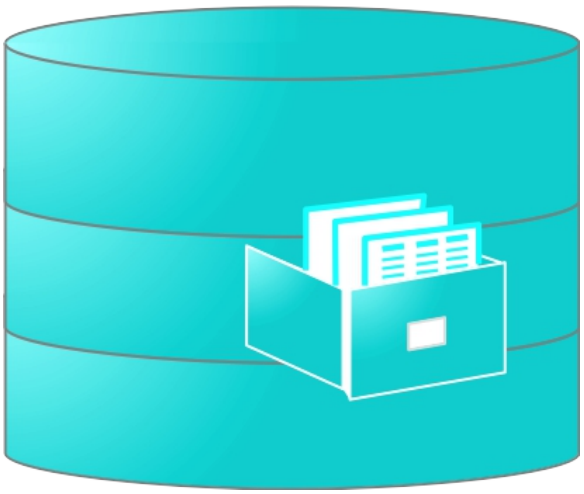
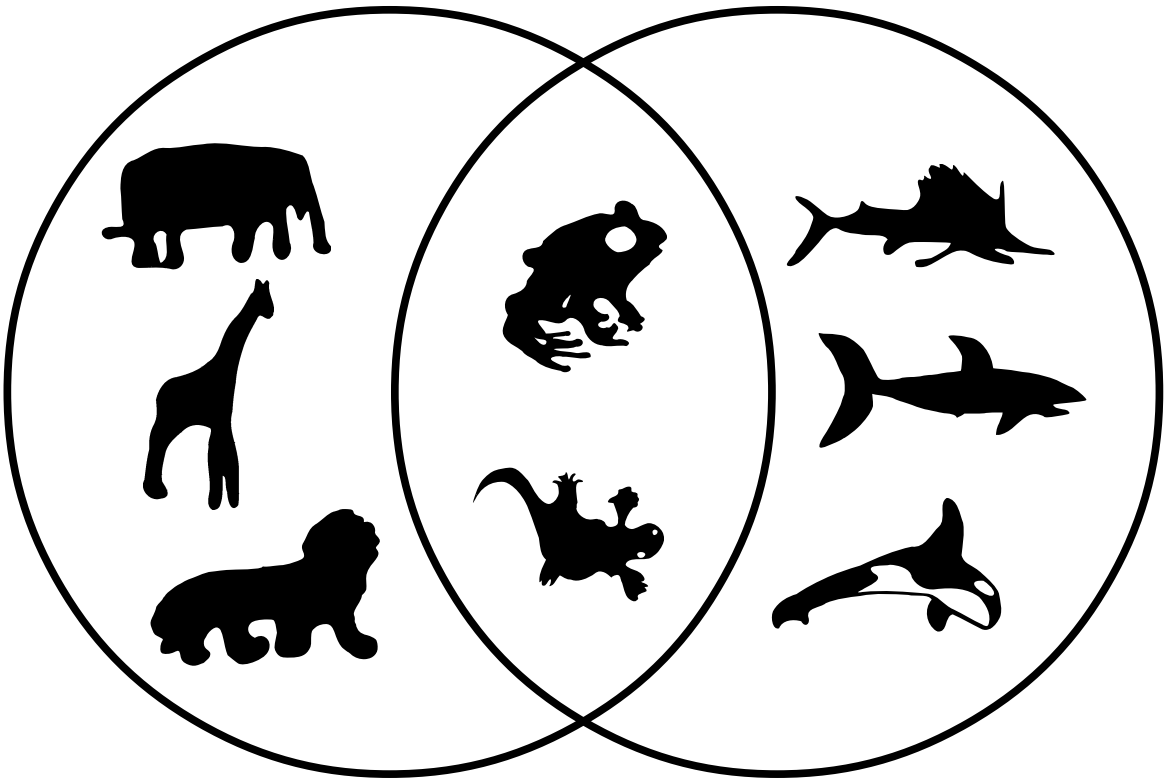


SQL Notes

Tikeswar Naik



About the **SQL Notes**:

Starting from the very basics, it covers enough material to make you feel confident and hit the ground running. After this, you can learn any advanced materials on the go.

1st edition, Aug 2020 (last edited Aug 2023)
`https://boo9.com`

Contents

Chapter 1

Introduction

1.1 Introduction

When you shop online, or call your dentist to book an appointment, or plan your next vacation, or <you get the idea>, data is the key enabler behind the scene.

For example, if you shop for t-shirts online, data may include information such as size, color, material composition, quantity available, price, reviews, etc. Based on your search query your app will try to get you the most relevant results by looking at the data available.

Data is any actionable information, and a database (DB, a hardware) stores data and provides ways to access and manipulate the data through a database management system (DBMS, a software). The database together with the DBMS is often loosely referred to as "Database" (Fig. 1.1).

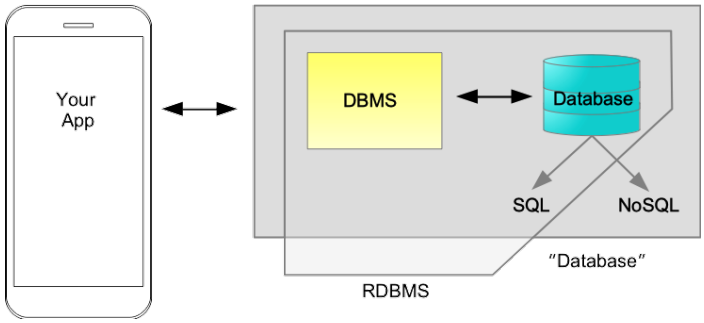


Fig. 1.1: RDBMS (Relational DataBase Management System)

Broadly speaking there are two types of database based on how data is stored (Fig. 1.2):

1. Relational DBMS (RDBMS) that store data in a tabular format
   - either as a row-store or column-store,
   - and use a structured query language (SQL) to write and query the data, and
2. NoSQL database that manage data in non-tabular format (e.g., key:value pair). You will hear this in big-data (out of scope here, but may be in the future).

The row-store is the most basic and intuitive way of thinking about storing data. In fact the row-store RDBMS are the ones most widely used today, and will be our main focus going forward.

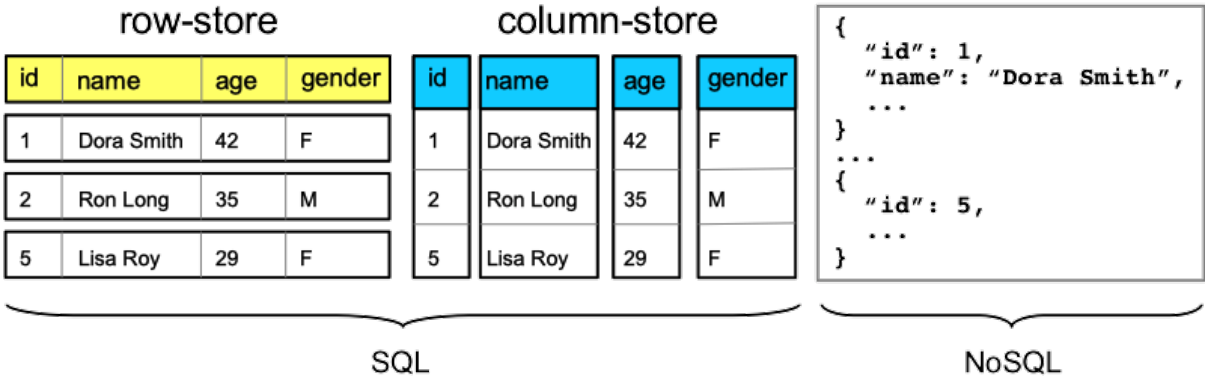


Fig. 1.2: SQL vs. NoSQL

There are different RDBMS offering different features - e.g., MySQL, SQLite, PostgreSQL, OracleDB, Microsoft SQL Server, etc. Once you learn SQL, you will be well equipped to work with vast majority of these databases.

But before we start learning SQL, we need to install and set up a database server in our computer, which follows next.

1.2 Installation notes

You can think of a database server as something in our computer that will allow us to create and use databases (Fig.1.3). MySQL is one of the most popular RDBMS which also offers database server functionality, and we will use that here.[1]

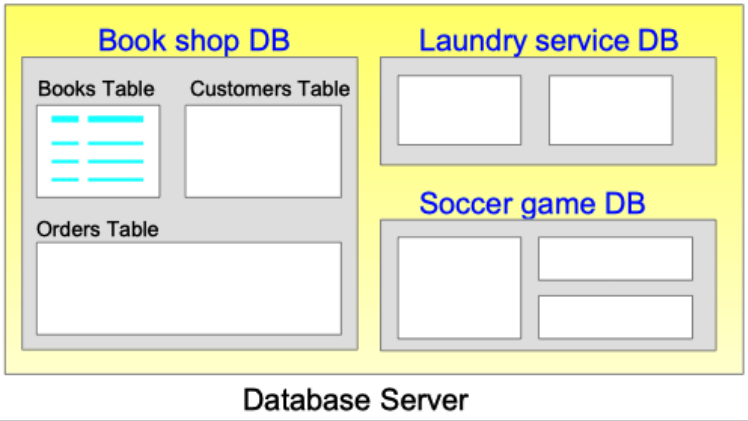


Fig. 1.3: Database Server

[1] You can choose to install a different database server, if you don't already have one. Most of the SQL we learn here will remain the same (with minor syntax changes among different databases). In fact you can also choose not to install any database, and use an online service to run SQL.

The installation process itself can be bit involved.[2] Besides, new versions (see 1.2.1) come out often which can have their own nuances for installation, making book-printed instructions obsolete soon after. We hope to put detailed and up-to-date installation instructions in the website (`https://boo9.com`) at some point, but until then looking online is your best bet (remember, you can also choose not to install any database and use an online service instead to run SQL).

If you do chose to install MySQL in your computer[3], Table 1.1 shows how to make sure you are all good to go! A screenshot of the terminal showing the same information is in Fig. 1.4.

```
1  Tikeswars-MacBook-Pro:~ tikka$ mysql -u root -p ← Log in as root user
2  Enter password: ← Enter your password
3  Welcome to the MySQL monitor.
4  ... ← Some output omitted for brevity
5  Type 'help;' or '\h' for help. ...
6
7  mysql> ← If you see this, you are good to go!
8  mysql> exit ← Type exit to get out of mysql
9  Bye
10 Tikeswars-MacBook-Pro:~ tikka$
```

Table 1.1: MySQL good to go!

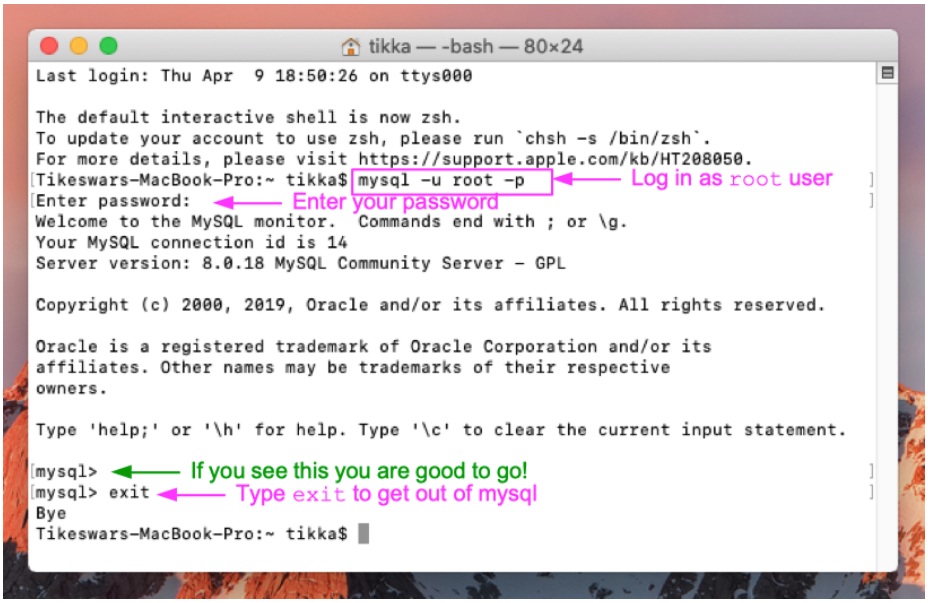


Fig. 1.4: Terminal - MySQL good to go!

1.2.1 Version

It is good to know how to find the version of the software you are using. There are couple of easy ways, see Table 1.2.

[2] Warning: installation + set up is the most frustrating part when learning a new language. But fear not, there will be soothing light at the end of the tunnel!

[3] Although it can be frustrating, trying to install and make it work would be a valuable experience. Remember one piece of advice - if you run into any issue during installation or programming in general, Google search is your best friend!

```
1  Using mysql -V:
2  Tikeswars-MacBook-Pro:~ tikka$ mysql -V ← Check the version
3  mysql  Ver 8.0.18 for macos10.14 on x86_64 (MySQL Community Server - GPL) ← 8.0.18
4  Tikeswars-MacBook-Pro:~ tikka$
5
6  Using SELECT VERSION():
7  mysql> SELECT VERSION(); ← Check the version
8  +-----------+
9  | VERSION() |
10 +-----------+
11 | 8.0.18    | ← We are using version 8.0.18, also confirmed above
12 +-----------+
13 1 row in set (0.00 sec)
14
15 mysql>
```

Table 1.2: MySQL version

The best way to learn SQL is by doing it. So let's get straight to it next.

Chapter 2

Database table

2.1 Conventions

Below are some conventions followed by most programmers:

1. Database name is all lowercase, and words separated by an underscore
   - e.g., 'book' instead of 'Book', or 'book_shop' instead of 'BookShop'
2. Table name is all lowercase and plurallized
   - e.g., 'customers' instead of 'customer' or 'Customers'
3. SQL commands are all in uppercase (but I won't blame you on this one :)
   - e.g., 'CREATE DATABASE' instead of 'create database'
4. SQL command ends with a semicolon (this is a must in MySQL)
5. **Use indentation/formatting**, especially for longer commands. It not only makes the code more readable, but also makes it much easier to write complex queries.

2.2 DATABASE commands

Table 2.1 shows some key commands to create a new database and work in that database. Most of the commands in the table are self explanatory, but a few notes:

- Single line comments start from two hyphens (`--`) to end of line. Multi-line comments start from slash and asterisk (`/*`) to asterisk and slash (`*/`).
- The `DROP` command will delete the database and all the tables in that database, so be careful! The `IF EXISTS` is optional, but not including it will give error if you try to delete a database that does not exist.
- I have used `my_db` as the database name here, but feel free to use a more descriptive name!

Let's see an example run of the commands in Table 2.1.

```
SHOW DATABASES; -- show all the databases in the server
DROP DATABASE IF EXISTS my_db; -- delete the database if exists
CREATE DATABASE my_db; -- create a new database
USE my_db; -- choose the database to use
SELECT DATABASE(); -- show the database in use currently
```

Table 2.1: DATABASE commands

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema | ← When you run SHOW DATABASES for the first time, you will see some pre-existing system databases. Do not mess with these! Just crate your own database and work inside that.
| mysql              | ←
| performance_schema | ←
| sys                | ←
+--------------------+
4 rows in set (0.00 sec)

mysql> DROP DATABASE my_db;
ERROR 1008 (HY000): Can't drop database 'my_db'; database doesn't exist ← Trying to delete a database that does not yet exist gives error
mysql> DROP DATABASE IF EXISTS my_db;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> CREATE DATABASE my_db; ← Create a new database called my_db
Query OK, 1 row affected (0.00 sec)

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| my_db              | ← You can see the newly created database my_db listed here
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)

mysql> USE my_db; ← Choose the database to use
Database changed
mysql> SELECT DATABASE(); ← Show the database in use currently
+------------+
| DATABASE() |
+------------+
| my_db      |
+------------+
1 row in set (0.00 sec)

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

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)

mysql>
```

2.3 TABLE commands

Table 2.2 shows some key commands to create a new table and insert data into that table. Most of the commands in the table are self explanatory, but a few notes:

- Remember, for tables we will be using the row-store format (see Fig. 1.2).
- With the `CREATE TABLE` command we are creating a table called `friends` which stores the name and age of our friends (in columns named `name` and `age` respectively).

  - The `name` column stores data in a string format, as `VARCHAR(255)` to be more specific (meaning, as a VARiable length CHARacter string with maximum length of 255 characters - this is a commonly used string format).
  - The `age` column stores data in INTeger format, as `INT` .
  - We will see more SQL data types in Section 3.2.

- When inserting data into a table, the order of the columns specified can be arbitrary, as long as the data being inserted match the specified order.
- Watch the quotes!
  - For string data type, it does not matter whether single or double quotes.
  - But it must be straight quotes, else it will give error. Especially if you copy-paste code and get some error, check the quotes.
- The `SHOW WARNINGS` works right after an error occurs, and then it is forgotten.
  - For example, in this case trying to assign the string value `'twenty nine'` for the `age` creates an error. However, if SQL can convert to the correct value it works fine (e.g., specifying `'29'` instead of `29` works fine).

```sql
CREATE TABLE friends ( -- create a new table
  name VARCHAR(255),
  age INT
);

INSERT INTO friends -- insert data into the table
  (age, name)
VALUES
  (42, 'Dora Smith'),
  (35, 'Ron Long');

SHOW WARNINGS; -- show any errors/warnings
SHOW TABLES; -- show all the tables in the current database
SHOW COLUMNS FROM friends; -- show the columns of the table
DESC friends;  -- describes the structure of the table
DROP TABLE IF EXISTS friends; -- delete the table if exists

SELECT * FROM friends; -- show all the table rows
```

Table 2.2: TABLE commands

Let's see an example run of the commands in Table 2.2.

```
mysql> USE my_db;
Database changed
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| my_db      | ← First make sure you are in the right database
+------------+
1 row in set (0.00 sec)

mysql>
mysql> SHOW TABLES;
Empty set (0.00 sec)

mysql> CREATE TABLE friends ( ← Create a new table called friends
    ->   name VARCHAR(255),
```

```
    ->   age INT
    -> ); ← mysql will keep giving the '->' automatically until it encounters a ';' indicating end-of-statement
Query OK, 0 rows affected (0.01 sec)

mysql> SHOW TABLES;
+----------------+
| Tables_in_my_db |
+----------------+
| friends         | ← The friends table we just created
+----------------+
1 row in set (0.00 sec)

mysql>
mysql> SHOW COLUMNS FROM friends;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| name  | varchar(255) | YES  |     | NULL    |       |
| age   | int(11)      | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> DESC friends; ← DESCribes the structure of the table (similar to SHOW COLUMNS)
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| name  | varchar(255) | YES  |     | NULL    |       |
| age   | int(11)      | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql>
mysql> SELECT * FROM friends;
Empty set (0.00 sec) ← No data in the table yet

mysql> INSERT INTO friends
    ->   (age, name) ← Note that the columns can be specified in any order
    -> VALUES
    ->   (42, 'Dora Smith'),
    ->   (35, 'Ron Long'); ← You can insert multiple rows of data at once separated by comma
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM friends;
+------------+------+
| name       | age  |
+------------+------+
| Dora Smith |   42 | ← The table now has the data we just inserted
| Ron Long   |   35 | ←
+------------+------+
2 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO friends (name, age) VALUES ('Lisa Roy', 'twenty nine');
ERROR 1366 (HY000): Incorrect integer value: 'twenty nine' for column 'age' at row 1 ← Error because SQL cannot convert the string
      'twenty nine' to INT
mysql> SHOW WARNINGS; ← Shows any errors in the preceding command
+-------+------+-----------------------------------------------------------------+
| Level | Code | Message                                                         |
+-------+------+-----------------------------------------------------------------+
| Error | 1366 | Incorrect integer value: 'twenty nine' for column 'age' at row 1 |
+-------+------+-----------------------------------------------------------------+
1 row in set (0.00 sec)

mysql>
mysql> INSERT INTO friends (name, age) VALUES ('Lisa Roy', '29');
Query OK, 1 row affected (0.00 sec) ← No error here because SQL can convert the string '29' to INT

mysql> SHOW WARNINGS;
Empty set (0.00 sec) ← The previous error is forgotten at this point

mysql>
mysql> SELECT * FROM friends;
+------------+------+
| name       | age  |
+------------+------+
| Dora Smith |   42 |
| Ron Long   |   35 |
| Lisa Roy   |   29 | ← Just making sure the string '29' worked fine
+------------+------+
3 rows in set (0.00 sec)

mysql>
mysql> DROP TABLE IF EXISTS friends; ← Deleting the table we had just created
Query OK, 0 rows affected (0.01 sec)

mysql> SHOW TABLES;
Empty set (0.00 sec)

mysql>
```

2.3.1 Script file

By now you have probably noticed how cumbersome it is to type the SQL commands directly in the terminal, especially long or multi-line commands. On top of that, if there is a typo or mistake in a command, going back to fix and re-run is a nightmare. Not to mention, we need a way to save the awesome queries we write! So here we will see how to write the SQL commands to a file and then run the file in the terminal.[1]

It is fairly straightforward - write the commands in a text editor of your choice (which has syntax highlighting preferably) → save the file as `file_name.sql` → run the script file in the terminal using the `SOURCE` command (see Table 2.3).

- The `file_name` can be any valid file name (make it descriptive though), but needs to have the `.sql` extension.
- The `file_path` can be absolute or relative path. Need to specify the file path if the script file is not in the same directory from where you entered the `mysql` prompt.

```
SOURCE file_path/file_name.sql;
```

Table 2.3: Running a script file

Fig. 2.1 shows the screenshot of a simple script file. Let's see an example run of the commands in Table 2.3 using the script file shown in Fig. 2.1.

```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| my_db      | ← Make sure you are in a database (btw, we could have used the script file itself to get inside a database)
+------------+
1 row in set (0.00 sec)

mysql> SHOW TABLES;
Empty set (0.00 sec) ← The database is empty to begin with)

mysql> SOURCE scripts/my_first_script.sql; ← Run the script file
Query OK, 0 rows affected, 1 warning (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

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

mysql> SELECT * FROM friends;
+------------+------+
| name       | age  |
+------------+------+
| Dora Smith |   42 | ← Using the script file a new table was created and data were inserted into the table, as confirmed here
| Ron Long   |   35 | ←
+------------+------+
2 rows in set (0.00 sec)

mysql>
```

Congratulations! Now you know how to create a new database, and then create a new table in that database, and insert data into that table. This is an important milestone, but fun has just started! Let's keep going.

[1] You could also use an IDE (Integrated Development Environment). IDEs provide extra bells and whistles, but may require additional installation and set up (feel free to explore this on your own).

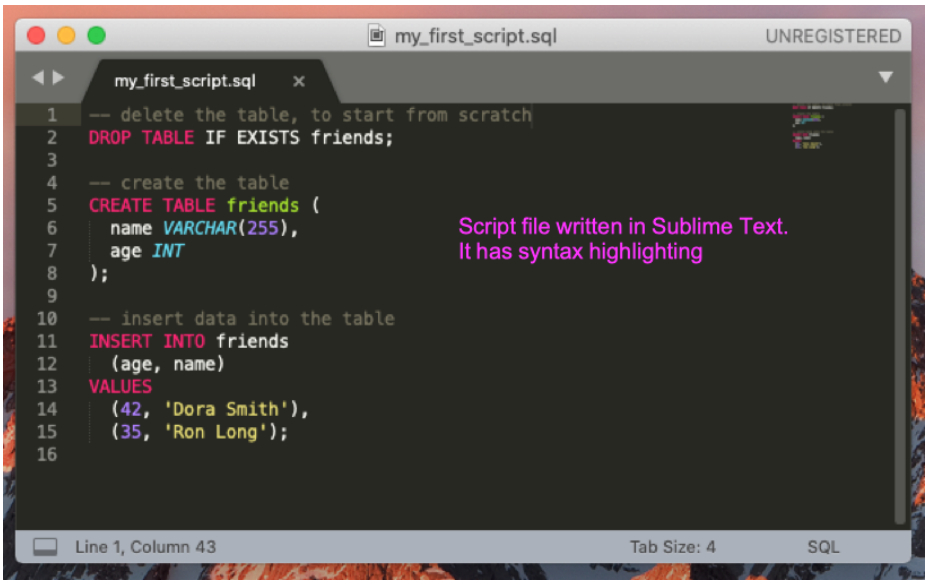


Fig. 2.1: Screenshot of a script file

Chapter 3

CRUD

3.1 CRUD

CRUD stands for Create, Read, Update, and Delete - the four basic operations in any persistent storage. Let's take a look into these operations individually.

3.1.1 Create

We already know how to create a database (Sec. 2.2)!

We also know how to create a table - in Sec. 2.3 we created a simple table with two columns. However, table columns come with many different options for real world applications, so let's delve into that a bit more here.

NOT NULL; DEFAULT

As the name suggests, if a column is marked `NOT NULL`, that value cannot be null. `DEFAULT` assigns a default value if that field is not specified.

A typical usage is shown in Table 3.1.

```
1 CREATE TABLE users (
2   first_name VARCHAR(255),
3   last_name VARCHAR(255) NOT NULL,
4   status VARCHAR(255) DEFAULT 'active'
5 );
```

Table 3.1: CRUD - Create (NOT NULL; DEFAULT)

Let's see how it works with an example.

```
1 mysql> CREATE TABLE users (
2     ->   first_name VARCHAR(255),
3     ->   last_name VARCHAR(255) NOT NULL, ← last_name cannot be null
4     ->   status VARCHAR(255) DEFAULT 'active' ← If status is not specified use 'active' as default
5     -> );
6 Query OK, 0 rows affected (0.02 sec)
7
```

```
mysql> DESC users;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra | ← Look at the Null and Default columns
+------------+--------------+------+-----+---------+-------+
| first_name | varchar(255) | YES  |     | NULL    |       |
| last_name  | varchar(255) | NO   |     | NULL    |       |
| status     | varchar(255) | YES  |     | active  |       |
+------------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO users
    ->   (first_name, last_name, status)
    -> VALUES
    ->   ('Dora', 'Smith', 'inactive');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM users;
+------------+-----------+----------+
| first_name | last_name | status   |
+------------+-----------+----------+
| Dora       | Smith     | inactive |
+------------+-----------+----------+
1 row in set (0.00 sec)

mysql>
mysql> INSERT INTO users
    ->   (last_name)
    -> VALUES
    ->   ('Long');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM users;
+------------+-----------+----------+
| first_name | last_name | status   |
+------------+-----------+----------+
| Dora       | Smith     | inactive |
| NULL       | Long      | active   | ← Notice the first_name and status columns (only the last_name was specified)
+------------+-----------+----------+
2 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO users
    ->   (first_name)
    -> VALUES
    ->   ('Lisa');
ERROR 1364 (HY000): Field 'last_name' doesn't have a default value ← We got error because we didn't specify the last_name, which cannot
      be null and doesn't have a default value
mysql> SELECT * FROM users;
+------------+-----------+----------+
| first_name | last_name | status   |
+------------+-----------+----------+
| Dora       | Smith     | inactive |
| NULL       | Long      | active   |
+------------+-----------+----------+
2 rows in set (0.00 sec) ← As expected, 'Lisa' wasn't inserted

mysql>
```

PRIMARY KEY, AUTO_INCREMENT; UNIQUE

A `PRIMARY KEY` is an unique identifier of a row in a table.

- And thus it cannot be null.
- It can be formed using a single column or multiple columns.
- If it is a numerical id, it can be assigned automatically using `AUTO_INCREMENT` (highly recommended).

`UNIQUE` means that column cannot have any duplicate values.

- You can also specify the `UNIQUE` constraint to combination of columns.
- vs. `PRIMARY KEY`
  - Used for enforcing uniqueness for column(s) that are not part of the `PRIMARY KEY`.
  - A table can have only one `PRIMARY KEY` constraint, but multiple `UNIQUE` constraints.

– Unlike `PRIMARY KEY`, there can be null values.

A typical usage is shown in Table 3.2.

```
CREATE TABLE users (
  id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
  email VARCHAR(255) UNIQUE,
  name VARCHAR(255) NOT NULL
);
```

Table 3.2: CRUD - Create (PRIMARY KEY, AUTO_INCREMENT; UNIQUE)

Let's see how it works with an example.

```
mysql> CREATE TABLE users (
    ->   id INT AUTO_INCREMENT NOT NULL PRIMARY KEY, ← id is the primary key, which is auto incremented
    ->   email VARCHAR(255) UNIQUE, ← email has to be unique
    ->   name VARCHAR(255) NOT NULL
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> DESC users;
+-------+--------------+------+-----+---------+----------------+
| Field | Type         | Null | Key | Default | Extra          | ← Look at the Key and Extra columns
+-------+--------------+------+-----+---------+----------------+
| id    | int(11)      | NO   | PRI | NULL    | auto_increment |
| email | varchar(255) | YES  | UNI | NULL    |                |
| name  | varchar(255) | NO   |     | NULL    |                |
+-------+--------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO users
    ->   (email, name)
    -> VALUES
    ->    ('dora@gmail.com', 'Dora Smith');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM users;
+----+----------------+------------+
| id | email          | name       |
+----+----------------+------------+
|  1 | dora@gmail.com | Dora Smith | ← id is auto-incremented starting from 1 (you can change this default starting)
+----+----------------+------------+
1 row in set (0.00 sec)

mysql>
mysql> INSERT INTO users
    ->   (name)
    -> VALUES
    ->    ('Ron Long');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM users;
+----+----------------+------------+
| id | email          | name       |
+----+----------------+------------+
|  1 | dora@gmail.com | Dora Smith |
|  2 | NULL           | Ron Long   | ← email is null because it wasn't specified. Also, notice that the id incremented automatically.
+----+----------------+------------+
2 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO users
    ->   (name)
    -> VALUES
    ->    ('Lisa Roy');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM users;
+----+----------------+------------+
| id | email          | name       |
+----+----------------+------------+
|  1 | dora@gmail.com | Dora Smith |
|  2 | NULL           | Ron Long   |
|  3 | NULL           | Lisa Roy   | ← The UNIQUE constraint allows multiple null values
+----+----------------+------------+
3 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO users
    ->   (email, name)
    -> VALUES
```

```
    ->    ('dora@gmail.com', 'Dora Smith');
ERROR 1062 (23000): Duplicate entry 'dora@gmail.com' for key 'email' ← Error because of duplicate email
mysql> SELECT * FROM users;
+----+----------------+------------+
| id | email          | name       |
+----+----------------+------------+
|  1 | dora@gmail.com | Dora Smith |
|  2 | NULL           | Ron Long   |
|  3 | NULL           | Lisa Roy   |
+----+----------------+------------+
3 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO users
    ->   (email, name)
    -> VALUES
    ->    ('dora_smith@gmail.com', 'Dora Smith');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM users;
+----+----------------------+------------+
| id | email                | name       |
+----+----------------------+------------+
|  1 | dora@gmail.com       | Dora Smith |
|  2 | NULL                 | Ron Long   |
|  3 | NULL                 | Lisa Roy   |
|  5 | dora_smith@gmail.com | Dora Smith | ← The name column doesn't have UNIQUE constraint, so duplicate values are allowed. Also notice
      the id column, the earlier invalid INSERT INTO still incremented the id.
+----+----------------------+------------+
4 rows in set (0.00 sec)

mysql>
```

Note that while inserting data into a table we need to specify at least the bare minimum fields, keeping in mind the `DEFAULT` values if any.

3.1.2 Read

In most jobs, databases and tables will be already there (you won't have to create a new database or table). There will be 'special' people doing update and delete to the databases, as these are highly risky operations (you don't want to mess with the data). So out of the CRUD, read is the operation you will be doing most, if not all, of the time.

We read from a database using the `SELECT` command, and we have already used that a few times in its most basic form (`SELECT * FROM` *table_name*). Table 3.3 shows a few more options to customize the reading of the data. Throughout the rest of this chapter we will see how to write very powerful read queries building on top of these options.

```
SELECT * FROM friends;

SELECT age, last_name FROM friends
LIMIT 3 OFFSET 2;

SELECT
  first_name AS Name, -- using alias
  last_name AS 'Last Name' -- use quotes if the alias contains any spaces
FROM friends
WHERE last_name='LonG';
```

Table 3.3: CRUD - Read

Let's see how these options in Table 3.3 work with an example.

```
mysql> CREATE TABLE friends (
    ->   id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    ->   first_name VARCHAR(255),
    ->   last_name VARCHAR(255),
    ->   age INT
```

```
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> INSERT INTO friends
    ->   (first_name, last_name, age)
    -> VALUES
    ->    ('Dora', 'Smith', 42),
    ->    ('Ron', 'Long', 35),
    ->    ('Lisa', 'Roy', 29),
    ->    ('Dora', 'G.', 21),
    ->    ('Tara', 'Long', 29),
    ->    ('Ryan', 'Davis', 40);
Query OK, 6 rows affected (0.00 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql>
mysql> SELECT * FROM friends; ← Read all the rows
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  1 | Dora       | Smith     |   42 |
|  2 | Ron        | Long      |   35 |
|  3 | Lisa       | Roy       |   29 |
|  4 | Dora       | G.        |   21 |
|  5 | Tara       | Long      |   29 |
|  6 | Ryan       | Davis     |   40 |
+----+------------+-----------+------+
6 rows in set (0.00 sec)

mysql>
mysql> SELECT age, last_name FROM friends ← Choose the columns and the order
    -> LIMIT 3 OFFSET 2; ← Limit to reading 3 rows from the top, offset by 2 rows
+------+-----------+
| age  | last_name |
+------+-----------+
|   29 | Roy       |
|   21 | G.        |
|   29 | Long      |
+------+-----------+
3 rows in set (0.00 sec)

mysql>
mysql> SELECT
    ->   first_name AS Name, ← Using alias
    ->   last_name AS 'Last Name' ← Use quotes if the alias contains any space characters
    -> FROM friends
    -> WHERE last_name='LonG'; ← Read only the rows with specific last name (note, capitalization does not matter while comparing strings)
+------+-----------+
| Name | Last Name |
+------+-----------+
| Ron  | Long      |
| Tara | Long      |
+------+-----------+
2 rows in set (0.00 sec)

mysql>
```

We will see a lot more of the `SELECT` command throughout the rest of this chapter.

3.1.3 Update and Delete

While we will be doing the read operation most of the time, occasionally we may need to update or delete records in a database. Table 3.4 shows some basic options for that.

If you ever have to do update or delete, it is highly recommended that you do a **`SELECT` before any `UPDATE` or `DELETE`** to make sure you are affecting only the right rows. There is no undo (undoing a `UPDATE` or `DELETE` is very cumbersome, and often impossible)! Also, it may be useful to have a column to record when a row was last updated.

Let's see how these options in Table 3.4 work with an example.

```
mysql> CREATE TABLE friends ( ← Create a table
    ->   id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    ->   first_name VARCHAR(255),
    ->   last_name VARCHAR(255),
    ->   age INT,
    ->   updated_at TIMESTAMP DEFAULT NOW() ON UPDATE NOW()
    -> );
Query OK, 0 rows affected (0.01 sec)

```

```sql
CREATE TABLE friends (
  ...,
  -- introducing ON UPDATE
  updated_at TIMESTAMP DEFAULT NOW() ON UPDATE NOW()
);

UPDATE friends
SET last_name = 'L.', age = age + 1
WHERE last_name='Long';

DELETE FROM friends
WHERE first_name='Dora';

DELETE FROM friends; -- WARNING, this will delete all the rows!
```

Table 3.4: CRUD - Update & Delete

```
mysql>
mysql> INSERT INTO friends ← Add some data
    ->   (first_name, last_name, age)
    -> VALUES
    ->    ('Dora', 'Smith', 42),
    ->    ('Ron', 'Long', 35),
    ->    ('Lisa', 'Roy', 29),
    ->    ('Dora', 'G.', 21),
    ->    ('Tara', 'Long', 29),
    ->    ('Ryan', 'Davis', 40);
Query OK, 6 rows affected (0.01 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM friends;
+----+------------+-----------+------+---------------------+
| id | first_name | last_name | age  | updated_at          |
+----+------------+-----------+------+---------------------+
|  1 | Dora       | Smith     |   42 | 2020-04-22 06:27:42 |
|  2 | Ron        | Long      |   35 | 2020-04-22 06:27:42 |
|  3 | Lisa       | Roy       |   29 | 2020-04-22 06:27:42 |
|  4 | Dora       | G.        |   21 | 2020-04-22 06:27:42 |
|  5 | Tara       | Long      |   29 | 2020-04-22 06:27:42 |
|  6 | Ryan       | Davis     |   40 | 2020-04-22 06:27:42 |
+----+------------+-----------+------+---------------------+
6 rows in set (0.00 sec)

mysql>
mysql> SELECT * FROM friends ← SELECT before an UPDATE!
    -> WHERE last_name='Long';
+----+------------+-----------+------+---------------------+
| id | first_name | last_name | age  | updated_at          |
+----+------------+-----------+------+---------------------+
|  2 | Ron        | Long      |   35 | 2020-04-22 06:27:42 |
|  5 | Tara       | Long      |   29 | 2020-04-22 06:27:42 |
+----+------------+-----------+------+---------------------+
2 rows in set (0.01 sec)

mysql> UPDATE friends ← Updating the friends table
    -> SET last_name = 'L.', age = age + 1 ← Fields being updated
    -> WHERE last_name='Long'; ← Specific rows being updated
Query OK, 2 rows affected (0.00 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> SELECT * FROM friends;
+----+------------+-----------+------+---------------------+
| id | first_name | last_name | age  | updated_at          | ← Look at the last_name, age, as well as the updated_at fields
+----+------------+-----------+------+---------------------+
|  1 | Dora       | Smith     |   42 | 2020-04-22 06:27:42 |
|  2 | Ron        | L.        |   36 | 2020-04-22 06:28:20 | ← Updated
|  3 | Lisa       | Roy       |   29 | 2020-04-22 06:27:42 |
|  4 | Dora       | G.        |   21 | 2020-04-22 06:27:42 |
|  5 | Tara       | L.        |   30 | 2020-04-22 06:28:20 | ← Updated
|  6 | Ryan       | Davis     |   40 | 2020-04-22 06:27:42 |
+----+------------+-----------+------+---------------------+
6 rows in set (0.00 sec)

mysql>
mysql> SELECT * FROM friends ← SELECT before a DELETE!
    -> WHERE first_name='Dora';
+----+------------+-----------+------+---------------------+
| id | first_name | last_name | age  | updated_at          |
+----+------------+-----------+------+---------------------+
```

```
|  1 | Dora       | Smith     |   42 | 2020-04-22 06:27:42 |
|  4 | Dora       | G.        |   21 | 2020-04-22 06:27:42 |
+----+------------+-----------+------+---------------------+
2 rows in set (0.00 sec)

mysql> DELETE FROM friends ← Delete from the friends table
    -> WHERE first_name='Dora'; ← Specific rows being deleted
Query OK, 2 rows affected (0.00 sec)

mysql> SELECT * FROM friends;
+----+------------+-----------+------+---------------------+
| id | first_name | last_name | age  | updated_at          |
+----+------------+-----------+------+---------------------+
|  2 | Ron        | L.        |   36 | 2020-04-22 06:28:20 |
|  3 | Lisa       | Roy       |   29 | 2020-04-22 06:27:42 |
|  5 | Tara       | L.        |   30 | 2020-04-22 06:28:20 |
|  6 | Ryan       | Davis     |   40 | 2020-04-22 06:27:42 |
+----+------------+-----------+------+---------------------+ ← The deleted rows are no longer present
4 rows in set (0.00 sec)

mysql>
mysql> DELETE FROM friends; ← WARNING, this will delete all the rows!
Query OK, 4 rows affected (0.00 sec)

mysql> SELECT * FROM friends;
Empty set (0.00 sec) ← Empty table, with all rows deleted!

mysql>
mysql> INSERT INTO friends
    ->   (first_name, last_name, age)
    -> VALUES
    ->    ('Dora', 'Smith', 42),
    ->    ('Ron', 'Long', 35);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM friends;
+----+------------+-----------+------+---------------------+
| id | first_name | last_name | age  | updated_at          |
+----+------------+-----------+------+---------------------+
|  7 | Dora       | Smith     |   42 | 2020-04-22 07:13:05 | ← Note: If you insert new data, deleted ids are not recycled
|  8 | Ron        | Long      |   35 | 2020-04-22 07:13:05 |
+----+------------+-----------+------+---------------------+
2 rows in set (0.00 sec)

mysql>
```

3.1.4 CRUD syntax tips

Table 3.5 shows some tips to remember the CRUD syntax. Once you have enough practice the syntax will come naturally, but this is especially handy in the beginning when you are learning.

```
CREATE TABLE table (col1 type1, ...);
INSERT INTO table (col3, ...) VALUES (val3, ...), ...;

SELECT cols FROM table WHERE row=?;

UPDATE table SET col=? WHERE row=?;

DELETE FROM table WHERE row=?;
```

Table 3.5: CRUD - Syntax tips

Now we know the basic CRUD operations and how to write simple queries. Next let's take a look into the different data types SQL offers.

3.2 Data types

In real world applications data come in different types, such as boolean (yes/no, or true/false), integer (1, -19, 35, ...), real number (57.98, 0.0, -12.001, ...), string ('hello world!', 'Dora Smith',

...), date and time (18 August 1985, 2020-04-22 06:28:20, ...), etc. As you could imagine, SQL offers different data types to handle these.

We have already seen a few types earlier, but a more comprehensive list of SQL data types is shown in Table 3.6. Only a few of these (that are highlighted) are used most frequently. However, it is a good idea to just keep in mind that there are many data types available in case you need.

As a rule of thumb ...

- Use `DECIMAL(size, d)` when you know how many decimal points you want to track (e.g., two decimal points for money). Use `DOUBLE` when variable decimal precision is needed.
- `VARCHAR(size)` is a variable length string. The size parameter specifies the maximum number of characters (can be from 0 to 65535). Typically `VARCHAR(255)` is used.
- Use `DATETIME` to store a specific date-and-time value (e.g., when an event had occurred, or will occur). Use `TIMESTAMP` for time stamping (e.g., when was a record created or last updated). The general format for both is '`YYYY-MM-DD hh:mm:ss[.fraction]`'. Use the `DATE` type when you don't need to store the time part.
- **Range and memory**: Know that different data types accept different ranges of values, and thus have different memory requirements. You can explore more about data types here: `https://dev.mysql.com/doc/refman/8.0/en/data-types.html`

Numeric Types	String Types	Date Types
INT(size)	CHAR(size)	DATE
TINYINT(size)	VARCHAR(size)	DATETIME
SMALLINT(size)	BINARY(size)	TIMESTAMP
MEDIUMINT(size)	VARBINARY(size)	TIME
BIGINT(size)	BLOB(size)	YEAR
DECIMAL(size, d)	TINYBLOB	
NUMERIC	MEDIUMBLOB	
FLOAT(p)	LONGBLOB	
DOUBLE(size, d)	TEXT(size)	
BIT(size)	TINYTEXT	
BOOL	MEDIUMTEXT	
	LONGTEXT	
	ENUM(val1, val2, val3, ...)	
	SET(val1, val2, val3, ...)	

Table 3.6: SQL data types

Let's take a look at some of these data types through examples - first the numeric types.

```
1  mysql> CREATE TABLE my_table ( ← Note the numeric types
2      ->   value_int INT, ←
3      ->   value_decimal1 DECIMAL, ← Default is zero digits after the decimal point
4      ->   value_decimal2 DECIMAL(8,2), ← Total 8 digits, including 2 digits after the decimal point
5      ->   value_double DOUBLE ←
6      -> );
7  Query OK, 0 rows affected (0.01 sec)
8
9  mysql>
10 mysql> INSERT INTO my_table
11     ->   (value_int, value_decimal1, value_decimal2, value_double)
12     -> VALUES
13     ->    (1234.5678, 1234.5678, 1234.5678, 1234.5678);
14 Query OK, 1 row affected, 2 warnings (0.00 sec)
15
```

```
mysql> SHOW WARNINGS;
+-------+------+-----------------------------------------------------+
| Level | Code | Message                                             |
+-------+------+-----------------------------------------------------+
| Note  | 1265 | Data truncated for column 'value_decimal1' at row 1 |
| Note  | 1265 | Data truncated for column 'value_decimal2' at row 1 |
+-------+------+-----------------------------------------------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM my_table;
+-----------+----------------+----------------+--------------+
| value_int | value_decimal1 | value_decimal2 | value_double |
+-----------+----------------+----------------+--------------+
|      1235 |           1235 |        1234.57 |    1234.5678 | ← Note the truncation
+-----------+----------------+----------------+--------------+
1 row in set (0.00 sec)

mysql>
mysql> INSERT INTO my_table
    ->   (value_int, value_decimal1, value_decimal2, value_double)
    -> VALUES
    ->    (1/3, 1/3, 1/3, 1/3);
Query OK, 1 row affected, 2 warnings (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+-----------------------------------------------------+
| Level | Code | Message                                             |
+-------+------+-----------------------------------------------------+
| Note  | 1265 | Data truncated for column 'value_decimal1' at row 1 |
| Note  | 1265 | Data truncated for column 'value_decimal2' at row 1 |
+-------+------+-----------------------------------------------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM my_table;
+-----------+----------------+----------------+--------------+
| value_int | value_decimal1 | value_decimal2 | value_double |
+-----------+----------------+----------------+--------------+
|      1235 |           1235 |        1234.57 |    1234.5678 |
|         0 |              0 |           0.33 |  0.333333333 | ← Note the truncation
+-----------+----------------+----------------+--------------+
2 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO my_table
    ->   (value_int, value_decimal1, value_decimal2, value_double)
    -> VALUES
    ->    (1/3*12, 1/3*12, 1/3*12, 1/3*12);
Query OK, 1 row affected, 2 warnings (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+-----------------------------------------------------+
| Level | Code | Message                                             |
+-------+------+-----------------------------------------------------+
| Note  | 1265 | Data truncated for column 'value_decimal1' at row 1 |
| Note  | 1265 | Data truncated for column 'value_decimal2' at row 1 |
+-------+------+-----------------------------------------------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM my_table;
+-----------+----------------+----------------+--------------+
| value_int | value_decimal1 | value_decimal2 | value_double |
+-----------+----------------+----------------+--------------+
|      1235 |           1235 |        1234.57 |    1234.5678 |
|         0 |              0 |           0.33 |  0.333333333 |
|         4 |              4 |           4.00 |  3.999999996 | ← Note the precision
+-----------+----------------+----------------+--------------+
3 rows in set (0.00 sec)

mysql>
mysql> SELECT
    ->   value_int*12, value_decimal1*12, value_decimal2*12, value_double*12
    -> FROM my_table;
+--------------+-------------------+-------------------+--------------------+
| value_int*12 | value_decimal1*12 | value_decimal2*12 | value_double*12    |
+--------------+-------------------+-------------------+--------------------+
|        14820 |             14820 |          14814.84 | 14814.813600000001 | ← Note the precision
|            0 |                 0 |              3.96 |        3.999999996 | ←
|           48 |                48 |             48.00 |       47.999999952 | ←
+--------------+-------------------+-------------------+--------------------+
3 rows in set (0.00 sec)

mysql>
```

We will also encounter the `BOOL` type often. Let's take a look at that with an example.

```
mysql> USE my_db;
Database changed
mysql> DROP TABLE IF EXISTS bool_demo;
Query OK, 0 rows affected (0.01 sec)

```

```
mysql>
mysql> CREATE TABLE bool_demo (
    ->   bool_expr CHAR(10),
    ->   bool_value BOOL ← The BOOL type
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> DESC bool_demo;
+------------+------------+------+-----+---------+-------+
| Field      | Type       | Null | Key | Default | Extra |
+------------+------------+------+-----+---------+-------+
| bool_expr  | char(10)   | YES  |     | NULL    |       |
| bool_value | tinyint(1) | YES  |     | NULL    |       | ← Boolean values are stored as TINYINT(1) (tiny int of size 1)
+------------+------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql>
mysql> INSERT INTO bool_demo
    ->   (bool_expr, bool_value)
    -> VALUES
    ->    ('true', TRUE),
    ->    ('1<2', 1<2), ← You can also specify an expression
    ->    ('false', FALSE),
    ->    ('1>2', 1>2);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM bool_demo;
+-----------+------------+
| bool_expr | bool_value |
+-----------+------------+
| true      |          1 | ← TRUE is stored as 1
| 1<2       |          1 | ←
| false     |          0 | ← FALSE is stored as 0
| 1>2       |          0 | ←
+-----------+------------+
4 rows in set (0.00 sec)

mysql>
```

Next let's take a look at the string types.

```
mysql> CREATE TABLE my_table (
    ->   name VARCHAR(8) ← We are saying the name can be max 8 chars long
    -> );
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> INSERT INTO my_table
    ->   (name)
    -> VALUES
    ->    ('Dora S.');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM my_table;
+---------+
| name    |
+---------+
| Dora S. |
+---------+
1 row in set (0.00 sec)

mysql>
mysql> INSERT INTO my_table
    ->   (name)
    -> VALUES
    ->    ('Dora Smith'); ← Value for name has 10 chars
ERROR 1406 (22001): Data too long for column 'name' at row 1 ← Error because the value for name has > 8 chars
mysql> SHOW WARNINGS;
+-------+------+-----------------------------------------+
| Level | Code | Message                                 |
+-------+------+-----------------------------------------+
| Error | 1406 | Data too long for column 'name' at row 1 |
+-------+------+-----------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM my_table;
+---------+
| name    |
+---------+
| Dora S. |
+---------+ ← Confirmation that 'Dora Smith' was not inserted
1 row in set (0.00 sec)

mysql>
```

Finally let's take a look at the date types.

```
mysql> CREATE TABLE users (
    ->   id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    ->   name VARCHAR(255),
    ->   dob DATE, ← We need only the date, and not the time
    ->   next_reminder DATETIME, ← Storing a future event date and time
    ->   updated_at TIMESTAMP NOT NULL DEFAULT NOW() ON UPDATE NOW() ← Time stamping last update to this record
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> INSERT INTO users
    ->   (name, dob, next_reminder)
    -> VALUES
    ->   ('Dora Smith', '1978-04-28', '2020-06-20 10:30:00'), ← Note the input format
    ->   ('Ron Long', '1985-07-10', '2020-12-20 10:30:00'); ←
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM users;
+----+------------+------------+---------------------+---------------------+
| id | name       | dob        | next_reminder       | updated_at          |
+----+------------+------------+---------------------+---------------------+
|  1 | Dora Smith | 1978-04-28 | 2020-06-20 10:30:00 | 2020-04-23 05:18:10 |
|  2 | Ron Long   | 1985-07-10 | 2020-12-20 10:30:00 | 2020-04-23 05:18:10 |
+----+------------+------------+---------------------+---------------------+
2 rows in set (0.00 sec)

mysql>
```

3.3 Practice problems - CRUD ***

Problem 1 (Create a database and a table)
<Link to the solution>

Let's build the database for a running app, where we want to store individual runner's information. Create a database called `running_app` and a table in that database called `runners`. Each runner should have a unique id which is auto incremented, a unique username (cannot be null), and an email (cannot be null). We also want to store each runner's first name, last name (cannot be null), gender (as M or F), and date of birth. Besides, we want to keep a record of the total distance in miles for each runner (default value is 0 and it can have arbitrary decimal precision), and the date and time of last activity.

Once you have created the table show its structure - it should look something like this.

```
+---------------+--------------+------+-----+---------+----------------+
| Field         | Type         | Null | Key | Default | Extra          |
+---------------+--------------+------+-----+---------+----------------+
| id            | int(11)      | NO   | PRI | NULL    | auto_increment |
| username      | varchar(255) | NO   | UNI | NULL    |                |
| email         | varchar(255) | NO   |     | NULL    |                |
| first_name    | varchar(255) | YES  |     | NULL    |                |
| last_name     | varchar(255) | NO   |     | NULL    |                |
| gender        | char(1)      | YES  |     | NULL    |                |
| dob           | date         | YES  |     | NULL    |                |
| total_miles   | double       | YES  |     | 0       |                |
| last_activity | datetime     | YES  |     | NULL    |                |
+---------------+--------------+------+-----+---------+----------------+
```

Hint:

- We are asked to store the gender as M (for male) or F (for female). So you could use `CHAR(1)` (fixed 1 char long) to save on memory.

Problem 2 (Insert data into the table)
<Link to the solution>

Insert the following runners' data into the table you just created (see Problem 1).

Username	Email	First Name	Last Name
dsmith	dora@gmail.com	Dora	Smith
ron123	ron123@yahoo.com	Ron	Long
lisaaa	lisaroy@comcast.net	Lisa	Roy
tlong	tlong@gmail.com	Tara	Long

After you insert the data and select all rows it should look something like this.

```
+----+----------+---------------------+------------+-----------+--------+------+-------------+---------------+
| id | username | email               | first_name | last_name | gender | dob  | total_miles | last_activity |
+----+----------+---------------------+------------+-----------+--------+------+-------------+---------------+
|  1 | dsmith   | dora@gmail.com      | Dora       | Smith     | NULL   | NULL |           0 | NULL          |
|  2 | ron123   | ron123@yahoo.com    | Ron        | Long      | NULL   | NULL |           0 | NULL          |
|  3 | lisaaa   | lisaroy@comcast.net | Lisa       | Roy       | NULL   | NULL |           0 | NULL          |
|  4 | tlong    | tlong@gmail.com     | Tara       | Long      | NULL   | NULL |           0 | NULL          |
+----+----------+---------------------+------------+-----------+--------+------+-------------+---------------+
```

Hint:

- Remember, you have to enclose the string values within quotes!

Problem 3 (Update the table)

<Link to the solution>

Now let's update some of the runners' data in the table (see Problem 2).

- Ron was born on July 10th, 1985. Update his date of birth.
- Update `dsmith` and `lisaaa`'s gender to Female.
- Everyone just completed a 5k running event, so update their total miles.

After you make all the updates it should look something like this.

```
+----+----------+---------------------+------------+-----------+--------+------------+-------------+---------------+
| id | username | email               | first_name | last_name | gender | dob        | total_miles | last_activity |
+----+----------+---------------------+------------+-----------+--------+------------+-------------+---------------+
|  1 | dsmith   | dora@gmail.com      | Dora       | Smith     | F      | NULL       |         3.1 | NULL          |
|  2 | ron123   | ron123@yahoo.com    | Ron        | Long      | NULL   | 1985-07-10 |         3.1 | NULL          |
|  3 | lisaaa   | lisaroy@comcast.net | Lisa       | Roy       | F      | NULL       |         3.1 | NULL          |
|  4 | tlong    | tlong@gmail.com     | Tara       | Long      | NULL   | NULL       |         3.1 | NULL          |
+----+----------+---------------------+------------+-----------+--------+------------+-------------+---------------+
```

Hint:

- Remember, if you have used `CHAR(1)` for the gender type, you have to specify the gender value in a single character only as 'M' or 'F'. If you specify the gender as 'Male' or 'Female' instead, it will give error.
- 5k run is in km, which is about 3.1 miles.

Problem 4 (Delete some records)

<Link to the solution>

Runners with the last name 'Long' (see Problem 3) decided to leave our app :(Let's delete their records.

After the delete it should look something like this.

```
+----+----------+---------------------+------------+-----------+--------+------+-------------+---------------+
| id | username | email               | first_name | last_name | gender | dob  | total_miles | last_activity |
+----+----------+---------------------+------------+-----------+--------+------+-------------+---------------+
|  1 | dsmith   | dora@gmail.com      | Dora       | Smith     | F      | NULL |         3.1 | NULL          |
|  3 | lisaaa   | lisaroy@comcast.net | Lisa       | Roy       | F      | NULL |         3.1 | NULL          |
+----+----------+---------------------+------------+-----------+--------+------+-------------+---------------+
```

Chapter 4

Refining selections

4.1 Functions

You will be using these functions a lot to manipulate and transform the data. We will take a look at some of the commonly used functions here that will cover most of your needs. But know that if you think you need a function that isn't mentioned here, search for it[1] before even thinking of making your own.

4.1.1 Numeric functions

Table 4.1 shows a list of commonly used numeric functions, loosely grouped based on similarity.

Let's see how the functions in Table 4.1 work with some examples.

```
1  SELECT <numeric function> → Sample result → Explanation
2  ──────────────────────────────────────────────────────
3  SELECT ABS(-5.3), ABS(5.3); → 5.3, 5.3
4  → Absolute value.
5
6  SELECT SIGN(-1.2), SIGN(1.2), SIGN(0); → -1, 1, 0
7  → Sign of a number.
8  ──────────────────────────────────
9
10 SELECT SQRT(9); → 3
11 → Square root of a number. SQRT(x) = √x
12
13 SELECT EXP(1); → 2.718281828459045
14 → EXP(x) = e^x
15
16 SELECT POWER(2, 3); → 8
17 → POWER(x,y) = x^y
18 ──────────────────────────────────
19
20 SELECT LOG(2, 8); → 3
```

[1] Remember, Google search is your best friend while coding! Also it's an extremely valuable skill to have.

```
1  SELECT ABS(-5.3), ABS(5.3);
2  SELECT SIGN(-1.2), SIGN(1.2), SIGN(0);
3
4  SELECT SQRT(9);
5  SELECT EXP(1);
6  SELECT POWER(2, 3);
7
8  SELECT LOG(2, 8);
9  SELECT LOG2(8);
10 SELECT LN(EXP(2));
11 SELECT LOG10(100);
12 SELECT PI();
13
14 SELECT DEGREES(PI());
15 SELECT RADIANS(180)/PI();
16 SELECT SIN(PI()/2);
17 SELECT ASIN(1)/(PI()/2);
18 -- similarly COS, ACOS, TAN, ATAN, and COT
19
20 SELECT LEAST(1, -9, 5);
21 SELECT GREATEST(1, -9, 5);
22
23 SELECT ROUND(185.385, 2), ..., ROUND(-185.385, -2);
24 SELECT TRUNCATE(185.385, 2), ..., TRUNCATE(-185.385, -2);
25 SELECT FLOOR(1.0), ..., FLOOR(-2.6);
26 SELECT CEIL(1.0), ..., CEIL(-2.6);
27
28 SELECT RAND();
29 SELECT RAND(100);
30
31 --
32 SELECT 17 DIV 6, ..., -17 DIV -6;
33 SELECT MOD(17, 6), ..., -17 % -6; -- MOD is also written as %
```

Table 4.1: SQL numeric functions

```
→ Log of 8 with base 2. LOG(x, y) = log_x y

LOG2(8); → 3
→ Log of 8 with base 2. LOG2(y) = log_2 y

SELECT LN(EXP(2)); → 2
→ Natural log. LN(y) = log_e y := ln y

SELECT LOG10(100); → 2
→ Log of 100 with base 10. LOG10(y) = log_10 y

SELECT PI(); → 3.141593
→ The math constant π.
-----------------------------------

SELECT DEGREES(PI()); → 180
→ Convert to degrees.

SELECT RADIANS(180)/PI(); → 1
→ Convert to radians.

SELECT SIN(PI()/2); → 1
→ Sine of an angle (in radians).

SELECT ASIN(1)/(PI()/2); → 1
→ Arc sine of a number.

-- similarly COS, ACOS, TAN, ATAN, and COT
-----------------------------------

SELECT LEAST(1, -9, 5); → -9
→ Min of the numbers.

SELECT GREATEST(1, -9, 5); → 5
→ Max of the numbers.
-----------------------------------

SELECT ROUND(185.385, 2), ROUND(-185.385, 2), ROUND(185.385, -2), ROUND(-185.385, -2);
→ 185.39, -185.39, 200, -200
→ Round to the given decimal points. Notice when the second argument is negative, it makes sense!

SELECT TRUNCATE(185.385, 2), TRUNCATE(-185.385, 2), TRUNCATE(185.385, -2), TRUNCATE
    (-185.385, -2);
→ 185.38, -185.38, 100, -100
→ Truncate to the given decimal points. Notice when the second argument is negative, it makes
    sense!

SELECT FLOOR(1.0), FLOOR(1.49), FLOOR(1.5), FLOOR(1.6), FLOOR(-2.0), FLOOR(-2.49),
    FLOOR(-2.5), FLOOR(-2.6);
→ 1, 1, 1, 1, -2, -3, -3, -3
→ Take the floor (meaning, largest integer <= the number).

SELECT CEIL(1.0), CEIL(1.49), CEIL(1.5), CEIL(1.6), CEIL(-2.0), CEIL(-2.49), CEIL
    (-2.5), CEIL(-2.6);
→ 1, 2, 2, 2, -2, -2, -2, -2
→ Take the ceiling (meaning, smallest integer >= the number).
-----------------------------------

SELECT RAND(), RAND(), RAND();
```

```
→ 0.7307410777832763, 0.8740376726482414, 0.177965160625023
→ Random numbers in [0, 1), meaning between 0 (incl.) and 1 (excl.). Here we have run the RAND() function thrice, and got a different number each time. If I run again I will get yet different numbers. Your result will differ because it is random!

SELECT RAND(100), RAND(100), RAND(100), RAND(200), RAND(200);
→ 0.17353134804734155, 0.17353134804734155, 0.17353134804734155, 0.19184226839974733, 0.19184226839974733
→ Reproducible random numbers in [0, 1) using a seed. You will get the same number if you use the same seed.
```

All of the above functions are self explanatory. However, we left out couple of functions (`DIV` and `MOD`) that require some explanation. Let's take a look at those next.

The `DIV` gives the quotient, while `MOD` gives the remainder in an integer division. For example, 17 ÷ 6 = 2 remainder 5, so `17 DIV 6 = 2` and `MOD(17, 6) = 5`. But things get tricky for negative integers. On top of that, in some cases SQL gives a different result than Python! While you will rarely encounter negative numbers in `DIV` and `MOD`, it is nice to know how they work.

Lets's first compare the SQL and Python results. We will use the Python notation here for simplicity (17 DIV 6 is written as `17//6`, and MOD(17, 6) is written as `17%6`).

```
i DIV j             SQL           Python
_____________________________________________
17 // 6             2              2
-17 // 6           -2             -3 ← notice the difference SQL vs. Python
17 // -6           -2             -3 ←
-17 // -6           2              2

MOD(i, j)           SQL           Python
_____________________________________________
17 % 6               5              5
-17 % 6             -5              1 ← notice the difference SQL vs. Python
17 % -6              5             -1 ←
-17 % -6            -5             -5
```

When both the numbers are positive integers, the result is intuitive and makes sense. But how do you explain the result when one or both the numbers are negative, also the discrepancy between SQL and Python?

The explanation involves two steps:

1. The SQL DIV is straightforward - do a normal division and drop the decimal part. In other words, SQL uses TRUNCATE. On the other hand, Python uses FLOOR for the DIV - do a normal division and take the FLOOR. This is where SQL and Python differ.

2. Finally, the DIV and MOD obey the quotient and remainder rule of division - i.e., if $a/b =$ quotient q and remainder r, then $q*b+r=a$. And thus, $(i//j)*j+(i\%j)=i$. From this we get the MOD since we already know the DIV from the first step. This works for both SQL and Python, and also explains the sign of the MOD values.

This explanation is verified as follows:

```
i DIV j       SQL     Python    SQL, TRUNCATE(i/j, 0)    Python, math.floor(i/j)
_____________________________________________________________________________
17 // 6        2       2         2                         2
-17 // 6      -2      -3        -2                        -3
17 // -6      -2      -3        -2                        -3
```

```
-17 // -6      2       2         2                          2

MOD(i, j)     SQL     Python.          (i//j)*j + (i%j): SQL,   Python
_____________________________________________________________________
17 % 6         5       5                                  17     17
-17 % 6       -5       1                                 -17    -17
17 % -6        5      -1                                  17     17
-17 % -6      -5      -5                                 -17    -17
```

Once again, you will seldom need to do DIV and MOD with negative integers, but if you ever do, now you understand how it works (and the nuances of SQL vs Python).

4.1.2 Aggregate functions

Table 4.2 lists some commonly used aggregate functions.

```sql
CREATE TABLE numbers (num INT);
INSERT INTO numbers (num) VALUES (1), (9), (5);

SELECT COUNT(num) FROM numbers;
SELECT MIN(num) FROM numbers;
SELECT MAX(num) FROM numbers;
SELECT SUM(num) FROM numbers;
SELECT AVG(num) FROM numbers;
```

Table 4.2: SQL aggregate functions

Let's see how these functions in Table 4.2 work with an example. The results are self explanatory.

```
SELECT <numeric function> → Sample result
_____________________________________________________________
CREATE TABLE numbers (num INT);
INSERT INTO numbers (num) VALUES (1), (9), (5);

SELECT COUNT(num) FROM numbers; → 3
SELECT MIN(num) FROM numbers; → 1
SELECT MAX(num) FROM numbers; → 9
SELECT SUM(num) FROM numbers; → 15
SELECT AVG(num) FROM numbers; → 5.0000
```

These aggregate functions are often used in conjunction with some grouping techniques (we will learn about those shortly in Sec. 4.4).

4.1.3 String functions

Table 4.3 lists some commonly used string functions, loosely grouped based on similarity.

Let's see how the functions in Table 4.3 work with some examples.

```
SELECT <numeric function> → Sample result → Explanation
_____________________________________________________________
SELECT CONCAT('abc', 'def', 'g'); → 'abcdefg'
→ Concat the strings.

```

```sql
SELECT CONCAT('abc', 'def', 'g');
SELECT CONCAT_WS('-', 'abc', 'def', 'g'); -- WS means with separator

SELECT SUBSTRING('123456789', 2, 4);
SELECT SUBSTRING('123456789', 6);
SELECT SUBSTRING('123456789', -3);
SUBSTRING_INDEX(str, delimiter, count); → See the solution to Problem 9

SELECT REPLACE('Hello WorLd', 'l', '*');
SELECT REVERSE('12 345');

SELECT UPPER('Hello World');
SELECT LOWER('Hello World');

SELECT CHAR_LENGTH('12345 67');

SELECT LTRIM('  abc  ');
SELECT RTRIM('  abc  ');
SELECT TRIM('  abc  ');
```

Table 4.3: SQL string functions

```sql
SELECT CONCAT_WS('-', 'abc', 'def', 'g'); → 'abc-def-g'
→ Concat with separator (first argument is the separator).

SELECT SUBSTRING('123456789', 2, 4); → '2345'
→ Substring starting form index 2, and get total of 4 chars. Indexing starts from 1 (unlike most
    programming languages where indexing starts from 0).

SELECT SUBSTRING('123456789', 6); → '6789'
→ Substring starting form index 6, and until the end.

SELECT SUBSTRING('123456789', -3); → '789'
→ Substring starting form position 3 counting from the end, and until the end.

SELECT REPLACE('Hello WorLd', 'l', '*'); → 'He**o WorLd'
→ Replace all occurrences of 'l' with '*' (case sensitive).

SELECT REVERSE('12 345'); → '543 21'
→ Reverse the string.

SELECT UPPER('Hello World'); → 'HELLO WORLD'
→ Convert the string to all uppercase.

SELECT LOWER('Hello World'); → 'hello world'
→ Convert the string to all lowercase.

SELECT CHAR_LENGTH('12345 67'); → 8
→ Count the number of chars.

SELECT LTRIM('  abc  '); → 'abc ' → Trim whitespaces from left.
SELECT RTRIM('  abc  '); → ' abc' → Trim whitespaces from right.
SELECT TRIM('  abc  '); → 'abc'→ Trim whitespaces from both left and right.
```

4.1.4 Date functions

Table 4.4 shows a list of commonly used date functions, loosely grouped based on similarity.

```
-- A more comprehensive list of date functions available at
-- https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html
SELECT CURDATE();
SELECT CURTIME();
SELECT CURRENT_TIMESTAMP, NOW();

SELECT YEAR('2020-02-10 18:24:37');
SELECT MONTH('2020-02-10 18:24:37');
SELECT DAY('2020-02-10 18:24:37');
SELECT HOUR('2020-02-10 18:24:37');2020-02-10
SELECT MINUTE('2020-02-10 18:24:37');
SELECT SECOND('2020-02-10 18:24:37');
SELECT MICROSECOND('2020-02-10 18:24:37.000028');
SELECT DATE('2020-02-10 18:24:37.000028');
SELECT TIME('2020-02-10 18:24:37.000028');

SELECT MONTHNAME('2020-02-10 18:24:37');
SELECT DAYNAME('2020-02-10');
SELECT DAYOFWEEK('2020-02-10');
SELECT DAYOFMONTH('2020-02-10');
SELECT DAYOFYEAR('2020-02-10');

-- DATE_FORMAT(date, format)
-- No need to remember the formats, just look it up at
-- https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function_date-
    format
SELECT DATE_FORMAT('2020-02-10', '%M %D %Y');
SELECT DATE_FORMAT('2020-02-10', '%M %d, %Y');
SELECT DATE_FORMAT('2020-02-10 18:24:37', '%W, %M %d %Y at %T');

SELECT DATE_ADD('2020-02-10 18:24:37', INTERVAL 40 DAY);
SELECT DATE_SUB('2020-02-10 18:24:37', INTERVAL 5 MICROSECOND);
SELECT DATEDIFF('2020-02-10 18:24:37', '2020-03-10 00:00:00');
SELECT TIMEDIFF('2020-02-10 18:24:37', '2020-02-10 00:00:00');
```

Table 4.4: SQL date functions

Let's see how the functions in Table 4.4 work with some examples.

```
SELECT <numeric function> → Sample result → Explanation
_____________________________________________________________
-- A more comprehensive list of date functions available at
-- https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html
SELECT CURDATE(); → 2020-04-29 → Current date when the command is run.
SELECT CURTIME(); → 00:15:29 → Current time when the command is run.
SELECT CURRENT_TIMESTAMP, NOW(); → 2020-04-29 00:16:04, 2020-04-29 00:16:04
→ Current date and time when the command is run.

SELECT YEAR('2020-02-10 18:24:37'); → 2010 → Extract the year.
SELECT MONTH('2020-02-10 18:24:37'); → 2 → Extract the month.
SELECT DAY('2020-02-10 18:24:37'); → 10 → Extract the day.
SELECT HOUR('2020-02-10 18:24:37'); → 18 → Extract the hour.
SELECT MINUTE('2020-02-10 18:24:37'); → 24 → Extract the minute.
SELECT SECOND('2020-02-10 18:24:37'); → 37 → Extract the second.
```

```
SELECT MICROSECOND('2020-02-10 18:24:37.000028'); → 28 → Extract the microsecond.
SELECT DATE('2020-02-10 18:24:37.000028'); → 2020-02-10 → Extract the date.
SELECT TIME('2020-02-10 18:24:37.000028'); → 18:24:37.000028 → Extract the time.

SELECT MONTHNAME('2020-02-10'); → February → Extract the month name.
SELECT DAYNAME('2020-02-10'); → Monday → Extract the day name.
SELECT DAYOFWEEK('2020-02-10'), DAYOFMONTH('2020-02-10'), DAYOFYEAR('2020-02-10');
→ 2, 10, 41
→ Extract the day of week (starts from Sunday), day of month, and day of year.

-- DATE_FORMAT(date, format)
-- No need to remember the formats, just look it up at
-- https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function_date-
    format
SELECT DATE_FORMAT('2020-02-10', '%M %D %Y'); → February 10th 2020
SELECT DATE_FORMAT('2020-02-10', '%M %d, %Y'); → February 10, 2020
SELECT DATE_FORMAT('2020-02-10 18:24:37', '%W, %M %d %Y at %T');
→ Monday, February 10 2020 at 18:24:37

SELECT DATE_ADD('2020-02-10 18:24:37', INTERVAL 40 DAY);
→ 2020-03-21 18:24:37
SELECT DATE_SUB('2020-02-10 18:24:37', INTERVAL 5 MICROSECOND);
→ 2020-02-10 18:24:36.999995
SELECT DATEDIFF('2020-02-10 18:24:37', '2020-03-10 00:00:00'); → -29
SELECT TIMEDIFF('2020-02-10 18:24:37', '2020-02-10 00:00:00'); → 18:24:37
```

4.1.5 Other functions

Table 4.5 shows a few other commonly used functions.

```
SELECT IF(1<2, 'yes', 'no');
SELECT IFNULL(NULL, '0'), IFNULL(1, '0');
SELECT ISNULL(NULL), ISNULL(1);
SELECT CAST('2020-02-10' AS DATETIME);
```

Table 4.5: SQL other functions

Let's see how the functions in Table 4.5 work with some examples.

```
SELECT <numeric function> → Sample result → Explanation
_____________________________________________________________
SELECT IF(1<2, 'yes', 'no'); → yes
→ First argument is the condition to check. If true, return the 2nd argument, else return the 3rd
    argument.

SELECT IFNULL(NULL, '0'), IFNULL(1, '0'); → 0, 1
→ Check the first argument. If null, return the 2nd argument, else return the 1st argument.

SELECT ISNULL(NULL), ISNULL(1); → 1, 0
→ Is the argument null? If true, return 1 (= true), else 0 (= false).

SELECT CAST('2020-02-10' AS DATETIME); → 2020-02-10 00:00:00
→ CAST(expression AS datatype)
→ For permitted data types see:
    https://stackoverflow.com/questions/12126991/cast-from-varchar-to-int-mysql
```

4.2 Operators

Next we will look into some of the commonly used operators, see Table 4.6.

Operator	Description
Arithmetic Operators	
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo
Comparison Operators	
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>, or !=	Not equal to
Logical Operators	
AND, or &&	TRUE if all the conditions are TRUE, FALSE otherwise
OR, or ||	TRUE if any of the conditions is TRUE, FALSE otherwise
NOT	Negates a boolean value, e.g., NOT TRUE = FALSE, or 0
IN	TRUE if the operand is in the given list.
BETWEEN	TRUE if the operand is within the given range (inclusive).
LIKE	TRUE if the operand matches the given pattern. **Wildcards**: % ⇒ ⩾ 0 chars; _ ⇒ 1 char Use the back slash (\) to escape any wildcard.

Table 4.6: SQL operators

Let's take a look at these operators through some examples.

```
Arithmetic Operators

mysql> SELECT 5 + 3;
+-------+
| 5 + 3 |
+-------+
|     8 |
+-------+
1 row in set (0.00 sec)

mysql> SELECT 5 - 3;
+-------+
| 5 - 3 |
+-------+
|     2 |
+-------+
1 row in set (0.00 sec)

mysql> SELECT 5 * 3;
+-------+
| 5 * 3 |
+-------+
|    15 |
+-------+
1 row in set (0.00 sec)

mysql> SELECT 5 / 3;
+--------+
| 5 / 3  |
```

```
+--------+
| 1.6667 |
+--------+
1 row in set (0.00 sec)

mysql> SELECT 5 % 3;
+-------+
| 5 % 3 |
+-------+
|     2 | ← 5/3 = quotient 1, remainder 2. See also the DIV and MOD functions in Section 4.1.1.
+-------+
1 row in set (0.00 sec)

Comparison Operators
--------------------------------------------------------------------------------
mysql> SELECT 5 = 3;
+-------+
| 5 = 3 |
+-------+
|     0 | ← Remember, boolean values are represented as 1 (for TRUE) and 0 (for FALSE). See the example for BOOL type in Section 3.2.
+-------+
1 row in set (0.00 sec)

mysql> SELECT 5 > 3;
+-------+
| 5 > 3 |
+-------+
|     1 |
+-------+
1 row in set (0.00 sec)

mysql> SELECT 5 < 3;
+-------+
| 5 < 3 |
+-------+
|     0 |
+-------+
1 row in set (0.00 sec)

mysql> SELECT 5 >= 3;
+--------+
| 5 >= 3 |
+--------+
|      1 |
+--------+
1 row in set (0.00 sec)

mysql> SELECT 5 <= 3;
+--------+
| 5 <= 3 |
+--------+
|      0 |
+--------+
1 row in set (0.00 sec)

mysql> SELECT 5 != 3;
+--------+
| 5 != 3 |
+--------+
|      1 |
+--------+
1 row in set (0.00 sec)

Logical Operators
--------------------------------------------------------------------------------
mysql> SELECT TRUE AND TRUE, TRUE AND FALSE;
+---------------+----------------+
| TRUE AND TRUE | TRUE AND FALSE |
+---------------+----------------+
|             1 |              0 |
+---------------+----------------+
1 row in set (0.00 sec)

mysql> SELECT TRUE OR FALSE, FALSE OR FALSE;
+---------------+----------------+
| TRUE OR FALSE | FALSE OR FALSE |
+---------------+----------------+
|             1 |              0 |
+---------------+----------------+
1 row in set (0.00 sec)

mysql> SELECT NOT TRUE, NOT 0;
+----------+-------+
| NOT TRUE | NOT 0 |
+----------+-------+
|        0 |     1 |
+----------+-------+
1 row in set (0.00 sec)

mysql>

We will use the following table to demo the other logical operators.
```

```
mysql> SELECT * FROM friends;
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  1 | Dora       | Smith     |   42 |
|  2 | Ron        | Long      |   35 |
|  3 | Lisa       | Roy       |   29 |
|  4 | Dora       | G.        |   21 |
|  5 | Tara       | Long      |   29 |
|  6 | Ryan       | Davis     |   40 |
+----+------------+-----------+------+
6 rows in set (0.00 sec)

mysql> SELECT * FROM friends
    -> WHERE last_name IN ('Long', 'Roy', 'Kim'); ← Select the table rows that have last_name Long, Roy, or Kim.
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  2 | Ron        | Long      |   35 |
|  3 | Lisa       | Roy       |   29 |
|  5 | Tara       | Long      |   29 |
+----+------------+-----------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM friends
    -> WHERE age BETWEEN 25 AND 35; ← Select the table rows that have age between 25 and 35 (inclusive).
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  2 | Ron        | Long      |   35 |
|  3 | Lisa       | Roy       |   29 |
|  5 | Tara       | Long      |   29 |
+----+------------+-----------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM friends
    -> WHERE age NOT BETWEEN 25 AND 35; ← Note the NOT.
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  1 | Dora       | Smith     |   42 |
|  4 | Dora       | G.        |   21 |
|  6 | Ryan       | Davis     |   40 |
+----+------------+-----------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM friends
    -> WHERE first_name LIKE '%ra%'; ← The first_name should contain 'ra'.
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  1 | Dora       | Smith     |   42 |
|  4 | Dora       | G.        |   21 |
|  5 | Tara       | Long      |   29 |
+----+------------+-----------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM friends
    -> WHERE last_name LIKE '_o%'; ← The last_name should contain the character 'o' at the 2nd place.
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  2 | Ron        | Long      |   35 |
|  3 | Lisa       | Roy       |   29 |
|  5 | Tara       | Long      |   29 |
+----+------------+-----------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM friends
    -> WHERE age LIKE '_9'; ← Also works with numbers.
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  3 | Lisa       | Roy       |   29 |
|  5 | Tara       | Long      |   29 |
+----+------------+-----------+------+
2 rows in set (0.00 sec)

mysql> SELECT 029.378 LIKE '__.%'; ← Checking for two digits before the decimal point. Note that it ignores the leading zero!
+---------------------+
| 029.378 LIKE '__.%' |
+---------------------+
|                   1 |
+---------------------+
1 row in set (0.00 sec)

mysql> SELECT '%' LIKE '%\%%'; ← Escaping the % with a \.
+-----------------+
| '%' LIKE '%\%%' |
+-----------------+
|               1 |
+-----------------+
```

```
1 row in set (0.00 sec)

mysql> SELECT '_' LIKE '%\_%'; ← Escaping the _ with a \.
+-----------------+
| '_' LIKE '%\_%' |
+-----------------+
|               1 |
+-----------------+
1 row in set (0.00 sec)

mysql> SELECT '(123)456-7890' LIKE '(___)___-____'; ← Checking phone number formatting.
+--------------------------------------+
| '(123)456-7890' LIKE '(___)___-____' |
+--------------------------------------+
|                                    1 |
+--------------------------------------+
1 row in set (0.00 sec)

mysql>
```

4.3 Branching

There will be times when you want (to do) different things depending on different conditions. In SQL you can accomplish this using the `CASE` statement. See Table 4.7 for the syntax.

```
CASE
  WHEN condition1 THEN result1
  [WHEN condition2 THEN result2] -- optional
  [...]
  [ELSE default_result] -- optional
END
```

Table 4.7: Branching with `CASE`

Let's take a look how to use the `CASE` statement with some examples.

```
Arithmetic Operators
--------------------------------------------------------------------------------

mysql> SELECT
    -> CASE
    ->   WHEN FALSE THEN 'here'
    -> END AS case1; ← Using alias (see Table 3.3).
+-------+
| case1 |
+-------+
| NULL  | ← Because no condition was satisfied and there was no default, it returned NULL.
+-------+
1 row in set (0.00 sec)

mysql>
mysql> SELECT
    -> CASE
    ->   WHEN FALSE THEN 'here 1'
    ->   WHEN FALSE THEN 'here 2'
    ->   WHEN TRUE THEN 'here 3'
    ->   WHEN TRUE THEN 'here 4'
    -> END AS case2;
+--------+
| case2  |
+--------+
| here 3 | ← It checks the conditions from top to bottom.
+--------+
1 row in set (0.00 sec)

mysql>
mysql> SELECT
    -> CASE
    ->   WHEN FALSE THEN 'here 1'
    ->   WHEN FALSE THEN 'here 2'
    ->   ELSE 'default_here'
    -> END AS case3;
+--------------+
| case3        |
+--------------+
| default_here | ← No condition was true, but there was a default.
+--------------+
1 row in set (0.00 sec)
```

```
Some more examples using a table.
mysql> SELECT * FROM friends;
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  1 | Dora       | Smith     |   42 |
|  2 | Ron        | Long      |   35 |
|  3 | Lisa       | Roy       |   29 |
|  4 | Dora       | G.        |   21 |
|  5 | Tara       | Long      |   29 |
|  6 | Ryan       | Davis     |   40 |
+----+------------+-----------+------+
6 rows in set (0.00 sec)

mysql>
mysql> SELECT
    ->   CASE
    ->     WHEN age < 30 THEN CONCAT(first_name, ' ***')
    ->     ELSE CONCAT('*** ', last_name)
    ->   END AS name,
    ->   age
    -> FROM friends;
+-----------+------+
| name      | age  |
+-----------+------+
| *** Smith |   42 |
| *** Long  |   35 |
| Lisa ***  |   29 |
| Dora ***  |   21 |
| Tara ***  |   29 |
| *** Davis |   40 |
+-----------+------+
6 rows in set (0.00 sec)

mysql>
mysql> SELECT * FROM friends
    -> ORDER BY
    ->   (CASE
    ->     WHEN age < 30 THEN first_name
    ->     ELSE last_name
    ->   END);
+----+------------+-----------+------+
| id | first_name | last_name | age  |← Based on the CASE it has sorted as follows.
+----+------------+-----------+------+
|  6 | Ryan       | Davis     |   40 | ← Davis
|  4 | Dora       | G.        |   21 | ← Dora
|  3 | Lisa       | Roy       |   29 | ← Lisa
|  2 | Ron        | Long      |   35 | ← Long
|  1 | Dora       | Smith     |   42 | ← Smith
|  5 | Tara       | Long      |   29 | ← Tara
+----+------------+-----------+------+
6 rows in set (0.00 sec)

mysql>
```

4.4 Grouping

Say you want to find out how many of your friends are in their 20s, 30s, etc. Or, you want to know the monthly sale volume from your online store. SQL offers couple of different ways to group the records of a table and compute useful statistics on them. One is the more commonly used `GROUP BY` and the other one is a bit more advanced window functions. We will look at both of them here.

4.4.1 GROUP BY ... HAVING

The `GROUP BY` syntax is more straightforward (Table 4.8), and you will see this a lot. It is often used with the aggregate functions (see Section 4.1.2).

Let's look at some examples of `GROUP BY`.

```
We will use our prervious friends table for the demo (see below).
mysql> SELECT * FROM friends;
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  1 | Dora       | Smith     |   42 |
|  2 | Ron        | Long      |   35 |
```

```
SELECT
  COUNT(*),
  AVG(column_name),
  aggregate_column_names,
  ...
FROM table_name
[WHERE conditions]
GROUP BY column_names
[HAVING group_conditions]
[ORDER BY column_names];
```

Table 4.8: GROUP BY ... HAVING

```
|  3 | Lisa       | Roy       |   29 |
|  4 | Dora       | G.        |   21 |
|  5 | Tara       | Long      |   29 |
|  6 | Ryan       | Davis     |   40 |
+----+------------+-----------+------+
6 rows in set (0.00 sec)

mysql> SELECT
    ->   TRUNCATE(age, -1) AS age_group, ← Remember how TRUNCATE works, see Section 4.1.1.
    ->   COUNT(*)
    -> FROM friends
    -> GROUP BY age_group; ← We are groping friends based on their age group as defined above.
+-----------+----------+
| age_group | COUNT(*) |
+-----------+----------+
|        40 |        2 |
|        30 |        1 |
|        20 |        3 |
+-----------+----------+ ← We have 2 friends in their 40s, 1 friend in their 30s, and 3 friends in their 20s (as you can easily verify from the
    friends table).
3 rows in set (0.00 sec)

mysql> SELECT
    ->   TRUNCATE(age, -1) AS age_group,
    ->   COUNT(*) AS no_of_friends
    -> FROM friends
    -> GROUP BY age_group
    -> ORDER BY no_of_friends; ← Same as the above grouping, but just ordering the result by the number of friends in that age group (in
        increasing order by default).
+-----------+---------------+
| age_group | no_of_friends |
+-----------+---------------+
|        30 |             1 |
|        40 |             2 |
|        20 |             3 |
+-----------+---------------+
3 rows in set (0.01 sec)

mysql> SELECT
    ->   TRUNCATE(age, -1) AS age_group,
    ->   COUNT(*) AS no_of_friends
    -> FROM friends
    -> WHERE age >= 30 ← Include only the friends who are 30 or older in the grouping.
    -> GROUP BY age_group
    -> ORDER BY no_of_friends;
+-----------+---------------+
| age_group | no_of_friends |
+-----------+---------------+
|        30 |             1 |
|        40 |             2 |
+-----------+---------------+ ← Notice that friends in their 20s are not included now.
2 rows in set (0.00 sec)

mysql> SELECT
    ->   TRUNCATE(age, -1) AS age_group,
    ->   COUNT(*) AS no_of_friends
    -> FROM friends
    -> WHERE last_name LIKE '___%' ← Ok, just for a demo, now we want to include only the friends who have 3 or more characters in their
        last name! See Section 4.2 for the LIKE operator.
    -> GROUP BY age_group ← Grouping by their age group as before.
    -> HAVING no_of_friends > 1 ← This time we want only the age groups HAVING more than 1 friend.
    -> ORDER BY age_group; ← Finally, order the results by the age group.
+-----------+---------------+
| age_group | no_of_friends |
+-----------+---------------+
|        20 |             2 |
|        40 |             2 |
+-----------+---------------+ ← Note - now the age group 20 has only 2 friends because of the WHERE condition, and the age group 30 does not
    show up because of the HAVING condition.
```

```
2 rows in set (0.00 sec)

mysql>
```

4.4.2 Window functions - OVER

Imagine you have a table containing total sale amount for each day, and you want to know how you did on a particular day compared to 7-days average centered around that day. This kind of analysis would be very challenging to do using the `GROUP BY` (Section 4.4.1), but would be straight forward using the window functions.

There are different types of window functions:

- Aggregate functions: COUNT(), MIN(), MAX(), SUM(), AVG(), ...
- Ranking functions: RANK(), DENSE_RANK(), ROW_NUMBER(), NTILE(), ...
- Value functions: LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE() ...

The syntax for the window functions is shown in Table 4.9.

```
window_function_name(expression) OVER (
  [partition_defintion] -- define how to partition the rows
  [order_definition] -- define how to order the rows within a partition
  [frame_definition] -- define the window of rows to apply the function
)

For example:
SELECT
  *,
  COUNT(age) OVER(
    PARTITION BY gender
    ORDER BY age
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS count_
FROM friends;
```

Table 4.9: Window functions syntax

Let's take a look at the aggregate type window functions with some examples, but keep in mind also the other types in case you need them.

```
Note, we have a new friends table that has a gender column added.
mysql> SELECT * FROM friends;
+----+------------+-----------+------+--------+
| id | first_name | last_name | age  | gender | ← Notice the gender column added.
+----+------------+-----------+------+--------+
|  1 | Dora       | Smith     |   42 | F      |
|  2 | Ron        | Long      |   35 | M      |
|  3 | Lisa       | Roy       |   29 | F      |
|  4 | Dora       | G.        |   21 | F      |
|  5 | Tara       | Long      |   29 | F      |
|  6 | Ryan       | Davis     |   40 | M      |
+----+------------+-----------+------+--------+
6 rows in set (0.00 sec)

mysql> SELECT
    ->   id,
    ->   CONCAT_WS(' ', first_name, last_name) AS name,
    ->   age,
    ->   gender,
    ->   COUNT(age) OVER() AS count_, ← The OVER clause with no arguments.
    ->   MIN(age) OVER() AS min_,
    ->   MAX(age) OVER() AS max_
    -> FROM friends;
+----+------------+------+--------+--------+------+------+
| id | name       | age  | gender | count_ | min_ | max_ |
```

```
+----+------------+------+--------+--------+------+------+
|  1 | Dora Smith |   42 | F      |      6 |   21 |   42 |
|  2 | Ron Long   |   35 | M      |      6 |   21 |   42 |
|  3 | Lisa Roy   |   29 | F      |      6 |   21 |   42 |
|  4 | Dora G.    |   21 | F      |      6 |   21 |   42 |
|  5 | Tara Long  |   29 | F      |      6 |   21 |   42 |
|  6 | Ryan Davis |   40 | M      |      6 |   21 |   42 |
+----+------------+------+--------+--------+------+------+ ← The OVER clause with no arguments takes all the rows as a single partition, preserves the pre-existing order of the rows, and has a single frame that includes all the rows from the first to the last for applying the window function.
6 rows in set (0.00 sec)

mysql> SELECT
    ->   id,
    ->   CONCAT_WS(' ', first_name, last_name) AS name,
    ->   age,
    ->   gender,
    ->   COUNT(age) OVER(PARTITION BY gender) AS count_, ← The OVER clause with only the PARTITION BY argument.
    ->   MIN(age) OVER(PARTITION BY gender) AS min_,
    ->   MAX(age) OVER(PARTITION BY gender) AS max_
    -> FROM friends;
+----+------------+------+--------+--------+------+------+
| id | name       | age  | gender | count_ | min_ | max_ |
+----+------------+------+--------+--------+------+------+
|  1 | Dora Smith |   42 | F      |      4 |   21 |   42 |
|  3 | Lisa Roy   |   29 | F      |      4 |   21 |   42 |
|  4 | Dora G.    |   21 | F      |      4 |   21 |   42 |
|  5 | Tara Long  |   29 | F      |      4 |   21 |   42 |
|  2 | Ron Long   |   35 | M      |      2 |   35 |   40 |
|  6 | Ryan Davis |   40 | M      |      2 |   35 |   40 |
+----+------------+------+--------+--------+------+------+ ← The OVER clause with only the PARTITION BY argument preserves the pre-existing order of the rows within a given partition, and has a single frame within a given partition that includes all the rows in that partition for applying the window function.
6 rows in set (0.00 sec)

mysql> SELECT
    ->   id,
    ->   CONCAT_WS(' ', first_name, last_name) AS name,
    ->   age,
    ->   gender,
    ->   COUNT(age) OVER(ORDER BY age) AS count_, ← The OVER clause with only the ORDER BY argument.
    ->   MIN(age) OVER(ORDER BY age) AS min_,
    ->   MAX(age) OVER(ORDER BY age) AS max_
    -> FROM friends;
+----+------------+------+--------+--------+------+------+
| id | name       | age  | gender | count_ | min_ | max_ |
+----+------------+------+--------+--------+------+------+
|  4 | Dora G.    |   21 | F      |      1 |   21 |   21 |
|  3 | Lisa Roy   |   29 | F      |      3 |   21 |   29 | ← Note the count is 3 here, instead of 2. See the explanation below.
|  5 | Tara Long  |   29 | F      |      3 |   21 |   29 |
|  2 | Ron Long   |   35 | M      |      4 |   21 |   35 |
|  6 | Ryan Davis |   40 | M      |      5 |   21 |   40 |
|  1 | Dora Smith |   42 | F      |      6 |   21 |   42 |
+----+------------+------+--------+--------+------+------+ ← Specifying ORDER BY without specifying ROWS BETWEEN takes rows from the first row to the last row corresponding to the current sorted value (instead of from the first row to the current row)!
6 rows in set (0.00 sec)

mysql> SELECT
    ->   id,
    ->   CONCAT_WS(' ', first_name, last_name) AS name,
    ->   age,
    ->   gender,
    ->   COUNT(age) OVER(PARTITION BY gender ORDER BY age) AS count_, ← Specifying ORDER BY without specifying ROWS BETWEEN.
    ->   MIN(age) OVER(PARTITION BY gender ORDER BY age) AS min_,
    ->   MAX(age) OVER(PARTITION BY gender ORDER BY age) AS max_
    -> FROM friends;
+----+------------+------+--------+--------+------+------+
| id | name       | age  | gender | count_ | min_ | max_ |
+----+------------+------+--------+--------+------+------+
|  4 | Dora G.    |   21 | F      |      1 |   21 |   21 |
|  3 | Lisa Roy   |   29 | F      |      3 |   21 |   29 | ← Again, note the count is 3 here, instead of 2.
|  5 | Tara Long  |   29 | F      |      3 |   21 |   29 |
|  1 | Dora Smith |   42 | F      |      4 |   21 |   42 |
|  2 | Ron Long   |   35 | M      |      1 |   35 |   35 |← Count etc. reset in a new partition.
|  6 | Ryan Davis |   40 | M      |      2 |   35 |   40 |
+----+------------+------+--------+--------+------+------+ ← Like noted before, specifying ORDER BY without specifying ROWS BETWEEN takes rows from the first row to the last row corresponding to the current sorted value, this time within a given partition.
6 rows in set (0.00 sec)

mysql> SELECT
    ->   id,
    ->   CONCAT_WS(' ', first_name, last_name) AS name,
    ->   age,
    ->   gender,
    ->   COUNT(age) OVER(ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS count_, ← We have to specify ROWS BETWEEN x AND y, where x and y are w.r.t. the current row. Here, x = UNBOUNDED PRECEDING meaning the first row, and y = UNBOUNDED FOLLOWING meaning the last row. See Fig. 4.2.
    ->   MIN(age) OVER(ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS min_,
    ->   MAX(age) OVER(ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS max_
    -> FROM friends;
+----+------------+------+--------+--------+------+------+
| id | name       | age  | gender | count_ | min_ | max_ |
+----+------------+------+--------+--------+------+------+
|  1 | Dora Smith |   42 | F      |      6 |   21 |   42 |
```

```
|  2 | Ron Long   |   35 | M      |      6 |   21 |   42 |
|  3 | Lisa Roy   |   29 | F      |      6 |   21 |   42 |
|  4 | Dora G.    |   21 | F      |      6 |   21 |   42 |
|  5 | Tara Long  |   29 | F      |      6 |   21 |   42 |
|  6 | Ryan Davis |   40 | M      |      6 |   21 |   42 |
+----+------------+------+--------+--------+------+------+
6 rows in set (0.00 sec)

mysql> SELECT
    ->   id,
    ->   CONCAT_WS(' ', first_name, last_name) AS name,
    ->   age,
    ->   gender,
    ->   COUNT(age) OVER(
    ->     PARTITION BY gender
    ->     ORDER BY age
    ->     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS count_, ← Here all the three arguments for the OVER clause are specified.
    ->   MIN(age) OVER(
    ->     PARTITION BY gender
    ->     ORDER BY age
    ->     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS min_,
    ->   MAX(age) OVER(
    ->     PARTITION BY gender
    ->     ORDER BY age
    ->     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS max_
    -> FROM friends;
+----+------------+------+--------+--------+------+------+
| id | name       | age  | gender | count_ | min_ | max_ |
+----+------------+------+--------+--------+------+------+
|  4 | Dora G.    |   21 | F      |      1 |   21 |   21 |
|  3 | Lisa Roy   |   29 | F      |      2 |   21 |   29 | ← Note the count is 2 here as per the specified ROWS BETWEEN.
|  5 | Tara Long  |   29 | F      |      3 |   21 |   29 |
|  1 | Dora Smith |   42 | F      |      4 |   21 |   42 |
|  2 | Ron Long   |   35 | M      |      1 |   35 |   35 |
|  6 | Ryan Davis |   40 | M      |      2 |   35 |   40 |
+----+------------+------+--------+--------+------+------+
6 rows in set (0.00 sec)

mysql> SELECT
    ->   id,
    ->   CONCAT_WS(' ', first_name, last_name) AS name,
    ->   age,
    ->   gender,
    ->   COUNT(age) OVER(
    ->     PARTITION BY gender
    ->     ORDER BY age ROWS
    ->     BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS count_, ← Another example where all the three arguments for the
    OVER clause are specified.
    ->   MIN(age) OVER(
    ->     PARTITION BY gender
    ->     ORDER BY age
    ->     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS min_,
    ->   MAX(age) OVER(
    ->     PARTITION BY gender
    ->     ORDER BY age
    ->     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS max_
    -> FROM friends;
+----+------------+------+--------+--------+------+------+
| id | name       | age  | gender | count_ | min_ | max_ |
+----+------------+------+--------+--------+------+------+
|  4 | Dora G.    |   21 | F      |      4 |   21 |   42 |
|  3 | Lisa Roy   |   29 | F      |      4 |   21 |   42 |
|  5 | Tara Long  |   29 | F      |      4 |   21 |   42 |
|  1 | Dora Smith |   42 | F      |      4 |   21 |   42 |
|  2 | Ron Long   |   35 | M      |      2 |   35 |   40 |
|  6 | Ryan Davis |   40 | M      |      2 |   35 |   40 |
+----+------------+------+--------+--------+------+------+
6 rows in set (0.00 sec)


Some more examples using the following table.
CREATE TABLE sales (
  day INT,
  sale DECIMAL(5,2),
  rainy CHAR(5)
);
INSERT INTO sales
  (day, sale, rainy)
VALUES
   (1, 10, 'yes'),
   (2, 30, 'no'),
   (3, 20, 'yes'),
   (4, 40, 'yes'),
   (5, 20, 'yes'),
   (6, 60, 'no');

mysql> SELECT * FROM sales;
+------+-------+-------+
| day  | sale  | rainy |
+------+-------+-------+
|    1 | 10.00 | yes   |
|    2 | 30.00 | no    |
```

```
|    3 | 20.00 | yes   |
|    4 | 40.00 | yes   |
|    5 | 20.00 | yes   |
|    6 | 60.00 | no    |
+------+-------+-------+
6 rows in set (0.00 sec)

mysql> SELECT
    ->   *,
    ->   COUNT(sale) OVER(ORDER BY day ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING) AS count_,
    ->   AVG(sale) OVER(ORDER BY day ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING) AS average_ ← Computing 5-days average, centered
          around the current day.
    -> FROM sales;
+------+-------+-------+--------+-----------+
| day  | sale  | rainy | count_ | average_  |
+------+-------+-------+--------+-----------+
|    1 | 10.00 | yes   |      3 | 20.000000 | ← Note, there are no preceding rows for the first row. And thus, count = 0 preceding + 1
       current + 2 following = 3-days.
|    2 | 30.00 | no    |      4 | 25.000000 | ← There is only 1 preceding row for the second row. And thus, count = 1 preceding + 1
       current + 2 following = 4-days.
|    3 | 20.00 | yes   |      5 | 24.000000 | ← Count = 2 preceding + 1 current + 2 following = 5-days.
|    4 | 40.00 | yes   |      5 | 34.000000 | ← Count = 2 preceding + 1 current + 2 following = 5-days.
|    5 | 20.00 | yes   |      4 | 35.000000 | ← Count = 2 preceding + 1 current + 1 following = 4-days.
|    6 | 60.00 | no    |      3 | 40.000000 | ← Count = 2 preceding + 1 current + 0 following = 3-days.
+------+-------+-------+--------+-----------+
6 rows in set (0.00 sec)

mysql> SELECT
    ->   *,
    ->   COUNT(sale) OVER(PARTITION BY rainy ORDER BY day ROWS BETWEEN 1 PRECEDING AND 0 FOLLOWING) AS count_,
    ->   AVG(sale) OVER(PARTITION BY rainy ORDER BY day ROWS BETWEEN 1 PRECEDING AND 0 FOLLOWING) AS average_
    -> FROM sales;
+------+-------+-------+--------+-----------+
| day  | sale  | rainy | count_ | average_  |
+------+-------+-------+--------+-----------+
|    2 | 30.00 | no    |      1 | 30.000000 |
|    6 | 60.00 | no    |      2 | 45.000000 |
|    1 | 10.00 | yes   |      1 | 10.000000 |
|    3 | 20.00 | yes   |      2 | 15.000000 |
|    4 | 40.00 | yes   |      2 | 30.000000 |
|    5 | 20.00 | yes   |      2 | 30.000000 |
+------+-------+-------+--------+-----------+ ← Note the table is now partitioned.
6 rows in set (0.00 sec)

mysql>
```

Fig. 4.1 shows a flows of the different arguments in the `OVER` clause (Table 4.9). Fig. 4.2 shows the convention for defining a window frame.

4.4.3 GROUP BY vs Window functions

As you might have already noticed, `GROUP BY` (4.4.1) reduces the number of result-rows to the number of final groups, while the window functions (4.4.2) compute the function values for every rows.

Window functions offer more flexibility, but I would just use the `GROUP BY` for simpler problems.

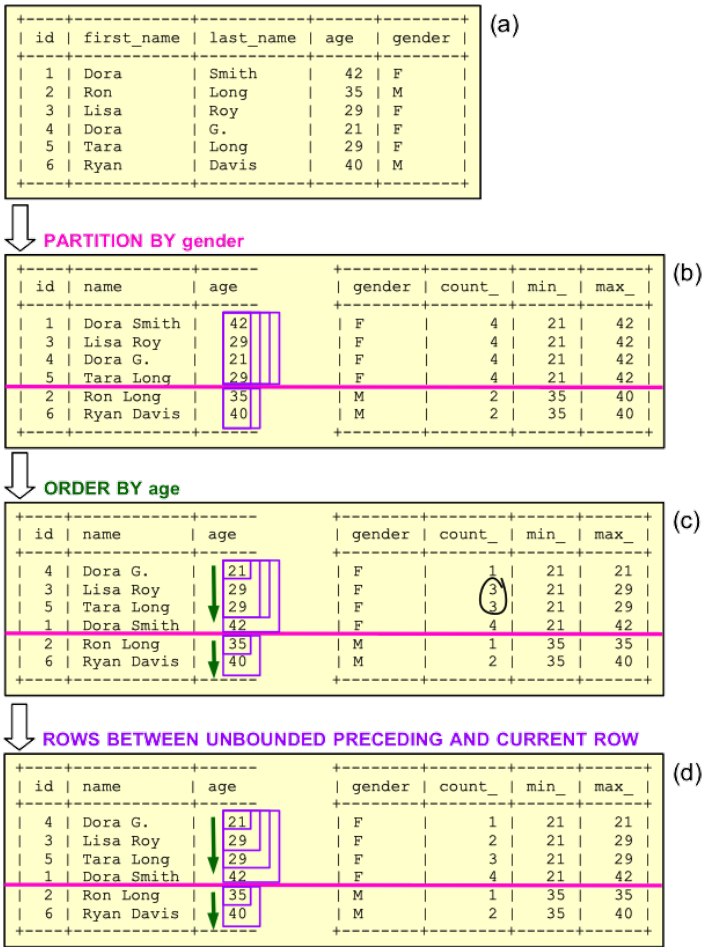


Fig. 4.1: Window functions arguments

We start with a table (a), partition by gender (b), order by age (c), and finally specify the frame (d). Note: Specifying `ORDER BY` without specifying `ROWS BETWEEN` (see (c)) takes rows from the first row to the last row corresponding to the **current sorted value** (instead of from the first row to the current row)!

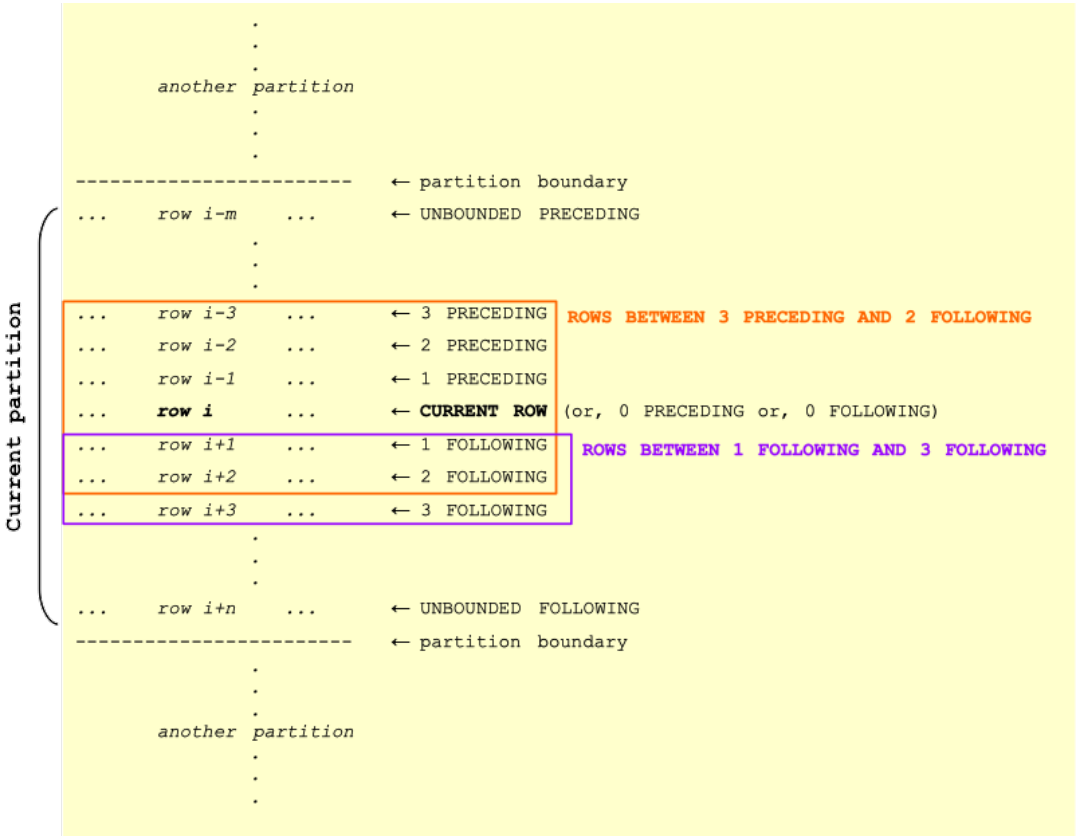


Fig. 4.2: ROWS BETWEEN conventions

`ROWS BETWEEN x AND y`: x = UNBOUNDED PRECEDING, CURRENT ROW, 3 PRECEDING, 1 FOLLOWING, etc., but not UNBOUNDED FOLLOWING. y = UNBOUNDED FOLLOWING, CURRENT ROW, 1 FOLLOWING, 3 PRECEDING, etc., but not UNBOUNDED PRECEDING. The frame starts from x **down to** y, and thus if row x happens to be below row y, count will be 0 and any aggregate result will be NULL.

4.5 Refining selections

We will learn a few more options here (see Table 4.10) that will help us refine the selections while querying a database.

```
-- DISTINCT
SELECT DISTINCT first_name, last_name ... -- distinct first_name AND last_name
SELECT COUNT(DISTINCT first_name, last_name) ...

-- ORDER BY
... ORDER BY last_name -- ASC is by default
... ORDER BY last_name ASC, 4 DESC

-- LIMIT
... LIMIT 3
... LIMIT 2, 3 -- 2 is the offset

-- WHERE
... WHERE age LIKE '3_'
```

Table 4.10: Refining selections

Let's take a look through some examples.

```
Let's use our friends table, as given below.
mysql> SELECT * FROM friends;
+----+------------+-----------+------+--------+
| id | first_name | last_name | age  | gender |
+----+------------+-----------+------+--------+
|  1 | Dora       | Smith     |   42 | F      |
|  2 | Ron        | Long      |   35 | M      |
|  3 | Lisa       | Roy       |   29 | F      |
|  4 | Dora       | G.        |   21 | F      |
|  5 | Tara       | Long      |   29 | F      |
|  6 | Ryan       | Davis     |   40 | M      |
|  7 | Ron        | long      |   30 | M      | ← Note, this is a new record added for the demo here.
+----+------------+-----------+------+--------+
7 rows in set (0.00 sec)

DISTINCT

mysql> SELECT DISTINCT
    ->   last_name ← Select only the unique last names.
    -> FROM friends;
+-----------+
| last_name |
+-----------+
| Smith     |
| Long      | ← Note that it ignores the case (e.g., it treats Long and long the same).
| Roy       |
| G.        |
| Davis     |
+-----------+
5 rows in set (0.00 sec)

mysql> SELECT DISTINCT
    ->   first_name, last_name ← The combination of first_name and last_name has to be unique (ignoring case).
    -> FROM friends;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Dora       | Smith     |
| Ron        | Long      | ← Note, 'Ron Long' and 'Ron long' are considered the same.
| Lisa       | Roy       |
| Dora       | G.        |
| Tara       | Long      | ← However, 'Ron Long' and 'Tara Long' are different (even though they have the same last name).
| Ryan       | Davis     |
+------------+-----------+
6 rows in set (0.00 sec)

mysql> SELECT
    ->   COUNT(DISTINCT first_name, last_name) ← Count the number of unique values (or unique combinations in this case).
    -> FROM friends;
+--------------------------------------+
| COUNT(DISTINCT first_name, last_name) |
+--------------------------------------+
```

```
|                                     6 | ← See the earlier query result to count.
+---------------------------------------+
1 row in set (0.00 sec)

ORDER BY
-------------------------------------------------------------------------------
mysql> SELECT
    ->   id,
    ->   first_name,
    ->   last_name,
    ->   age
    -> FROM friends
    -> ORDER BY last_name; ← Order by the last name (in ascending order by default).
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  6 | Ryan       | Davis     |   40 |
|  4 | Dora       | G.        |   21 |
|  2 | Ron        | Long      |   35 | ← In case of same last names (ignoring case), original row order is preserved.
|  5 | Tara       | Long      |   29 | ←
|  7 | Ron        | long      |   30 | ←
|  3 | Lisa       | Roy       |   29 |
|  1 | Dora       | Smith     |   42 |
+----+------------+-----------+------+
7 rows in set (0.00 sec)

mysql> SELECT
    ->   id, ← 1st select column
    ->   first_name, ← 2nd select column
    ->   last_name, ← 3rd select column
    ->   age ← 4th select column
    -> FROM friends
    -> ORDER BY last_name ASC, 4 DESC; ← First order by the last name (in ascending order), and then by the 4th select column (which is the age here) in descending order as specified. So you can specify the column name (or alias) or the select column number in the query. I would recommend specifying the column name (or alias) always for readability (plus, it is more robust if you rearrange the select columns order).
+----+------------+-----------+------+
| id | first_name | last_name | age  |
+----+------------+-----------+------+
|  6 | Ryan       | Davis     |   40 |
|  4 | Dora       | G.        |   21 |
|  2 | Ron        | Long      |   35 | ← Same last name, then order by the age (descending).
|  7 | Ron        | long      |   30 | ←
|  5 | Tara       | Long      |   29 | ←
|  3 | Lisa       | Roy       |   29 |
|  1 | Dora       | Smith     |   42 |
+----+------------+-----------+------+
7 rows in set (0.00 sec)

LIMIT
-------------------------------------------------------------------------------
mysql> SELECT *
    -> FROM friends
    -> LIMIT 3; ← Limit the selections to the top 3 rows.
+----+------------+-----------+------+--------+
| id | first_name | last_name | age  | gender |
+----+------------+-----------+------+--------+
|  1 | Dora       | Smith     |   42 | F      |
|  2 | Ron        | Long      |   35 | M      |
|  3 | Lisa       | Roy       |   29 | F      |
+----+------------+-----------+------+--------+
3 rows in set (0.00 sec)

mysql> SELECT *
    -> FROM friends
    -> LIMIT 2, 3; ← Limit the selections to the top 3 rows, after offset of 2 rows.
+----+------------+-----------+------+--------+
| id | first_name | last_name | age  | gender |
+----+------------+-----------+------+--------+
|  3 | Lisa       | Roy       |   29 | F      | ← Offset of 2 rows, so starting id is 3.
|  4 | Dora       | G.        |   21 | F      |
|  5 | Tara       | Long      |   29 | F      |
+----+------------+-----------+------+--------+
3 rows in set (0.00 sec)

WHERE
-------------------------------------------------------------------------------
mysql> SELECT *
    -> FROM friends
    -> WHERE age LIKE '3_'; ← Select where some conditions are met (we have seen this before).
+----+------------+-----------+------+--------+
| id | first_name | last_name | age  | gender |
+----+------------+-----------+------+--------+
|  2 | Ron        | Long      |   35 | M      |
|  7 | Ron        | long      |   30 | M      |
+----+------------+-----------+------+--------+
2 rows in set (0.00 sec)

mysql>
```

This is another important milestone, at this point we are well equipped to write some powerful queries!

4.6 Practice problems - refining selections ***

Problem 5 (Select specific friends)
<Link to the solution>

For this and the next few problems we will use an updated `friends` table, which includes their emails. Go ahead and create the `friends` table using the following script.

```
DROP TABLE IF EXISTS friends;

CREATE TABLE friends (
  id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
  first_name VARCHAR(255),
  last_name VARCHAR(255),
  email VARCHAR(255) UNIQUE, ← We are adding email info.
  age INT,
  gender CHAR(1)
);

INSERT INTO friends
  (first_name, last_name, email, age, gender)
VALUES
  ('Dora', 'Smith', 'dora@gmail.com', 42, 'F'),
  ('Ron', 'Long', 'ron123@comcast.net', 35, 'M'),
  ('Lisa', 'Roy', 'lisaroy@gmail.com', 29, 'F'),
  ('Dora', 'G.', 'dg@yahoo.com', 21, 'F'),
  ('Tara', 'Long', 'tlong@gmail.com', 29, 'F'),
  ('Ryan', 'Davis', 'ryan99@aol.com', 40, 'M'),
  ('Ron', 'long', 'ron1st@comcast.net', 30, 'M');

SELECT * FROM friends;
```

If you do `SELECT *`, the `friends` table should look something like this.

```
+----+------------+-----------+--------------------+------+--------+
| id | first_name | last_name | email              | age  | gender |
+----+------------+-----------+--------------------+------+--------+
|  1 | Dora       | Smith     | dora@gmail.com     |   42 | F      |
|  2 | Ron        | Long      | ron123@comcast.net |   35 | M      |
|  3 | Lisa       | Roy       | lisaroy@gmail.com  |   29 | F      |
|  4 | Dora       | G.        | dg@yahoo.com       |   21 | F      |
|  5 | Tara       | Long      | tlong@gmail.com    |   29 | F      |
|  6 | Ryan       | Davis     | ryan99@aol.com     |   40 | M      |
|  7 | Ron        | long      | ron1st@comcast.net |   30 | M      |
+----+------------+-----------+--------------------+------+--------+
```

List all your friends

- who are 30 years or older
- whose age is an odd number

The result should look something like this.

```
+----+------------+-----------+--------------------+------+--------+
| id | first_name | last_name | email              | age  | gender |
+----+------------+-----------+--------------------+------+--------+
|  1 | Dora       | Smith     | dora@gmail.com     |   42 | F      |
|  2 | Ron        | Long      | ron123@comcast.net |   35 | M      |
|  6 | Ryan       | Davis     | ryan99@aol.com     |   40 | M      |
|  7 | Ron        | long      | ron1st@comcast.net |   30 | M      |
+----+------------+-----------+--------------------+------+--------+

+----+------------+-----------+--------------------+------+--------+
| id | first_name | last_name | email              | age  | gender |
+----+------------+-----------+--------------------+------+--------+
|  2 | Ron        | Long      | ron123@comcast.net |   35 | M      |
|  3 | Lisa       | Roy       | lisaroy@gmail.com  |   29 | F      |
|  4 | Dora       | G.        | dg@yahoo.com       |   21 | F      |
|  5 | Tara       | Long      | tlong@gmail.com    |   29 | F      |
+----+------------+-----------+--------------------+------+--------+
```

Problem 6 (Name and email of oldest friend)
<Link to the solution>

Find the full name (first name space last name) and email of the oldest friend (use the table of Problem 5). Assume that there is only one friend matching the criteria.

The result should look something like this.

```
+------------+-----------------+
| name       | email           |
+------------+-----------------+
| Dora Smith | dora@gmail.com |
+------------+-----------------+
```

Problem 7 (Name and email of oldest male friend)
<Link to the solution>

Find the name and email of the oldest *male* friend (use the table of Problem 5). Again, assume that there is only one friend matching the criteria.

The result should look something like this.

```
+------------+-----------------+
| name       | email           |
+------------+-----------------+
| Ryan Davis | ryan99@aol.com |
+------------+-----------------+
```

Problem 8 (Email providers and count 1)
<Link to the solution>

Show a count of how many of your friends use gmail, yahoo email, and other emails (use the table of Problem 5). Show the counts in decreasing order.

The result should look something like this.

```
+----------+-------------+
| provider | total_users |
+----------+-------------+
| gmail    |           3 |
| other    |           3 |
| yahoo    |           1 |
+----------+-------------+
```

Problem 9 (Email providers and count 2)
<Link to the solution>

List the different email providers in alphabetical order (use the table of Problem 5). The result should look something like this.

```
+----------+
| provider |
+----------+
| aol      |
| comcast  |
| gmail    |
| yahoo    |
+----------+
```

Next, list the different email providers and count (in decreasing order). The result should look something like this (contrast with Problem 8).

```
+----------+-------------+
| provider | total_users |
+----------+-------------+
| gmail    |           3 |
| comcast  |           2 |
| yahoo    |           1 |
| aol      |           1 |
+----------+-------------+
```

Problem 10 (Masked emails)
<Link to the solution>

Show the id, name (first name space last initial capitalized), and masked email (email username replaced by the first and last chars separated by ***) of all the friends. Use the table of Problem 5.

The result should look something like this.

```
+----+---------+-------------------+
| id | name    | masked email      |
+----+---------+-------------------+
|  1 | Dora S. | d***a@gmail.com   |
|  2 | Ron L.  | r***3@comcast.net |
|  3 | Lisa R. | l***y@gmail.com   |
|  4 | Dora G. | d***g@yahoo.com   |
|  5 | Tara L. | t***g@gmail.com   |
|  6 | Ryan D. | r***9@aol.com     |
|  7 | Ron L.  | r***t@comcast.net |
+----+---------+-------------------+
```

Problem 11 (Machine logs)
<Link to the solution>

For this and the next few problems we will use machine logs from a factory. Go ahead and create the `machinelogs` table using the following script.

```
DROP TABLE IF EXISTS machinelogs;

CREATE TABLE machinelogs (
  id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
  machineid VARCHAR(255),
  start_time DATETIME,
  stop_time DATETIME,
  stop_mode VARCHAR(255)
);

INSERT INTO machinelogs
  (machineid, start_time, stop_time, stop_mode)
VALUES
  ('bay28', '2019-08-10 01:30:00', '2019-08-10 21:30:00', 'normal'),
  ('bay43', '2019-08-10 02:00:00', '2019-08-10 02:10:00', 'failure'),
  ('bay01', '2019-08-10 02:30:00', '2019-08-10 10:00:00', 'normal'),
  ('bay71', '2019-08-10 04:00:00', '2019-08-10 12:30:00', 'normal'),
  ('bay28', '2019-08-10 12:30:00', '2019-08-10 13:00:00', 'failure'),
  ('bay43', '2019-08-10 15:00:00', '2019-08-10 15:15:00', 'failure');

SELECT * FROM machinelogs;
```

If you do `SELECT *`, the `machinelogs` table should look something like this.

```
+----+-----------+---------------------+---------------------+-----------+
| id | machineid | start_time          | stop_time           | stop_mode |
+----+-----------+---------------------+---------------------+-----------+
|  1 | bay28     | 2019-08-10 01:30:00 | 2019-08-10 21:30:00 | normal    |
|  2 | bay43     | 2019-08-10 02:00:00 | 2019-08-10 02:10:00 | failure   |
|  3 | bay01     | 2019-08-10 02:30:00 | 2019-08-10 10:00:00 | normal    |
|  4 | bay71     | 2019-08-10 04:00:00 | 2019-08-10 12:30:00 | normal    |
|  5 | bay28     | 2019-08-10 12:30:00 | 2019-08-10 13:00:00 | failure   |
|  6 | bay43     | 2019-08-10 15:00:00 | 2019-08-10 15:15:00 | failure   |
+----+-----------+---------------------+---------------------+-----------+
```

Here `machineid` is the machine id (unique for a given machine), `start_time` is when the machine started running, `stop_time` is when the machine stopped running, and `stop_mode` is why the machine stopped (can be either `normal` or `failure`).

List all the runs in decreasing order of their run time in hours (truncate to 2 decimal places). The result should look something like this.

```
+----+-----------+---------------------+----------+
| id | machineid | start_time          | run_time |
+----+-----------+---------------------+----------+
|  1 | bay28     | 2019-08-10 01:30:00 |    20.00 |
|  4 | bay71     | 2019-08-10 04:00:00 |     8.50 |
|  3 | bay01     | 2019-08-10 02:30:00 |     7.50 |
|  5 | bay28     | 2019-08-10 12:30:00 |     0.50 |
|  6 | bay43     | 2019-08-10 15:00:00 |     0.25 |
|  2 | bay43     | 2019-08-10 02:00:00 |     0.17 |
+----+-----------+---------------------+----------+
```

Problem 12 (Failed runs)
<Link to the solution>

List the stop time of the failed runs (use the table of Problem 11).

The result should look something like this.

```
+----+-----------+---------------------+
| id | machineid | stop_time           |
+----+-----------+---------------------+
|  2 | bay43     | 2019-08-10 02:10:00 |
|  5 | bay28     | 2019-08-10 13:00:00 |
|  6 | bay43     | 2019-08-10 15:15:00 |
+----+-----------+---------------------+
```

Chapter 5

Joins

5.1 Relationships and joins

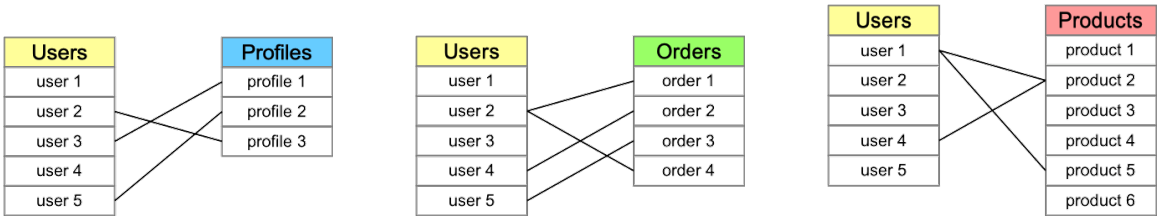
In real world applications different classes of data are stored in different tables (for efficiency, security, and practicality). For example, an e-commerce app might have a users table to store the login information (username, password, email, ...), a products table to store product information (sku, price, stock quantity, ...), an orders table to store different orders placed (order number, customer id, products ordered, total amount, ...), etc. There will be some relationships among these tables, and often we need to gather data from two or more tables (= join) to be able to answer business questions.

There are three types of relationships we need to know while designing a database schema:

1. One-to-one
    - E.g., Users ↔ Profiles
    - Every profile belongs to a specific user (although, every user need not have a profile).
2. One-to-many
    - E.g., Users ↔ Orders
    - A user can have many orders, but an order belongs to a specific user.
3. Many-to-many
    - E.g., Users ↔ Products
    - A user can review many products, and a produce can be reviewed by many users.

This is a very important concept, and having a concrete example in mind helps a lot. These relationships are depicted pictorially in Fig. 5.1.

Relationships and joins go hand in hand - relationships help split the data into different tables, and joins help gather relevant data from different tables to answer business questions. We will look into the different types of relationships, and how to model them, in more details shortly. But given two table, first let's see how we can join them.



(a) One-to-one *(b)* One-to-many *(c)* Many-to-many

Fig. 5.1: SQL relationships

5.1.1 Joins

The different types of joins are best depicted with Venn diagrams, see Figure 5.2. For illustration, we have set `A` of land animals, set `B` of water animals, and amphibians are common to both sets.

Among all the joins, you will be using the `LEFT` and `INNER` most often, and `LEFT EXCLUSIVE` occasionally. If you ever need the other types, they can be derived from these three. For example, `RIGHT` is same as `LEFT`, except switch `A` and `B`. The syntax for the three types of joins are shown in Table 5.1.

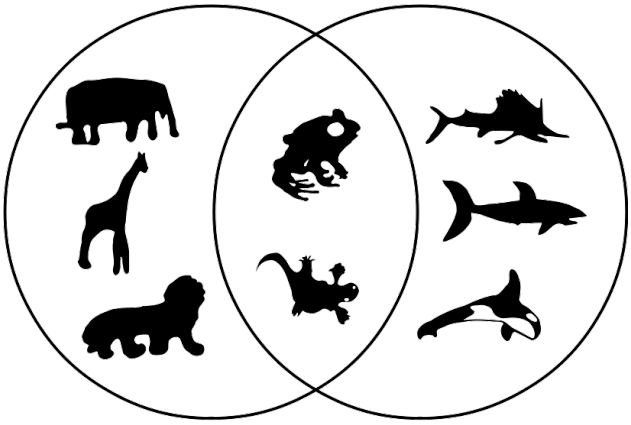
```
1  -- LEFT
2  SELECT *
3  FROM A
4  LEFT JOIN B
5    ON A.key = B.key;
6
7  -- LEFT EXCLUSIVE
8  SELECT *
9  FROM A
10 LEFT JOIN B
11   ON A.key = B.key;
12 WHERE B.key IS NULL;
13
14 -- INNER
15 SELECT *
16 FROM A
17 INNER JOIN B
18   ON A.key = B.key;
```

Table 5.1: SQL joins syntax

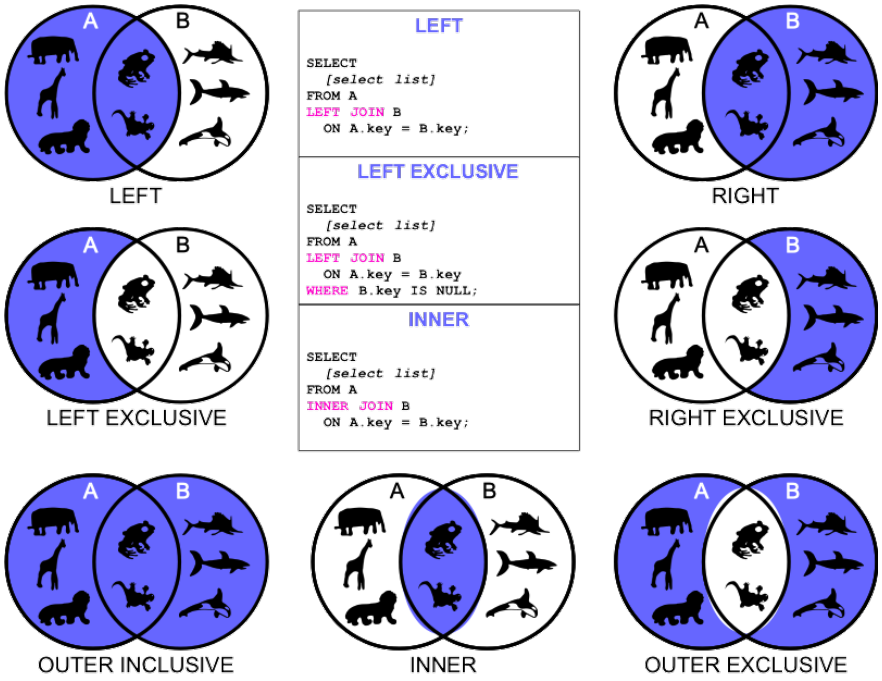
Let's take a look at the three types of joins (`LEFT`, `LEFT EXCLUSIVE`, and `INNER`) with some examples.

```
1  For the demo we will use couple of animals tables, as given below.[1]
2
3  mysql> DROP TABLE IF EXISTS land_animals;
4  Query OK, 0 rows affected (0.01 sec)
5
6  mysql> CREATE TABLE land_animals ( ← Land animals (animals that can live on the land)
7      ->   name VARCHAR(255),
8      ->   land_speed_mph INT
9      -> );
10 Query OK, 0 rows affected (0.02 sec)
11
```

[1]The animal speed in the tables are representative only, researched by my 8 yo son. The animal artwork credit also goes to him.



(a) Sets of animals



(b) Different joins

Fig. 5.2: SQL joins

(a) Pure land animals: Elephant, Giraffe, Lion; Pure water animals: Sailfish, Shark, Whale; Amphibians: Frog, Salamander (b) Different joins are highlighted, most used are the `LEFT` and `INNER` joins

```
mysql> INSERT INTO land_animals
    ->   (name, land_speed_mph)
    -> VALUES
    ->   ('elephant', 15),
    ->   ('frog', 2),
    ->   ('giraffe', 30),
    ->   ('lion', 50),
    ->   ('salamander', 10);
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM land_animals;
+------------+----------------+
| name       | land_speed_mph |
+------------+----------------+
| elephant   |             15 |
| frog       |              2 | ← Amphibian
| giraffe    |             30 |
| lion       |             50 |
| salamander |             10 | ←
+------------+----------------+
5 rows in set (0.00 sec)

mysql>
mysql> DROP TABLE IF EXISTS water_animals;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE water_animals ( ← Water animals (animals that can live in the water)
    ->   name VARCHAR(255),
    ->   water_speed_mph INT
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO water_animals
    ->   (name, water_speed_mph)
    -> VALUES
    ->   ('frog', 5),
    ->   ('sailfish', 70),
    ->   ('salamander', 20),
    ->   ('shark', 25),
    ->   ('whale', 35);
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM water_animals;
+------------+-----------------+
| name       | water_speed_mph |
+------------+-----------------+
| frog       |               5 | ← Amphibian
| sailfish   |              70 |
| salamander |              20 | ←
| shark      |              25 |
| whale      |              35 |
+------------+-----------------+
5 rows in set (0.00 sec)

LEFT join
```

```
mysql> SELECT *
    -> FROM land_animals
    -> LEFT JOIN water_animals ← LEFT JOIN, here land_animals is the left table and water_animals is the right table (see Fig. 5.2b)
    ->   ON land_animals.name = water_animals.name; ← Match the name of land animal with the name of water animal for joining
+------------+----------------+------------+-----------------+
| name       | land_speed_mph | name       | water_speed_mph |
+------------+----------------+------------+-----------------+
| frog       |              2 | frog       |               5 | ← Matching row
| salamander |             10 | salamander |              20 | ←
| elephant   |             15 | NULL       |            NULL | ← Non-matching row
| giraffe    |             30 | NULL       |            NULL | ←
| lion       |             50 | NULL       |            NULL | ←
+------------+----------------+------------+-----------------+
5 rows in set (0.00 sec)
```

Explanation: We start with the rows of the left table (land animals). Then based on the match criteria (in this case the name) we include only the matching rows from the right table (water animals). The matching rows come first, and then the non-matching rows, of the left table. For the non-matching rows, the values corresponding to the right table columns are set to NULL.

```
LEFT EXCLUSIVE join
```

```
mysql> SELECT *
    -> FROM land_animals
    -> LEFT JOIN water_animals
    ->   ON land_animals.name = water_animals.name
    -> WHERE water_animals.name IS NULL; ← Same as the LEFT JOIN, except this WHERE clause which makes it LEFT EXCLUSIVE (see Fig. 5.2b)
+----------+----------------+------+-----------------+
| name     | land_speed_mph | name | water_speed_mph |
+----------+----------------+------+-----------------+
| elephant |             15 | NULL |            NULL |
| giraffe  |             30 | NULL |            NULL |
| lion     |             50 | NULL |            NULL |
+----------+----------------+------+-----------------+
3 rows in set (0.00 sec)
```

Explanation: Because of the WHERE clause, the result includes only the animals that are exclusively on the left table only, and not on the right

```
        table. These are essentially the non-matching rows of the left table (see the earlier LEFT join).
102
103 INNER join
104 ------------------------------------------------------------------------------------------
105 mysql> SELECT *
106     -> FROM land_animals
107     -> INNER JOIN water_animals ← INNER JOIN (see Fig. 5.2b)
108     ->   ON land_animals.name = water_animals.name;
109 +------------+----------------+------------+-----------------+
110 | name       | land_speed_mph | name       | water_speed_mph |
111 +------------+----------------+------------+-----------------+
112 | frog       |              2 | frog       |               5 |
113 | salamander |             10 | salamander |              20 |
114 +------------+----------------+------------+-----------------+
115 2 rows in set (0.00 sec)
116 Explanation: These are essentially the matching rows of the left table (see the earlier LEFT join).
```

You will be using the joins frequently, so make sure you are super clear how the different joins work (the Venn diagrams in Fig. 5.2 make it really easy to understand).

Next let's take a look at the different relationships.

5.1.2 Relationships

We have seen the different relationships pictorially in Fig. 5.1. Here we will see how to model those relationships. The modeling syntax is presented in Table 5.2.

A few things worth noting (see Table 5.2):

1. The syntax of one-to-one and one-to-many tables are the same, except
   - One-to-one has the `UNIQUE` restriction, to enforce the relationship. There cannot be multiple profiles associated with the same user. However, there can be multiple orders associated with the same user.
   - You can use `ON DELETE CASCADE` to clean up the `profiles` table - when a user is deleted it will automatically delete the corresponding profile if any.
2. For the many-to-many, you will encounter couple of scenarios
   - For tables like `authorships` or `reviews`, there cannot be duplicate id-pairs. Meaning, a given author cannot author the same book twice. Or, a given user cannot give duplicate reviews for the same book. In such cases you can use the id-pair as the `PRIMARY KEY`, to enforce the pair uniqueness.
   - On the other hand, for tables like `comments`, one user can comment multiple times for the same book. And therefore, duplicate id-pairs are allowed. And therefore, id-pair is not used as the `PRIMARY KEY`.

Now let's take a look at the different relationships with some examples.

```
1  For the demo we will use a bookstore example. The relevant tables are given below.
2  mysql> CREATE TABLE users ( ← users table
3      ->   id INT AUTO_INCREMENT PRIMARY KEY,
4      ->   email VARCHAR(255) UNIQUE NOT NULL,
5      ->   first_name VARCHAR(255),
6      ->   last_name VARCHAR(255)
7      -> );
8  Query OK, 0 rows affected (0.02 sec)
9
10 mysql> INSERT INTO users
11     ->   (email, first_name, last_name)
12     -> VALUES
13     ->   ('dora@gmail.com', 'Dora', 'Smith'),
14     ->   ('ron123@comcast.net', 'Ron', 'Long'),
15     ->   ('lisaroy@gmail.com', 'Lisa', 'Roy'),
16     ->   ('dg@yahoo.com', 'Dora', 'G.'),
17     ->   ('tlong@gmail.com', 'Tara', 'Long'),
18     ->   ('ryan99@aol.com', 'Ryan', 'Davis'),
```

```sql
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  ...
);

CREATE TABLE books (
  id INT AUTO_INCREMENT PRIMARY KEY,
  ...
);

-- One-to-one
CREATE TABLE profiles (
  id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT UNIQUE, ← Note the UNIQUE
  ...,
  FOREIGN KEY (user_id)
    REFERENCES users(id)
    ON DELETE CASCADE ← Note the CASCADE
);

-- One-to-many
CREATE TABLE orders (
  id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT,
  ...,
  FOREIGN KEY (user_id)
    REFERENCES users(id)
);

-- Many-to-many
CREATE TABLE authors (
  book_id INT,
  author_id INT,
  FOREIGN KEY (book_id) REFERENCES books(id),
  FOREIGN KEY (author_id) REFERENCES users(id),
  PRIMARY KEY(book_id, author_id) ← Note the PRIMARY KEY
);

CREATE TABLE comments (
  id INT AUTO_INCREMENT PRIMARY KEY,
  book_id INT,
  user_id INT,
  ...,
  FOREIGN KEY (book_id) REFERENCES books(id),
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Table 5.2: SQL relationships modeling

```
    ->   ('ron1st@comcast.net', 'Ron', 'long');
Query OK, 7 rows affected (0.00 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM users;
+----+--------------------+------------+-----------+
| id | email              | first_name | last_name |
+----+--------------------+------------+-----------+
|  1 | dora@gmail.com     | Dora       | Smith     |
|  2 | ron123@comcast.net | Ron        | Long      |
|  3 | lisaroy@gmail.com  | Lisa       | Roy       |
|  4 | dg@yahoo.com       | Dora       | G.        |
|  5 | tlong@gmail.com    | Tara       | Long      |
|  6 | ryan99@aol.com     | Ryan       | Davis     |
|  7 | ron1st@comcast.net | Ron        | long      |
+----+--------------------+------------+-----------+
7 rows in set (0.00 sec)

One-to-one
_______________________________________________________________________________
mysql> CREATE TABLE profiles ( ← profiles table, one-to-one
    ->   id INT AUTO_INCREMENT PRIMARY KEY,
    ->   user_id INT UNIQUE,
    ->   age INT,
    ->   gender CHAR(1),
    ->   address_state CHAR(2),
    ->   FOREIGN KEY (user_id)
    ->     REFERENCES users(id)
    ->     ON DELETE CASCADE
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO profiles
    ->   (user_id, age, gender, address_state)
    -> VALUES
    ->   (1, 42, 'F', 'CA'),
    ->   (4, 21, 'F', 'CA'),
    ->   (2, 35, 'M', 'NY'),
    ->   (7, 30, 'M', 'CA'),
    ->   (3, 29, 'F', 'CA'),
    ->   (5, 29, 'F', 'MN');
Query OK, 6 rows affected (0.01 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM profiles;
+----+---------+------+--------+---------------+
| id | user_id | age  | gender | address_state |
+----+---------+------+--------+---------------+
|  1 |       1 |   42 | F      | CA            |
|  2 |       4 |   21 | F      | CA            |
|  3 |       2 |   35 | M      | NY            |
|  4 |       7 |   30 | M      | CA            |
|  5 |       3 |   29 | F      | CA            |
|  6 |       5 |   29 | F      | MN            |
+----+---------+------+--------+---------------+
6 rows in set (0.00 sec)

mysql> SELECT *
    -> FROM users
    -> LEFT JOIN profiles ← join, one-to-one
    ->   ON users.id = profiles.user_id;
+----+--------------------+------------+-----------+------+---------+------+--------+---------------+
| id | email              | first_name | last_name | id   | user_id | age  | gender | address_state |
+----+--------------------+------------+-----------+------+---------+------+--------+---------------+
|  1 | dora@gmail.com     | Dora       | Smith     |    1 |       1 |   42 | F      | CA            |
|  2 | ron123@comcast.net | Ron        | Long      |    3 |       2 |   35 | M      | NY            |
|  3 | lisaroy@gmail.com  | Lisa       | Roy       |    5 |       3 |   29 | F      | CA            |
|  4 | dg@yahoo.com       | Dora       | G.        |    2 |       4 |   21 | F      | CA            |
|  5 | tlong@gmail.com    | Tara       | Long      |    6 |       5 |   29 | F      | MN            |
|  6 | ryan99@aol.com     | Ryan       | Davis     | NULL |    NULL | NULL | NULL   | NULL          |
|  7 | ron1st@comcast.net | Ron        | long      |    4 |       7 |   30 | M      | CA            |
+----+--------------------+------------+-----------+------+---------+------+--------+---------------+
7 rows in set (0.00 sec)

One-to-many
_______________________________________________________________________________
mysql> CREATE TABLE orders ( ← orders table, one-to-many
    ->   id INT AUTO_INCREMENT PRIMARY KEY,
    ->   user_id INT,
    ->   dollar_amount INT,
    ->   order_date DATE,
    ->   FOREIGN KEY (user_id)
    ->     REFERENCES users(id)
    -> );
Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO orders
    ->   (user_id, dollar_amount, order_date)
    -> VALUES
    ->   (5, 89, '2019-01-18'),
    ->   (1, 24, '2019-03-01'),
    ->   (5, 11, '2019-03-01'),
```

```
    ->   (7, 50, '2019-04-01'),
    ->   (2, 65, '2019-04-20'),
    ->   (5, 38, '2019-05-18'),
    ->   (5, 38, '2019-05-19'),
    ->   (1, 42, '2019-06-05'),
    ->   (7, 69, '2019-06-28');
Query OK, 9 rows affected (0.00 sec)
Records: 9  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM orders;
+----+---------+---------------+------------+
| id | user_id | dollar_amount | order_date |
+----+---------+---------------+------------+
|  1 |       5 |            89 | 2019-01-18 |
|  2 |       1 |            24 | 2019-03-01 |
|  3 |       5 |            11 | 2019-03-01 |
|  4 |       7 |            50 | 2019-04-01 |
|  5 |       2 |            65 | 2019-04-20 |
|  6 |       5 |            38 | 2019-05-18 |
|  7 |       5 |            38 | 2019-05-19 |
|  8 |       1 |            42 | 2019-06-05 |
|  9 |       7 |            69 | 2019-06-28 |
+----+---------+---------------+------------+
9 rows in set (0.00 sec)

mysql> SELECT *
    -> FROM users
    -> LEFT JOIN orders ← join, one-to-many
    ->   ON users.id = orders.user_id;
+----+--------------------+------------+-----------+------+---------+---------------+------------+
| id | email              | first_name | last_name | id   | user_id | dollar_amount | order_date |
+----+--------------------+------------+-----------+------+---------+---------------+------------+
|  1 | dora@gmail.com     | Dora       | Smith     |    2 |       1 |            24 | 2019-03-01 |
|  1 | dora@gmail.com     | Dora       | Smith     |    8 |       1 |            42 | 2019-06-05 |
|  2 | ron123@comcast.net | Ron        | Long      |    5 |       2 |            65 | 2019-04-20 |
|  3 | lisaroy@gmail.com  | Lisa       | Roy       | NULL |    NULL |          NULL | NULL       |
|  4 | dg@yahoo.com       | Dora       | G.        | NULL |    NULL |          NULL | NULL       |
|  5 | tlong@gmail.com    | Tara       | Long      |    1 |       5 |            89 | 2019-01-18 |
|  5 | tlong@gmail.com    | Tara       | Long      |    3 |       5 |            11 | 2019-03-01 |
|  5 | tlong@gmail.com    | Tara       | Long      |    6 |       5 |            38 | 2019-05-18 |
|  5 | tlong@gmail.com    | Tara       | Long      |    7 |       5 |            38 | 2019-05-19 |
|  6 | ryan99@aol.com     | Ryan       | Davis     | NULL |    NULL |          NULL | NULL       |
|  7 | ron1st@comcast.net | Ron        | long      |    4 |       7 |            50 | 2019-04-01 |
|  7 | ron1st@comcast.net | Ron        | long      |    9 |       7 |            69 | 2019-06-28 |
+----+--------------------+------------+-----------+------+---------+---------------+------------+
12 rows in set (0.00 sec)

Many-to-many
-------------------------------------------------------------------------------

mysql> CREATE TABLE books ( ← books table
    ->   id INT AUTO_INCREMENT PRIMARY KEY,
    ->   title VARCHAR(255),
    ->   pages INT
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO books
    ->   (title, pages)
    -> VALUES
    ->   ('Cook with Lisa and Ryan', 210),
    ->   ('Ryan\'s adventure', 150),
    ->   ('10 minutes workout with Tara', 60),
    ->   ('Ryan, Dora, and Ron\'s giude to SQL', 350),
    ->   ('Swim with Tara and run with Dora', 70);
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM books;
+----+-----------------------------------+-------+
| id | title                             | pages |
+----+-----------------------------------+-------+
|  1 | Cook with Lisa and Ryan           |   210 |
|  2 | Ryan's adventure                  |   150 |
|  3 | 10 minutes workout with Tara      |    60 |
|  4 | Ryan, Dora, and Ron's giude to SQL |   350 |
|  5 | Swim with Tara and run with Dora  |    70 |
+----+-----------------------------------+-------+
5 rows in set (0.00 sec)

mysql> DROP TABLE IF EXISTS authorships;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> CREATE TABLE authorships ( ← authorships table, many-to-many
    ->   book_id INT,
    ->   author_id INT,
    ->   FOREIGN KEY (book_id) REFERENCES books(id),
    ->   FOREIGN KEY (author_id) REFERENCES users(id),
    ->   PRIMARY KEY(book_id, author_id)
    -> );
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO authorships
```

```
    ->   (book_id, author_id)
    -> VALUES
    ->   (1, 3),
    ->   (1, 6),
    ->   (2, 6),
    ->   (3, 5),
    ->   (4, 6),
    ->   (4, 1),
    ->   (4, 7),
    ->   (5, 5),
    ->   (5, 1);
Query OK, 9 rows affected (0.01 sec)
Records: 9  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM authorships;
+---------+-----------+
| book_id | author_id |
+---------+-----------+
|       4 |         1 |
|       5 |         1 |
|       1 |         3 |
|       3 |         5 |
|       5 |         5 |
|       1 |         6 |
|       2 |         6 |
|       4 |         6 |
|       4 |         7 |
+---------+-----------+
9 rows in set (0.00 sec)

mysql> SELECT *
    -> FROM users
    -> LEFT JOIN authorships ← join, many-to-many
    ->   ON users.id = authorships.author_id
    -> LEFT JOIN books ←
    ->   ON authorships.book_id = books.id;
+----+--------------------+------------+-----------+---------+-----------+------+-----------------------------------+-------+
| id | email              | first_name | last_name | book_id | author_id | id   | title                             | pages |
+----+--------------------+------------+-----------+---------+-----------+------+-----------------------------------+-------+
|  1 | dora@gmail.com     | Dora       | Smith     |       4 |         1 |    4 | Ryan, Dora, and Ron's giude to SQL |   350 |
|  1 | dora@gmail.com     | Dora       | Smith     |       5 |         1 |    5 | Swim with Tara and run with Dora   |    70 |
|  2 | ron123@comcast.net | Ron        | Long      |    NULL |      NULL | NULL | NULL                              |  NULL |
|  3 | lisaroy@gmail.com  | Lisa       | Roy       |       1 |         3 |    1 | Cook with Lisa and Ryan           |   210 |
|  4 | dg@yahoo.com       | Dora       | G.        |    NULL |      NULL | NULL | NULL                              |  NULL |
|  5 | tlong@gmail.com    | Tara       | Long      |       3 |         5 |    3 | 10 minutes workout with Tara      |    60 |
|  5 | tlong@gmail.com    | Tara       | Long      |       5 |         5 |    5 | Swim with Tara and run with Dora   |    70 |
|  6 | ryan99@aol.com     | Ryan       | Davis     |       1 |         6 |    1 | Cook with Lisa and Ryan           |   210 |
|  6 | ryan99@aol.com     | Ryan       | Davis     |       2 |         6 |    2 | Ryan's adventure                  |   150 |
|  6 | ryan99@aol.com     | Ryan       | Davis     |       4 |         6 |    4 | Ryan, Dora, and Ron's giude to SQL |   350 |
|  7 | ron1st@comcast.net | Ron        | long      |       4 |         7 |    4 | Ryan, Dora, and Ron's giude to SQL |   350 |
+----+--------------------+------------+-----------+---------+-----------+------+-----------------------------------+-------+
11 rows in set (0.00 sec)

mysql>
mysql> CREATE TABLE comments ( ← comments table, many-to-many
    ->   id INT AUTO_INCREMENT PRIMARY KEY,
    ->   book_id INT,
    ->   user_id INT,
    ->   comment VARCHAR(255),
    ->   FOREIGN KEY (book_id) REFERENCES books(id),
    ->   FOREIGN KEY (user_id) REFERENCES users(id)
    -> );
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO comments
    ->   (book_id, user_id, comment)
    -> VALUES ← Values being inserted are sorted just for easy reference
    ->   (1, 1, 'Book 1 comment by user 1 - 1st time'),
    ->   (1, 1, 'Book 1 comment by user 1 - 2nd time'),
    ->   (1, 1, 'Book 1 comment by user 1 - 3rd time'),
    ->   (2, 1, 'Book 2 comment by user 1'),
    ->   (3, 1, 'Book 3 comment by user 1'),
    ->   (4, 1, 'Book 4 comment by user 1 - 1st time'),
    ->   (4, 1, 'Book 4 comment by user 1 - 2nd time'),
    ->   (5, 1, 'Book 5 comment by user 1'),
    ->   (3, 2, 'Book 3 comment by user 2'),
    ->   (1, 4, 'Book 1 comment by user 4'),
    ->   (1, 5, 'Book 1 comment by user 5'),
    ->   (2, 5, 'Book 2 comment by user 5'),
    ->   (3, 5, 'Book 3 comment by user 5 - 1st time'),
    ->   (3, 5, 'Book 3 comment by user 5 - 2nd time'),
    ->   (4, 5, 'Book 4 comment by user 5'),
    ->   (5, 5, 'Book 5 comment by user 5');
Query OK, 16 rows affected (0.00 sec)
Records: 16  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM comments;
+----+---------+---------+-------------------------------------+
| id | book_id | user_id | comment                             |
+----+---------+---------+-------------------------------------+
|  1 |       1 |       1 | Book 1 comment by user 1 - 1st time |
|  2 |       1 |       1 | Book 1 comment by user 1 - 2nd time |
```

```
|  3 |       1 |       1 | Book 1 comment by user 1 - 3rd time |
|  4 |       2 |       1 | Book 2 comment by user 1            |
|  5 |       3 |       1 | Book 3 comment by user 1            |
|  6 |       4 |       1 | Book 4 comment by user 1 - 1st time |
|  7 |       4 |       1 | Book 4 comment by user 1 - 2nd time |
|  8 |       5 |       1 | Book 5 comment by user 1            |
|  9 |       3 |       2 | Book 3 comment by user 2            |
| 10 |       1 |       4 | Book 1 comment by user 4            |
| 11 |       1 |       5 | Book 1 comment by user 5            |
| 12 |       2 |       5 | Book 2 comment by user 5            |
| 13 |       3 |       5 | Book 3 comment by user 5 - 1st time |
| 14 |       3 |       5 | Book 3 comment by user 5 - 2nd time |
| 15 |       4 |       5 | Book 4 comment by user 5            |
| 16 |       5 |       5 | Book 5 comment by user 5            |
+----+---------+---------+-------------------------------------+
16 rows in set (0.00 sec)

mysql> SELECT
    ->   users.id AS 'user id',
    ->   first_name,
    ->   comment,
    ->   books.id AS 'book id',
    ->   title, pages
    -> FROM users
    -> LEFT JOIN comments ← join, many-to-many
    ->   ON users.id = comments.user_id
    -> LEFT JOIN books ←
    ->   ON comments.book_id = books.id;
+---------+------------+-------------------------------------+---------+-----------------------------------+-------+
| user id | first_name | comment                             | book id | title                             | pages |
+---------+------------+-------------------------------------+---------+-----------------------------------+-------+
|       1 | Dora       | Book 1 comment by user 1 - 1st time |       1 | Cook with Lisa and Ryan           |   210 |
|       1 | Dora       | Book 1 comment by user 1 - 2nd time |       1 | Cook with Lisa and Ryan           |   210 |
|       1 | Dora       | Book 1 comment by user 1 - 3rd time |       1 | Cook with Lisa and Ryan           |   210 |
|       1 | Dora       | Book 2 comment by user 1            |       2 | Ryan's adventure                  |   150 |
|       1 | Dora       | Book 3 comment by user 1            |       3 | 10 minutes workout with Tara      |    60 |
|       1 | Dora       | Book 4 comment by user 1 - 1st time |       4 | Ryan, Dora, and Ron's giude to SQL |   350 |
|       1 | Dora       | Book 4 comment by user 1 - 2nd time |       4 | Ryan, Dora, and Ron's giude to SQL |   350 |
|       1 | Dora       | Book 5 comment by user 1            |       5 | Swim with Tara and run with Dora  |    70 |
|       2 | Ron        | Book 3 comment by user 2            |       3 | 10 minutes workout with Tara      |    60 |
|       3 | Lisa       | NULL                                |    NULL | NULL                              |  NULL |
|       4 | Dora       | Book 1 comment by user 4            |       1 | Cook with Lisa and Ryan           |   210 |
|       5 | Tara       | Book 1 comment by user 5            |       1 | Cook with Lisa and Ryan           |   210 |
|       5 | Tara       | Book 2 comment by user 5            |       2 | Ryan's adventure                  |   150 |
|       5 | Tara       | Book 3 comment by user 5 - 1st time |       3 | 10 minutes workout with Tara      |    60 |
|       5 | Tara       | Book 3 comment by user 5 - 2nd time |       3 | 10 minutes workout with Tara      |    60 |
|       5 | Tara       | Book 4 comment by user 5            |       4 | Ryan, Dora, and Ron's giude to SQL |   350 |
|       5 | Tara       | Book 5 comment by user 5            |       5 | Swim with Tara and run with Dora  |    70 |
|       6 | Ryan       | NULL                                |    NULL | NULL                              |  NULL |
|       7 | Ron        | NULL                                |    NULL | NULL                              |  NULL |
+---------+------------+-------------------------------------+---------+-----------------------------------+-------+
19 rows in set (0.00 sec)

mysql>
```

As mentioned before, in most real world problems you will need to apply relationships and joins. So I would say, getting to this point is a significant milestone - congratulations again! Now you can start working on real world applications.

5.2 Practice problems - relationships and joins ***

Problem 13 (Join tables)
<Link to the solution>

We will be using the bookstore tables in Section 5.1.2, namely the `users`, `profiles`, `orders`, `books`, `authorships`, and `comments` tables.

Go ahead and create the tables using the following script.

```
-- users
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  first_name VARCHAR(255),
  last_name VARCHAR(255)
);
INSERT INTO users
  (email, first_name, last_name)
```

```
VALUES
  ('dora@gmail.com', 'Dora', 'Smith'),
  ('ron123@comcast.net', 'Ron', 'Long'),
  ('lisaroy@gmail.com', 'Lisa', 'Roy'),
  ('dg@yahoo.com', 'Dora', 'G.'),
  ('tlong@gmail.com', 'Tara', 'Long'),
  ('ryan99@aol.com', 'Ryan', 'Davis'),
  ('ron1st@comcast.net', 'Ron', 'long');

-- profiles
CREATE TABLE profiles (
  id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT UNIQUE,
  age INT,
  gender CHAR(1),
  address_state CHAR(2),
  FOREIGN KEY (user_id)
    REFERENCES users(id)
    ON DELETE CASCADE
);
INSERT INTO profiles
  (user_id, age, gender, address_state)
VALUES
  (1, 42, 'F', 'CA'),
  (4, 21, 'F', 'CA'),
  (2, 35, 'M', 'NY'),
  (7, 30, 'M', 'CA'),
  (3, 29, 'F', 'CA'),
  (5, 29, 'F', 'MN');

-- orders
CREATE TABLE orders (
  id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT,
  dollar_amount INT,
  order_date DATE,
  FOREIGN KEY (user_id)
    REFERENCES users(id)
);
INSERT INTO orders
  (user_id, dollar_amount, order_date)
VALUES
  (5, 89, '2019-01-18'),
  (1, 24, '2019-03-01'),
  (5, 11, '2019-03-01'),
  (7, 50, '2019-04-01'),
  (2, 65, '2019-04-20'),
  (5, 38, '2019-05-18'),
  (5, 38, '2019-05-19'),
  (1, 42, '2019-06-05'),
  (7, 69, '2019-06-28');

-- books
CREATE TABLE books (
  id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(255),
  pages INT
);
INSERT INTO books
  (title, pages)
VALUES
  ('Cook with Lisa and Ryan', 210),
  ('Ryan\'s adventure', 150),
  ('10 minutes workout with Tara', 60),
  ('Ryan, Dora, and Ron\'s giude to SQL', 350),
  ('Swim with Tara and run with Dora', 70);

-- authorships
CREATE TABLE authorships (
  book_id INT,
  author_id INT,
  FOREIGN KEY (book_id) REFERENCES books(id),
  FOREIGN KEY (author_id) REFERENCES users(id),
  PRIMARY KEY(book_id, author_id)
);
INSERT INTO authorships
  (book_id, author_id)
VALUES
  (1, 3),
  (1, 6),
  (2, 6),
  (3, 5),
  (4, 6),
  (4, 1),
  (4, 7),
  (5, 5),
  (5, 1);

-- comments
CREATE TABLE comments (
  id INT AUTO_INCREMENT PRIMARY KEY,
  book_id INT,
```

```
  user_id INT,
  comment VARCHAR(255),
  FOREIGN KEY (book_id) REFERENCES books(id),
  FOREIGN KEY (user_id) REFERENCES users(id)
);
INSERT INTO comments
  (book_id, user_id, comment)
VALUES
  (1, 1, 'Book 1 comment by user 1 - 1st time'),
  (1, 1, 'Book 1 comment by user 1 - 2nd time'),
  (1, 1, 'Book 1 comment by user 1 - 3rd time'),
  (2, 1, 'Book 2 comment by user 1'),
  (3, 1, 'Book 3 comment by user 1'),
  (4, 1, 'Book 4 comment by user 1 - 1st time'),
  (4, 1, 'Book 4 comment by user 1 - 2nd time'),
  (5, 1, 'Book 5 comment by user 1'),
  (3, 2, 'Book 3 comment by user 2'),
  (1, 4, 'Book 1 comment by user 4'),
  (1, 5, 'Book 1 comment by user 5'),
  (2, 5, 'Book 2 comment by user 5'),
  (3, 5, 'Book 3 comment by user 5 - 1st time'),
  (3, 5, 'Book 3 comment by user 5 - 2nd time'),
  (4, 5, 'Book 4 comment by user 5'),
  (5, 5, 'Book 5 comment by user 5');
```

If you do `SELECT *`, the tables should look something like this.

```
-- users table
+----+--------------------+------------+-----------+
| id | email              | first_name | last_name |
+----+--------------------+------------+-----------+
|  1 | dora@gmail.com     | Dora       | Smith     |
|  2 | ron123@comcast.net | Ron        | Long      |
|  3 | lisaroy@gmail.com  | Lisa       | Roy       |
|  4 | dg@yahoo.com       | Dora       | G.        |
|  5 | tlong@gmail.com    | Tara       | Long      |
|  6 | ryan99@aol.com     | Ryan       | Davis     |
|  7 | ron1st@comcast.net | Ron        | long      |
+----+--------------------+------------+-----------+

-- profiles table
+----+---------+------+--------+---------------+
| id | user_id | age  | gender | address_state |
+----+---------+------+--------+---------------+
|  1 |       1 |   42 | F      | CA            |
|  2 |       4 |   21 | F      | CA            |
|  3 |       2 |   35 | M      | NY            |
|  4 |       7 |   30 | M      | CA            |
|  5 |       3 |   29 | F      | CA            |
|  6 |       5 |   29 | F      | MN            |
+----+---------+------+--------+---------------+

-- orders table
+----+---------+---------------+------------+
| id | user_id | dollar_amount | order_date |
+----+---------+---------------+------------+
|  1 |       5 |            89 | 2019-01-18 |
|  2 |       1 |            24 | 2019-03-01 |
|  3 |       5 |            11 | 2019-03-01 |
|  4 |       7 |            50 | 2019-04-01 |
|  5 |       2 |            65 | 2019-04-20 |
|  6 |       5 |            38 | 2019-05-18 |
|  7 |       5 |            38 | 2019-05-19 |
|  8 |       1 |            42 | 2019-06-05 |
|  9 |       7 |            69 | 2019-06-28 |
+----+---------+---------------+------------+

-- books table (I have purposely included the authors' name in the title for easy reference)
+----+-------------------------------------+-------+
| id | title                               | pages |
+----+-------------------------------------+-------+
|  1 | Cook with Lisa and Ryan             |   210 |
|  2 | Ryan's adventure                    |   150 |
|  3 | 10 minutes workout with Tara        |    60 |
|  4 | Ryan, Dora, and Ron's giude to SQL  |   350 |
|  5 | Swim with Tara and run with Dora    |    70 |
+----+-------------------------------------+-------+

-- authorships table
+---------+-----------+
| book_id | author_id |
+---------+-----------+
|       4 |         1 |
|       5 |         1 |
|       1 |         3 |
|       3 |         5 |
|       5 |         5 |
|       1 |         6 |
|       2 |         6 |
|       4 |         6 |
```

```
|       4 |         7 |
+---------+-----------+

-- comments table (Comments are sorted just for easy reference)
+----+---------+---------+-------------------------------------+
| id | book_id | user_id | comment                             |
+----+---------+---------+-------------------------------------+
|  1 |       1 |       1 | Book 1 comment by user 1 - 1st time |
|  2 |       1 |       1 | Book 1 comment by user 1 - 2nd time |
|  3 |       1 |       1 | Book 1 comment by user 1 - 3rd time |
|  4 |       2 |       1 | Book 2 comment by user 1            |
|  5 |       3 |       1 | Book 3 comment by user 1            |
|  6 |       4 |       1 | Book 4 comment by user 1 - 1st time |
|  7 |       4 |       1 | Book 4 comment by user 1 - 2nd time |
|  8 |       5 |       1 | Book 5 comment by user 1            |
|  9 |       3 |       2 | Book 3 comment by user 2            |
| 10 |       1 |       4 | Book 1 comment by user 4            |
| 11 |       1 |       5 | Book 1 comment by user 5            |
| 12 |       2 |       5 | Book 2 comment by user 5            |
| 13 |       3 |       5 | Book 3 comment by user 5 - 1st time |
| 14 |       3 |       5 | Book 3 comment by user 5 - 2nd time |
| 15 |       4 |       5 | Book 4 comment by user 5            |
| 16 |       5 |       5 | Book 5 comment by user 5            |
+----+---------+---------+-------------------------------------+
```

Print the user id, email, first name, age, gender, and state of everyone who has a profile, ordered by the first name. The result should look something like this.

```
+----+--------------------+------------+------+--------+---------------+
| id | email              | first_name | age  | gender | address_state |
+----+--------------------+------------+------+--------+---------------+
|  1 | dora@gmail.com     | Dora       |   42 | F      | CA            |
|  4 | dg@yahoo.com       | Dora       |   21 | F      | CA            |
|  3 | lisaroy@gmail.com  | Lisa       |   29 | F      | CA            |
|  2 | ron123@comcast.net | Ron        |   35 | M      | NY            |
|  7 | ron1st@comcast.net | Ron        |   30 | M      | CA            |
|  5 | tlong@gmail.com    | Tara       |   29 | F      | MN            |
+----+--------------------+------------+------+--------+---------------+
```

Problem 14 (Custom message with coupon code)
<Link to the solution>

Use the tables of Problem 13. We want to email coupon codes to all the users from CA. Coupon code is their email username in uppercase appended with an underscore followed by the percentage discount value. All users get 10% discount, except any female users over the age of 25 get 15% discount. And of course, we want to send coupon codes only to users who have a profile.

Print emails, and personalized messages addressing users by their first names (Mr. x for males, Ms. y for females) and containing the coupon code and percentage discount (z%), ordered by their first names.

The result should look something like this.

```
+--------------------+-------------------------------------------------------------------------+
| email              | message                                                                 |
+--------------------+-------------------------------------------------------------------------+
| dora@gmail.com     | Hello Ms. Dora, Use the coupon code DORA_15 to get a 15% discount.      |
| dg@yahoo.com       | Hello Ms. Dora, Use the coupon code DG_10 to get a 10% discount.        |
| lisaroy@gmail.com  | Hello Ms. Lisa, Use the coupon code LISAROY_15 to get a 15% discount.   |
| ron1st@comcast.net | Hello Mr. Ron, Use the coupon code RON1ST_10 to get a 10% discount.     |
+--------------------+-------------------------------------------------------------------------+
```

Hint: Build the solution query step by step.

Problem 15 (Order count)
<Link to the solution>

Use the tables of Problem 13. Print user id, email, first name, state, and order count for every user, ordered by the count (most orders first). The result should look something like this.

```
+----+--------------------+------------+---------------+--------+
| id | email              | first_name | address_state | count_ |
+----+--------------------+------------+---------------+--------+
|  5 | tlong@gmail.com    | Tara       | MN            |      4 |
|  1 | dora@gmail.com     | Dora       | CA            |      2 |
|  7 | ron1st@comcast.net | Ron        | CA            |      2 |
|  2 | ron123@comcast.net | Ron        | NY            |      1 |
|  3 | lisaroy@gmail.com  | Lisa       | CA            |      0 |
|  4 | dg@yahoo.com       | Dora       | CA            |      0 |
|  6 | ryan99@aol.com     | Ryan       | NULL          |      0 |
+----+--------------------+------------+---------------+--------+
```

Next, print the email and full name of the user who has the most number of orders (assume there is only one result). The result should look something like this.

```
+-----------------+-----------+
| email           | name      |
+-----------------+-----------+
| tlong@gmail.com | Tara Long |
+-----------------+-----------+
```

Problem 16 (Total sale by month)
<Link to the solution>

Use the tables of Problem 13, and answer the following questions.

(A) What is the total sale by month?

(B) Which months had sales over $100?

(C) Which month had the maximum sale (assume there is only one result)?

The results should look something like this.

```
(A) Total sale
+---------+-------+
| month   | total |
+---------+-------+
| January |    89 |
| March   |    35 |
| April   |   115 |
| May     |    76 |
| June    |   111 |
+---------+-------+

(B) Sales over $100
+-------+-------+
| month | total |
+-------+-------+
| April |   115 |
| June  |   111 |
+-------+-------+

(C) Maximum sale month
+-------+
| month |
+-------+
| April |
+-------+
```

Problem 17 (Books and authors)
<Link to the solution>

Use the tables of Problem 13, and answer the following questions.

(A) Which books have over 100 pages?

(B) Which authors have written more than 1 books? Order most books first, and then by last name ascending.

(C) Which books have more than 1 authors? Order most authors first, and then by title ascending.

The results should look something like this.

```
(A) Books having >100 pages
+----+-------------------------------------+-------+
| id | title                               | pages |
+----+-------------------------------------+-------+
|  1 | Cook with Lisa and Ryan             |   210 |
|  2 | Ryan's adventure                    |   150 |
|  4 | Ryan, Dora, and Ron's giude to SQL |   350 |
+----+-------------------------------------+-------+

(B) Authors having >1 books
+------------+-----------+-----------------+------+-------------+
| first_name | last_name | email           | age  | no_of_books |
+------------+-----------+-----------------+------+-------------+
| Ryan       | Davis     | ryan99@aol.com  | NULL |           3 |
| Tara       | Long      | tlong@gmail.com |   29 |           2 |
| Dora       | Smith     | dora@gmail.com  |   42 |           2 |
+------------+-----------+-----------------+------+-------------+

(C) Books having >1 authors
+-----------------------------------+---------------+
| title                             | no_of_authors |
+-----------------------------------+---------------+
| Ryan, Dora, and Ron's giude to SQL |             3 |
| Cook with Lisa and Ryan           |             2 |
| Swim with Tara and run with Dora  |             2 |
+-----------------------------------+---------------+
```

Problem 18 (Comments)
<Link to the solution>

Use the tables of Problem 13, and answer the following questions.

(A) Which users have commented more than once on a given book

(B) Which users have not commented on any of the books

(C) Which users have commented on all the books

The results should look something like this.

```
(A) Users who have commented more than once on a given book
+------------+-----------------+-----------------------------------+-----------------+
| name       | email           | title                             | times_commented |
+------------+-----------------+-----------------------------------+-----------------+
| Dora Smith | dora@gmail.com  | Cook with Lisa and Ryan           |               3 |
| Dora Smith | dora@gmail.com  | Ryan, Dora, and Ron's giude to SQL |               2 |
| Tara Long  | tlong@gmail.com | 10 minutes workout with Tara      |               2 |
+------------+-----------------+-----------------------------------+-----------------+
3 rows in set (0.00 sec)

(B) Users who have never commented
+----+------------+--------------------+-----------------+
| id | name       | email              | books_commented |
+----+------------+--------------------+-----------------+
|  3 | Lisa Roy   | lisaroy@gmail.com  |               0 |
|  6 | Ryan Davis | ryan99@aol.com     |               0 |
|  7 | Ron long   | ron1st@comcast.net |               0 |
+----+------------+--------------------+-----------------+
3 rows in set (0.00 sec)

(C) Users who have commented on all the books
+------+------------+-----------------+-----------------+
| id   | name       | email           | books_commented |
+------+------------+-----------------+-----------------+
|    1 | Dora Smith | dora@gmail.com  |               5 |
|    5 | Tara Long  | tlong@gmail.com |               5 |
+------+------------+-----------------+-----------------+
2 rows in set (0.00 sec)
```

Chapter 6

Resources

6.1 Further learning resources

Learning a new skill can be overwhelming in the beginning. So purposely, I have covered here only as much as needed to have a good level of understanding to hit the ground running. However, here are some FREE resources to explore more.

- MySQL 8.0 Reference Manual: https://dev.mysql.com/doc/refman/8.0/en/
- SQL Tutorial: https://www.w3schools.com/sql/
- The book website will eventually have video tutorials, and a lot more practice problems
- If you get stuck, just Google search (which will often lead you to https://stackoverflow.com/)!

6.2 Solutions to practice problems

Solution 1

<Link to Problem 1>

```sql
CREATE DATABASE running_app;
USE running_app;
SELECT DATABASE();

CREATE TABLE runners (
  -- unique id
  id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
  username VARCHAR(255) UNIQUE NOT NULL,
  email VARCHAR(255) NOT NULL,

  -- personal info
  first_name VARCHAR(255),
  last_name VARCHAR(255) NOT NULL,
  gender CHAR(1),
  dob DATE,

  -- run info
  total_miles DOUBLE DEFAULT 0,
  last_activity DATETIME
);
DESC runners;
```

Solution 2

<Link to Problem 2>

```
INSERT INTO runners
  (username, email, first_name, last_name)
VALUES
   ('dsmith', 'dora@gmail.com', 'Dora', 'Smith'),
   ('ron123', 'ron123@yahoo.com', 'Ron', 'Long'),
   ('lisaaa', 'lisaroy@comcast.net', 'Lisa', 'Roy'),
   ('tlong', 'tlong@gmail.com', 'Tara', 'Long');
SELECT * FROM runners;
```

Solution 3

<Link to Problem 3>

```
UPDATE runners SET dob='1985-07-10'
WHERE username='ron123';

UPDATE runners SET gender='F'
WHERE username='dsmith' or username='lisaaa';

UPDATE runners SET total_miles=3.1;

SELECT * FROM runners;
```

Solution 4

<Link to Problem 4>

```
DELETE FROM runners
WHERE last_name='long';

SELECT * FROM runners;
```

Solution 5

<Link to Problem 5>

```
SELECT * FROM friends WHERE age >= 30;

SELECT * FROM friends WHERE age % 2 = 1;
```

Solution 6

<Link to Problem 6>

Often times there will be multiple ways to solve a problem, some more efficient than the others. I wanted to demonstrate that through this simple problem.

Approach 1 - do not do this way!
It is given that there is only one friend matching the criteria. Typically `id` is unique for every row, so if we somehow know the `id` of the friend we can get the other information. One way is, do a `SELECT *` and go through the rows manually to find the `id` of the friend in question. In this case, the `id` of the oldest friend happens to be 1, so you could write the query as follows.

```
SELECT
  CONCAT_WS(' ', first_name, last_name) AS name,
  email
FROM friends
WHERE id = 1;
```

The issue with this approach is that it involves manually looking at the table to find the `id`, which was possible here because there are only a few rows of data. However, in real world application,

the table will have thousands or even millions of rows, and it would be practically impossible to go through all the rows. Also you want to avoid any manual work.

Approach 2 - a better way, but still not the most efficient way!
A better approach would be to first write a query to find the `id` of the oldest friend (step 1), and then use that result in the first approach (step 2). Both these steps can be combined in one query as follows:

```
SELECT
  CONCAT_WS(' ', first_name, last_name) AS name,
  email
FROM friends
WHERE id = (SELECT id FROM friends WHERE age = (SELECT MAX(age) FROM friends)); ← Nested SELECT, called 'subqueries'.

Explanation - break it into step by step:
```

$$id = \underbrace{(SELECT\ \ id\ \ FROM\ \ friends\ \ WHERE\ \ age\ \ =\ \ \underbrace{(SELECT\ \ MAX(age)\ \ FROM\ \ friends)}_{42})}_{1}$$

In fact, we do not have to use the `id` to look, we can use the max age instead as follows:

```
SELECT
  CONCAT_WS(' ', first_name, last_name) AS name,
  email
FROM friends
WHERE age = (SELECT MAX(age) FROM friends);

Explanation - break it into step by step:
```

$$age = \underbrace{(SELECT\ \ MAX(age)\ \ FROM\ \ friends)}_{42}$$

Approach 3 - best way!
Queries involving subqueries are typically slower. For this problem, there is a more efficient (faster) way than using subqueries.

```
SELECT
  CONCAT_WS(' ', first_name, last_name) AS name,
  email
FROM friends
ORDER BY age DESC
LIMIT 1;
```

We are simply ordering by the `age`, in descending order, and then taking the first row. No subqueries needed.

However, if there were multiple friends matching the crieteria and we wanted all of them, but we don't know how many, then the second query of **Approach 2** would work better. So it depends on the problem - and with practice you get better at it!

Solution 7

<Link to Problem 7>

```
SELECT
  CONCAT_WS(' ', first_name, last_name) AS name,
  email
FROM friends
WHERE gender = 'M' ← Filtering by gender
ORDER BY age DESC
LIMIT 1;
```

Solution 8

<Link to Problem 8>

```sql
SELECT
  CASE
    WHEN email LIKE '%@gmail.com' THEN 'gmail'
    WHEN email LIKE '%@yahoo.com' THEN 'yahoo'
    ELSE 'other'
  END as provider,
  COUNT(*) AS total_users
FROM friends
GROUP BY provider
ORDER BY total_users DESC;
```

Solution 9

<Link to Problem 9>

```sql
-- part 1
SELECT DISTINCT
  SUBSTRING_INDEX(SUBSTRING_INDEX(email, '@', -1), '.', 1) AS provider
FROM friends
ORDER BY provider;

-- part 2
SELECT
  SUBSTRING_INDEX(SUBSTRING_INDEX(email, '@', -1), '.', 1) AS provider,
  COUNT(*) AS total_users
FROM friends
GROUP BY provider
ORDER BY total_users DESC;

Explanation - example of how it works:
```

$$\underbrace{SUBSTRING\_INDEX(\underbrace{SUBSTRING\_INDEX('dora@gmail.com',\ '@',\ -1)}_{\text{Use '@' as delimiter and take the last element = 'gmail.com'}},\ '.',\ 1)}_{\text{Use '.' as delimiter and take the first element = 'gmail'}}$$

Note that we have used a new string function here `SUBSTRING_INDEX`, which I expected you to find out by searching online! Here are some examples of how it works.

```
mysql> SELECT SUBSTRING_INDEX('a.b.c', '.', 2);
+----------------------------------+
| SUBSTRING_INDEX('a.b.c', '.', 2) |
+----------------------------------+
| a.b                              |
+----------------------------------+
1 row in set (0.00 sec)

mysql> SELECT SUBSTRING_INDEX('a.b.c', '.', -2);
+-----------------------------------+
| SUBSTRING_INDEX('a.b.c', '.', -2) |
+-----------------------------------+
| b.c                               |
+-----------------------------------+
1 row in set (0.00 sec)

mysql>
```

Solution 10

<Link to Problem 10>

```sql
SELECT
  id,
  CONCAT_WS(
      ' ',
      first_name,
      CONCAT(UPPER(SUBSTRING(last_name, 1, 1)), '.')
    ) AS name,
  CONCAT(
      SUBSTRING(email, 1, 1),
      '***',
      SUBSTRING(SUBSTRING_INDEX(email, '@', 1), -1),
      '@',
      SUBSTRING_INDEX(email, '@', -1)
    ) AS 'masked email'
FROM friends;
```

It's essentially using a bunch of nested string functions, just for practice.

Solution 11

<Link to Problem 11>

```
SELECT
  id,
  machineid,
  start_time,
  ROUND(
    HOUR(TIMEDIFF(stop_time, start_time))
    + MINUTE(TIMEDIFF(stop_time, start_time))/60 ← Converting minutes to hours
    + SECOND(TIMEDIFF(stop_time, start_time))/360 ← Converting seconds to hours
    , 2) AS run_time ← Rounding to 2 decimal places
FROM machinelogs
ORDER BY run_time DESC;
```

Solution 12

<Link to Problem 12>

```
SELECT
  id,
  machineid,
  stop_time
FROM machinelogs
WHERE stop_mode = 'failure';
```

Solution 13

<Link to Problem 13>

```
SELECT
  users.id,
  email,
  first_name,
  age,
  gender,
  address_state
FROM users
INNER JOIN profiles
  ON users.id = profiles.user_id
ORDER BY first_name;
```

Note, we are doing `INNER JOIN` to include only the users who have a profile. Or, we could also do `LEFT JOIN users` with the `profiles` table as follows, and get the same result.

```
SELECT
  users.id,
  email,
  first_name,
  age,
  gender,
  address_state
FROM profiles
LEFT JOIN users
  ON profiles.user_id = users.id
ORDER BY first_name;
```

Solution 14

<Link to Problem 14>

Breakdown the problem and build the solution query step by step. Here is how I did this one.

First, do a quick `SELECT *` from users, and then we will build on top of that.

```
SELECT * ←
FROM users; ←

-- the result should look something like this
+----+--------------------+------------+-----------+
| id | email              | first_name | last_name |
+----+--------------------+------------+-----------+
|  1 | dora@gmail.com     | Dora       | Smith     |
```

```
|  2 | ron123@comcast.net | Ron        | Long      |
|  3 | lisaroy@gmail.com  | Lisa       | Roy       |
|  4 | dg@yahoo.com       | Dora       | G.        |
|  5 | tlong@gmail.com    | Tara       | Long      |
|  6 | ryan99@aol.com     | Ryan       | Davis     |
|  7 | ron1st@comcast.net | Ron        | long      |
+----+--------------------+------------+-----------+
7 rows in set (0.00 sec)
```

Next, `JOIN` the profiles.

```
SELECT *
FROM users
INNER JOIN profiles ←
  ON users.id = profiles.user_id; ←

-- the result should look something like this
+----+--------------------+------------+-----------+----+---------+------+--------+---------------+
| id | email              | first_name | last_name | id | user_id | age  | gender | address_state |
+----+--------------------+------------+-----------+----+---------+------+--------+---------------+
|  1 | dora@gmail.com     | Dora       | Smith     |  1 |       1 |   42 | F      | CA            |
|  4 | dg@yahoo.com       | Dora       | G.        |  2 |       4 |   21 | F      | CA            |
|  2 | ron123@comcast.net | Ron        | Long      |  3 |       2 |   35 | M      | NY            |
|  7 | ron1st@comcast.net | Ron        | long      |  4 |       7 |   30 | M      | CA            |
|  3 | lisaroy@gmail.com  | Lisa       | Roy       |  5 |       3 |   29 | F      | CA            |
|  5 | tlong@gmail.com    | Tara       | Long      |  6 |       5 |   29 | F      | MN            |
+----+--------------------+------------+-----------+----+---------+------+--------+---------------+
6 rows in set (0.01 sec)
```

Next, add the filter `WHERE`, and `ORDER`.

```
SELECT *
FROM users
INNER JOIN profiles
  ON users.id = profiles.user_id
WHERE address_state = 'CA' ←
ORDER BY first_name; ←

-- the result should look something like this
+----+--------------------+------------+-----------+----+---------+------+--------+---------------+
| id | email              | first_name | last_name | id | user_id | age  | gender | address_state |
+----+--------------------+------------+-----------+----+---------+------+--------+---------------+
|  1 | dora@gmail.com     | Dora       | Smith     |  1 |       1 |   42 | F      | CA            |
|  4 | dg@yahoo.com       | Dora       | G.        |  2 |       4 |   21 | F      | CA            |
|  3 | lisaroy@gmail.com  | Lisa       | Roy       |  5 |       3 |   29 | F      | CA            |
|  7 | ron1st@comcast.net | Ron        | long      |  4 |       7 |   30 | M      | CA            |
+----+--------------------+------------+-----------+----+---------+------+--------+---------------+
4 rows in set (0.01 sec)
```

Next, refine the selection, step by step. First let's just select the email.

```
SELECT
  email ←
FROM users
INNER JOIN profiles
  ON users.id = profiles.user_id
WHERE address_state = 'CA'
ORDER BY first_name;

-- the result should look something like this
+--------------------+
| email              |
+--------------------+
| dora@gmail.com     |
| dg@yahoo.com       |
| lisaroy@gmail.com  |
| ron1st@comcast.net |
+--------------------+
4 rows in set (0.00 sec)
```

Next, we can see that the custom message is basically concat-with-separator a bunch of fields. So, first add a skeleton `CONCAT_WS` with minimal fields, and then we can start filling in.

```
SELECT
  email,
  CONCAT_WS( ←
    ' ', ←
    'Hello', ←
    first_name) AS message ←
FROM users
INNER JOIN profiles
  ON users.id = profiles.user_id
```

```
WHERE address_state = 'CA'
ORDER BY first_name;

-- the result should look something like this
+--------------------+------------+
| email              | message    |
+--------------------+------------+
| dora@gmail.com     | Hello Dora |
| dg@yahoo.com       | Hello Dora |
| lisaroy@gmail.com  | Hello Lisa |
| ron1st@comcast.net | Hello Ron  |
+--------------------+------------+
4 rows in set (0.01 sec)
```

Next, we need to take care of Mr or Ms with a `CASE`. Also, `CONCAT` a comma after the first name.

```
SELECT
  email,
  CONCAT_WS(
    ' ',
    'Hello',
    CASE ←
      WHEN gender = 'M' THEN 'Mr.' ←
      ELSE 'Ms.' ←
    END, ←
    CONCAT(first_name, ',')) AS message ←
FROM users
INNER JOIN profiles
  ON users.id = profiles.user_id
WHERE address_state = 'CA'
ORDER BY first_name;

-- the result should look something like this
+--------------------+-----------------+
| email              | message         |
+--------------------+-----------------+
| dora@gmail.com     | Hello Ms. Dora, |
| dg@yahoo.com       | Hello Ms. Dora, |
| lisaroy@gmail.com  | Hello Ms. Lisa, |
| ron1st@comcast.net | Hello Mr. Ron,  |
+--------------------+-----------------+
4 rows in set (0.00 sec)
```

As you can see, we are getting there ... the custom message is taking shape now!

After a few more steps, adding in the other parts of the custom message, we have the final query as follows.

```
SELECT
  email,
  CONCAT_WS(
    ' ',
    'Hello',
    CASE
      WHEN gender = 'M' THEN 'Mr.'
      ELSE 'Ms.'
    END,
    CONCAT(first_name, ','),
    'Use the coupon code', ←
    CONCAT( ←
      UPPER(SUBSTRING_INDEX(email, '@', 1)), ←
      '_', ←
      CASE ←
        WHEN gender='F' AND age > 25 THEN 15 ←
        ELSE 10 ←
      END), ←
    'to get a', ←
    CONCAT( ←
      CASE ←
        WHEN gender='F' AND age > 25 THEN 15 ←
        ELSE 10 ←
      END, ←
      '%'), ←
    'discount.' ←
  ) AS message
FROM users
INNER JOIN profiles
  ON users.id = profiles.user_id
WHERE address_state = 'CA'
ORDER BY first_name;

-- the final result should look something like this
+--------------------+-------------------------------------------------------------------------+
| email              | message                                                                 |
```

```
+--------------------+-------------------------------------------------------------------------+
| dora@gmail.com     | Hello Ms. Dora, Use the coupon code DORA_15 to get a 15% discount.      |
| dg@yahoo.com       | Hello Ms. Dora, Use the coupon code DG_10 to get a 10% discount.        |
| lisaroy@gmail.com  | Hello Ms. Lisa, Use the coupon code LISAROY_15 to get a 15% discount.   |
| ron1st@comcast.net | Hello Mr. Ron, Use the coupon code RON1ST_10 to get a 10% discount.     |
+--------------------+-------------------------------------------------------------------------+
4 rows in set (0.00 sec)
```

As you can see, we can write powerful queries by combining simple things we have learned! Using proper formatting and adding bits at a time makes it a lot easier to write queries like this.

Solution 15

<Link to Problem 15>

Let's first join the relevant tables to see the orders.

```
SELECT
  users.id,
  email,
  first_name,
  address_state,
  dollar_amount
FROM users
LEFT JOIN profiles
  ON users.id = profiles.user_id
LEFT JOIN orders
  ON users.id = orders.user_id;

-- result should look something like this
+----+--------------------+------------+---------------+---------------+
| id | email              | first_name | address_state | dollar_amount |
+----+--------------------+------------+---------------+---------------+
|  1 | dora@gmail.com     | Dora       | CA            |            24 |
|  1 | dora@gmail.com     | Dora       | CA            |            42 |
|  2 | ron123@comcast.net | Ron        | NY            |            65 |
|  3 | lisaroy@gmail.com  | Lisa       | CA            |          NULL |
|  4 | dg@yahoo.com       | Dora       | CA            |          NULL |
|  5 | tlong@gmail.com    | Tara       | MN            |            89 |
|  5 | tlong@gmail.com    | Tara       | MN            |            11 |
|  5 | tlong@gmail.com    | Tara       | MN            |            38 |
|  5 | tlong@gmail.com    | Tara       | MN            |            38 |
|  6 | ryan99@aol.com     | Ryan       | NULL          |          NULL |
|  7 | ron1st@comcast.net | Ron        | CA            |            50 |
|  7 | ron1st@comcast.net | Ron        | CA            |            69 |
+----+--------------------+------------+---------------+---------------+
12 rows in set (0.00 sec)
```

Next, group and count.

```
SELECT
  users.id,
  email,
  first_name,
  address_state,
  COUNT(dollar_amount) AS count_
FROM users
LEFT JOIN profiles
  ON users.id = profiles.user_id
LEFT JOIN orders
  ON users.id = orders.user_id
GROUP BY users.id
ORDER BY count_ DESC;

-- and this should produce the desired result
+----+--------------------+------------+---------------+--------+
| id | email              | first_name | address_state | count_ |
+----+--------------------+------------+---------------+--------+
|  5 | tlong@gmail.com    | Tara       | MN            |      4 |
|  1 | dora@gmail.com     | Dora       | CA            |      2 |
|  7 | ron1st@comcast.net | Ron        | CA            |      2 |
|  2 | ron123@comcast.net | Ron        | NY            |      1 |
|  3 | lisaroy@gmail.com  | Lisa       | CA            |      0 |
|  4 | dg@yahoo.com       | Dora       | CA            |      0 |
|  6 | ryan99@aol.com     | Ryan       | NULL          |      0 |
+----+--------------------+------------+---------------+--------+
7 rows in set (0.01 sec)
```

Notes:

- We are using `LEFT JOIN` to keep all the users, even those with no orders

- If you do `COUNT(*)` instead of `COUNT(dollar_amount)`, you will get a count of 1 even for users with no orders (ref. Section **??**).

Finally, getting the user with the most number of orders is straightforward at this point.

```
SELECT
  email,
  CONCAT_WS(' ', first_name, last_name) AS name
FROM users
LEFT JOIN profiles
  ON users.id = profiles.user_id
LEFT JOIN orders
  ON users.id = orders.user_id
GROUP BY users.id
ORDER BY COUNT(dollar_amount) DESC
LIMIT 1;

-- and this should produce the desired result
+-----------------+-----------+
| email           | name      |
+-----------------+-----------+
| tlong@gmail.com | Tara Long |
+-----------------+-----------+
1 row in set (0.00 sec)
```

Solution 16

<Link to Problem 16>

```
(A) Total sale
SELECT
  MONTHNAME(order_date) AS month,
  SUM(dollar_amount) AS total
FROM orders
GROUP BY month;

(B) Sales over $100
SELECT
  MONTHNAME(order_date) AS month,
  SUM(dollar_amount) AS total
FROM orders
GROUP BY month
HAVING total > 100;

(C) Maximum sale month
SELECT
  MONTHNAME(order_date) AS month
FROM orders
GROUP BY month
ORDER BY SUM(dollar_amount) DESC
LIMIT 1;
```

Solution 17

<Link to Problem 17>

```
(A) Books having > 100 pages
SELECT *
FROM books
WHERE pages > 100;

-- result
+----+-------------------------------------+-------+
| id | title                               | pages |
+----+-------------------------------------+-------+
|  1 | Cook with Lisa and Ryan             |   210 |
|  2 | Ryan's adventure                    |   150 |
|  4 | Ryan, Dora, and Ron's giude to SQL  |   350 |
+----+-------------------------------------+-------+
3 rows in set (0.00 sec)

(B) Authors having >1 books
SELECT
  first_name,
  last_name,
  email,
  age,
  COUNT(book_id) AS no_of_books
FROM users ← Starting with the users table
LEFT JOIN profiles ← LEFT JOIN because some users may not have profiles, but may have authored books, and so we want to keep them
  ON users.id = profiles.user_id
INNER JOIN authorships ← INNER JOIN because we need to keep only the users who have authored books
```

```
  ON users.id = authorships.author_id
GROUP BY author_id
HAVING no_of_books > 1
ORDER BY no_of_books DESC, last_name ASC;

-- result
+------------+-----------+-----------------+------+-------------+
| first_name | last_name | email           | age  | no_of_books |
+------------+-----------+-----------------+------+-------------+
| Ryan       | Davis     | ryan99@aol.com  | NULL |           3 | ← If we did INNER JOIN profiles, we would have lost this row
| Tara       | Long      | tlong@gmail.com |   29 |           2 |
| Dora       | Smith     | dora@gmail.com  |   42 |           2 |
+------------+-----------+-----------------+------+-------------+
3 rows in set (0.00 sec)

(C) Books having >1 authors
SELECT
  title,
  COUNT(author_id) AS no_of_authors
FROM books
INNER JOIN authorships ← Note: If you did LEFT JOIN, you will get 'ERROR ...; this is incompatible with sql_mode=only_full_group_by'
  ON books.id = authorships.book_id
GROUP BY book_id
HAVING no_of_authors > 1
ORDER BY no_of_authors DESC, title ASC;

-- result
+-----------------------------------+----------------+
| title                             | no_of_authors |
+-----------------------------------+----------------+
| Ryan, Dora, and Ron's giude to SQL |              3 |
| Cook with Lisa and Ryan           |              2 |
| Swim with Tara and run with Dora  |              2 |
+-----------------------------------+----------------+
3 rows in set (0.00 sec)
```

Solution 18

<Link to Problem 18>

(A) Users who have commented more than once on a given book

We need to group by user id and book id, and then count the number of comments in each group.

```
SELECT
  CONCAT_WS(' ', first_name, last_name) AS name,
  email,
  title,
  COUNT(comment) AS times_commented
FROM comments
INNER JOIN users
  ON comments.user_id = users.id
INNER JOIN books
  ON comments.book_id = books.id
GROUP BY user_id, book_id
HAVING times_commented > 1
ORDER BY times_commented DESC, email;

-- result
+------------+-----------------+-----------------------------------+-----------------+
| name       | email           | title                             | times_commented |
+------------+-----------------+-----------------------------------+-----------------+
| Dora Smith | dora@gmail.com  | Cook with Lisa and Ryan           |               3 |
| Dora Smith | dora@gmail.com  | Ryan, Dora, and Ron's giude to SQL |               2 |
| Tara Long  | tlong@gmail.com | 10 minutes workout with Tara      |               2 |
+------------+-----------------+-----------------------------------+-----------------+
3 rows in set (0.00 sec)
```

(B) Users who have never commented

We need to group by user id, and then check which users have never commented on any book.

```
SELECT
  users.id,
  CONCAT_WS(' ', first_name, last_name) AS name,
  email,
  COUNT(book_id) AS books_commented
FROM users
LEFT JOIN comments
  ON users.id = comments.user_id
GROUP BY users.id
```

```
HAVING books_commented = 0;

-- result
+----+------------+--------------------+-----------------+
| id | name       | email              | books_commented |
+----+------------+--------------------+-----------------+
|  3 | Lisa Roy   | lisaroy@gmail.com  |               0 |
|  6 | Ryan Davis | ryan99@aol.com     |               0 |
|  7 | Ron long   | ron1st@comcast.net |               0 |
+----+------------+--------------------+-----------------+
3 rows in set (0.01 sec)
```

(C) Users who have commented on all the books

This one is bit tricky. We could count the number of books in the table, and then compare that with the total number of comments by each user. But this won't work because a user could comment more than once on a book.

So, one approach is you can use a derived table (see `https://dev.mysql.com/doc/refman/8.0/en/derived-tables.html`) that contains unique pairs of (user id, book id).

```
SELECT
  users.id,
  CONCAT_WS(' ', first_name, last_name) AS name,
  email,
  COUNT(book_id) AS books_commented
FROM (
  SELECT DISTINCT
    book_id,
    user_id
  FROM comments
) AS unique_comments ← derived table approach
LEFT JOIN users
  ON unique_comments.user_id = users.id
GROUP BY user_id
HAVING books_commented = (SELECT COUNT(*) FROM books);

-- result
+------+------------+-----------------+-----------------+
| id   | name       | email           | books_commented |
+------+------------+-----------------+-----------------+
|    1 | Dora Smith | dora@gmail.com  |               5 |
|    5 | Tara Long  | tlong@gmail.com |               5 |
+------+------------+-----------------+-----------------+
2 rows in set (0.01 sec)
```

A simpler approach is just count distinct book ids (compare this query to B).

```
SELECT
  users.id,
  CONCAT_WS(' ', first_name, last_name) AS name,
  email,
  COUNT(DISTINCT(book_id)) AS books_commented ← COUNT(DISTINCT(...)) approach
FROM users
LEFT JOIN comments
  ON users.id = comments.user_id
GROUP BY users.id
HAVING books_commented = (SELECT COUNT(*) FROM books);

-- result is the same
+----+------------+-----------------+-----------------+
| id | name       | email           | books_commented |
+----+------------+-----------------+-----------------+
|  1 | Dora Smith | dora@gmail.com  |               5 |
|  5 | Tara Long  | tlong@gmail.com |               5 |
+----+------------+-----------------+-----------------+
2 rows in set (0.01 sec)
```

Index

T

U

V