--------------------+------+
| @fid | @fid:=film.film_id | @fid |
+------+--------------------+------+
| NULL |                  1 | 1    |
| 1    |                  1 | 1    |
| 1    |                  1 | 1    |
...
```

```
| 4    |                  4 | 4    |
| 4    |                  4 | 4    |
+------+--------------------+------+
20 rows in set, 1 warning (0.00 sec)
```

Now that if `@fid` isn't empty, the initial query will produce some results:

```
mysql> SELECT @fid, @fid:=film.film_id, @fid FROM film,
inventory
    -> WHERE inventory.film_id = film.film_id;

+------+--------------------+------+
| @fid | @fid:=film.film_id | @fid |
+------+--------------------+------+
|    4 |                  1 |    1 |
|    1 |                  1 |    1 |
...
|    4 |                  4 |    4 |
|    4 |                  4 |    4 |
+------+--------------------+------+
20 rows in set, 1 warning (0.00 sec)
```

It's best to avoid such circumstances where the behavior is not guaranteed and is hence unpredictable.

Index

About the Authors

Sergey Kuzmichev is a Senior Support Engineer working for Percona and has a Master of Science degree. He represents Percona in open source conferences and meetups all around the globe. He contributes to the community writing articles and blog posts about MySQL and open source databases. Sergey started his career as an Oracle DBA, later venturing into the DevOps side of things, but then returning to what he likes most: databases.

Vinicius Grippa is a Senior Support Engineer working for Percona and an Oracle Ace Associate. Vinicius has a Bachelor’s degree in Computer Science and has been working with databases for 13 years. He has experience in designing databases for mission-critical applications and, in the last few years, has become a specialist in MySQL and MongoDB ecosystems. Working in the Support team, he has helped Percona customers with hundreds of different cases featuring a vast range of scenarios and complexities. Vinicius is also active in the OS community, participating in virtual rooms like Slack, and speaking at MeetUps, and presenting conferences in Europe, Asia, North and South America.

Colophon

The animals on the cover of *Learning MySQL* are blue spotted crows *(Euploea midamus), butterflies found in India and Southeast Asia*.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *FILL IN CREDITS*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

1. Preface
    a. Who This Book Is for
    b. How This Book Is Organized
        i. Starting with MySQL
        ii. Using MySQL
    c. Conventions Used in This Book
    d. Using Code Examples
    e. O’Reilly Online Learning
    f. How to Contact Us
    g. Acknowledgments
        i. From Vinicius Grippa
        ii. From Sergey Kuzmichev
2. 1. Installing MySQL
    a. MySQL Forks
        i. MySQL community
        ii. Percona Server for MySQL
        iii. MariaDB Server
        iv. MySQL Enterprise
    b. Installation Choices and Platforms
        i. 1. Download the distribution that you want to install
        ii. 2. Install the distribution

            iii. 3. Perform any necessary post-installation setup
            iv. 4. Run Benchmarks
    c. Installing MySQL on Linux
            i. Installing MySQL on Centos 7
            ii. Installing MySQL on Centos 8
            iii. Installing MySQL on Ubuntu 20.04 LTS (Focal Fossa)
    d. Installing MySQL on macOS Big Sur
    e. Installing MySQL on Windows 10
    f. The Contents of the MySQL Directory
            i. MySQL 5.7 default files
            ii. MySQL 8.0 default files
    g. Using the Command-Line Interface
    h. Using Docker
            i. Installing Docker
    i. Using Sandboxes
    j. Upgrading MySQL Server
3. 2. Modeling and Designing Databases
    a. How Not to Develop a Database
    b. The Database Design Process
    c. The Entity Relationship Model
            i. Representing Entities

ii. Representing Relationships

iii. Partial and Total Participation

iv. Entity or Attribute?

v. Entity or Relationship?

vi. Intermediate Entities

vii. Weak and Strong Entities

d. Database Normalization

e. Normalizing an Example Table

i. First normal form: No repeating groups

ii. Second normal form: Eliminate redundant data

iii. Third normal form: Eliminate data not dependent on key

f. Entity Relationship Modeling Examples

g. Using the Entity Relationship Model

i. Mapping Entities and Relationships to Database Tables

ii. Creating a Bank Database ER Model

iii. Converting the EER to a MySQL Database Using Workbench

4. 3. Basic SQL

a. Using the Sakila Database

b. The SELECT Statement and Basic Querying Techniques

i. Single Table SELECTs

ii. Choosing Columns
iii. Selecting Rows with the WHERE Clause
iv. ORDER BY Clauses
v. The LIMIT Clause
vi. Joining Two Tables

c. The INSERT Statement
i. INSERT Basics
ii. Alternative Syntaxes

d. The DELETE Statement
i. DELETE Basics
ii. Using WHERE, ORDER BY, and LIMIT
iii. Removing All Rows with TRUNCATE

e. The UPDATE Statement
i. Examples
ii. Using WHERE, ORDER BY, and LIMIT

f. Exploring Databases and Tables with SHOW and mysqlshow

5. 4. Working with Database Structures
a. Creating and Using Databases
b. Creating Tables
i. Basics
ii. Collation and Character Sets

iii. Other Features

iv. Column Types

v. Keys and Indexes

vi. The AUTO_INCREMENT Feature

c. Altering Structures

i. Adding, Removing, and Changing Columns

ii. Adding, Removing, and Changing Indexes

iii. Renaming Tables and Altering Other Structures

d. Deleting Structures

i. Dropping Databases

ii. Removing Tables

6. 5. Advanced Querying

a. Aliases

i. Column Aliases

ii. Table Aliases

b. Aggregating Data

i. The DISTINCT Clause

ii. The GROUP BY Clause

iii. The HAVING Clause

c. Advanced Joins

i. The Inner Join

ii. The Union

iii. The Left and Right Joins

iv. The Natural Join

v. Constant expressions in joins

d. Nested Queries

i. Nested Query Basics

ii. The ANY, SOME, ALL, IN, and NOT IN Clauses

iii. The EXISTS and NOT EXISTS Clauses

iv. Nested Queries in the FROM Clause

e. Nested Queries in JOINs

f. User Variables

7. 6. Transactions and Locking

a. Isolation Levels

i. REPEATABLE READ

ii. READ COMMITTED

iii. READ UNCOMMITTED

iv. SERIALIZABLE

b. Locking

i. Metadata Locks

ii. Row Locks

iii. Deadlocks

c. MySQL Parameters Related to Isolation and Locks

i. transaction_isolation

        ii. innodb_lock_wait_timeout
        iii. innodb_print_all_deadlocks
        iv. lock_wait_timeout
        v. innodb_deadlock_detect

8. 7. Doing More with MySQL
    a. Inserting Data Using Queries
    b. Loading Data from Comma-Delimited Files
    c. Writing Data into Comma-Delimited Files
    d. Creating Tables with Queries
    e. Updates and Deletes with Multiple Tables
        i. Deletion
        ii. Updates
    f. Replacing Data
    g. The EXPLAIN Statement
    h. Alternative Storage Engines
        i. InnoDB
        ii. MyISAM and Aria
        iii. MyRocks and TokuDB
        iv. Other Table Types
9. 8. Managing Users and Privileges
    a. Understanding Users and Privileges
    b. root user

c. Creating and Using New Users
d. Grant tables
e. User and Privileges Management Commands, Logging and Replication
f. Modifying and Dropping a User
    i. Modifying the User
    ii. Dropping the User
g. Privileges
h. Static Versus Dynamic Privileges
i. SUPER privilege
j. Privilege Management Commands
k. Checking Privileges
l. The GRANT OPTION Privilege
m. Roles
n. Changing root’s password and insecure startup
o. Some ideas for secure setup

10. 9. Using Option Files
a. Structure of the Option File
b. Scope of Options
c. Search Order for Option Files
d. Special Option Files
    i. Login Path Configuration File
    ii. Persistent System Variables Configuration File

- e. Determining the Options in Effect

11. 10. Backups and Recovery
    - a. Physical and Logical Backups
        - i. Logical Backups
        - ii. Physical Backups
        - iii. Overview of Logical and Physical Backups
    - b. Replication as a Backup Tool
        - i. Infrastructure Failure
        - ii. Deployment bug
    - c. The mysqldump Program
        - i. Bootstrapping Replication with mysqldump
    - d. Loading Data from an SQL Dump File
    - e. mysqlpump
    - f. mydumper and myloader
    - g. Cold Backup and Filesystem Snapshots
    - h. Percona XtraBackup
        - i. Backing up and Recovering
        - ii. Advanced Features
    - i. Incremental Backups with XtraBackup
    - j. Other Physical Backup Tools
        - i. MySQL Enterprise Backup
        - ii. mariabackup

k. Point-in-Time Recovery
    i. Binary Logs Technical Background
    ii. Preserving Binary Logs
    iii. Identifying a PITR Target
    iv. Example Point-in-Time-Recovery: XtraBackup
    v. Example Point-in-Time-Recovery: mysqldump
l. Export and Import InnoDB Tablespaces
    i. Technical Background
    ii. Export Tablespace
    iii. Import Tablespace
    iv. XtraBackup Single-Table Restore
m. Testing and Verifying Your Backups
n. Database Backup Strategy Primer
12. 11. Configuring and Tuning the Server
    a. The MySQL Server Daemon
    b. MySQL Server Variables
        i. Checking Server Settings
        ii. Best Practices
13. 12. Monitoring MySQL Servers
    a. Operating System Metrics
        i. CPU
        ii. Disk

iii. Memory

iv. Network

b. MySQL Server Observability

i. Status Variables

ii. Basic Monitoring Recipes

iii. The Slow Query Log

iv. InnoDB Engine Status Report

c. Investigation Methods

i. USE Method

ii. RED Method

d. MySQL Monitoring Tools

e. Incident/diagnostic and Manual Data Collection

i. Gathering System Status Variable Values Periodically

ii. Using pt-stalk to Collect MySQL and OS Metrics

iii. Extended Manual Data Collection

14. 13. High Availability

a. Asynchronous Replication

i. Basic Parameters to Set in the Source and the Replica

ii. Create a Replica Using the Percona Xtrabackup Tool

iii. Create a Replica Using the Clone Plugin

iv. Create a Replica Using mysqldump

v. Create a Replica Using mydumper and myloader

vi. Group Replication

b. Synchronous replication

i. Galera Cluster (Percona XtraDB Cluster)

15. 14. MySQL in the Cloud

a. Database as a service (DBaaS)

i. Amazon RDS for MySQL/MariaDB

ii. Google Cloud for MySQL

iii. Azure SQL

b. Amazon Aurora RDS

c. MySQL in the Cloud Instances

d. MySQL in Kubernetes

i. Deploying Percona XtraDB Cluster in Kubernetes

16. 15. Load Balancing MySQL

a. Load Balancing with Aplication Drivers

b. ProxySQL Load Balancer

i. Installing and Configuring ProxySQL

c. HAProxy Load Balancer

i. Installing and Configuring HAProxy

d. MySQL Router

17. 16. Miscellaneous Topics
    a. MySQL Shell
        i. Installing MySQL Shell
        ii. Installing MySQL Shell on Ubuntu 20.04 Focal Fossa
        iii. Installing MySQL Shell in Centos 8
        iv. Deploying a Sandbox InnoDB Cluster with MySQL Shell
        v. MySQL Shell Utilities
    b. Flame Graphs
    c. Building MySQL from Source
        i. Building MySQL for Ubuntu Focal Fossa and ARM processors
    d. Analyzing a MySQL Crash
18. Index