Introduction to SQL

Qitian Liao

University of California, Berkeley

Contents

1 Manipulation

Get up and running with SQL by learning commands to manipulate data stored in relational databases.

1.1 Introduction to SQL

SQL, **S**tructured **Q**uery **L**anguage, is a programming language designed to manage data stored in relational databases. SQL operates through simple, declarative statements. This keeps data accurate and secure, and helps maintain the integrity of databases, regardless of size.

The SQL language is widely used today across web frameworks and database applications. Knowing SQL gives you the freedom to explore your data, and the power to make better decisions. By learning SQL, you will also learn concepts that apply to nearly every data storage system.

The statements covered in this course use SQLite Relational Database Management System (RDBMS). You can also access a glossary of all the SQL commands taught in this chapter.

1.2 Relational Databases

In one line of code, we can return information from a relational database.

```
SELECT * FROM celebs;
```

A *relational database* is a database that organizes information into one or more tables. Here, the relational database contains one table.

A *table* is a collection of data organized into rows and columns. Tables are sometimes referred to as *relations*. Here the table is `celebs`.

A *column* is a set of data values of a particular type. Here, `id`, `name`, and `age` are the columns.

A *row* is a single record in a table.

All data stored in a relational database is of a certain data type. Some of the most common data types are:

- `INTEGER`, a positive or negative whole number
- `TEXT`, a text string
- `DATE`, the date formatted as YYYY-MM-DD
- `REAL`, a decimal value

1.3 Statements

The code below is a SQL statement. A *statement* is text that the database recognizes as a valid command. Statements always end in a semicolon `;`.

```
CREATE TABLE table_name (
   column_1 data_type,
   column_2 data_type,
   column_3 data_type
);
```

Let us break down the components of a statement:

1. CREATE TABLE is a *clause*. Clauses perform specific tasks in SQL. By convention, clauses are written in capital letters. Clauses can also be referred to as commands.

2. table_name refers to the name of the table that the command is applied to.

3. (column_1 data_type, column_2 data_type, column_3 data_type) is a *parameter*. A parameter is a list of columns, data types, or values that are passed to a clause as an argument. Here, the parameter is a list of column names and the associated data type.

The structure of SQL statements vary. The number of lines used does not matter. A statement can be written all on one line, or split up across multiple lines if it makes it easier to read.

1.4 Create

CREATE statements allow us to create a new table in the database. You can use the CREATE statement anytime you want to create a new table from scratch. The statement below creates a new table named celebs.

```sql
CREATE TABLE celebs (
   id INTEGER,
   name TEXT,
   age INTEGER
);
```

1. CREATE TABLE is a clause that tells SQL you want to create a new table.

2. celebs is the name of the table.

3. (id INTEGER, name TEXT, age INTEGER) is a list of parameters defining each column, or attribute in the table and its data type:

   - id is the first column in the table. It stores values of data type INTEGER
   - name is the second column in the table. It stores values of data type TEXT
   - age is the third column in the table. It stores values of data type INTEGER

1.5 Insert

The INSERT statement inserts a new row into a table. You can use the INSERT statement when you want to add new records. The statement below enters a record for Justin Bieber into the celebs table.

```sql
INSERT INTO celebs (id, name, age)
VALUES (1, "Justin Bieber", 22);
```

1. INSERT INTO is a clause that adds the specified row or rows.

2. celebs is the name of the table the row is added to.

3. (id, name, age) is a parameter identifying the columns that data will be inserted into.

4. VALUES is a clause that indicates the data being inserted. (1, "Justin Bieber", 22) is a parameter identifying the values being inserted.

- 1 is an integer that will be inserted into the id column
- "Justin Bieber" is text that will be inserted into the name column
- 22 is an integer that will be inserted into the age column

1.6 Select

SELECT statements are used to fetch data from a database. In the statement below, SELECT returns all data in the name column of the celebs table.

```sql
SELECT name FROM celebs;
```

1. SELECT is a clause that indicates that the statement is a query. You will use SELECT every time you query data from a database.
2. name specifies the column to query data from.
3. FROM celebs specifies the name of the table to query data from. In this statement, data is queried from the celebs table.

You can also query data from all columns in a table with SELECT.

```sql
SELECT * FROM celebs;
```

* is a special wildcard character that we have been using. It allows you to select every column in a table without having to name each one individually. Here, the result set contains every column in the celebs table.

SELECT statements always return a new table called the *result set*.

1.7 Alter

The ALTER TABLE statement adds a new column to a table. You can use this command when you want to add columns to a table. The statement below adds a new column twitter_handle to the celebs table.

```sql
ALTER TABLE celebs
ADD COLUMN twitter\_handle TEXT;
```

1. ALTER TABLE is a clause that lets you make the specified changes.
2. celebs is the name of the table that is being changed.
3. ADD COLUMN is a clause that lets you add a new column to a table:
   - twitter_handle is the name of the new column being added
   - TEXT is the data type for the new column
4. NULL is a special value in SQL that represents missing or unknown data. Here, the rows that existed before the column was added have NULL (∅) values for twitter_handle.

1.8 Update

The UPDATE statement edits a row in a table. You can use the UPDATE statement when you want to change existing records. The statement below updates the record with an id value of 4 to have the twitter_handle @taylorswift13.

```
UPDATE celebs
SET twitter_handle = "@taylorswift13"
WHERE id = 4;
```

1. UPDATE is a clause that edits a row in the table.
2. celebs is the name of the table.
3. SET is a clause that indicates the column to edit.
    - twitter_handle is the name of the column that is going to be updated
    - @taylorswift13 is the new value that is going to be inserted into the twitter_handle column.
    - WHERE is a clause that indicates which row(s) to update with the new column value. Here the row with a 4 in the id column is the row that will have the twitter_handle updated to @taylorswift13.

1.9 Delete

The DELETE FROM statement deletes one or more rows from a table. You can use the statement when you want to delete existing records. The statement below deletes all records in the celeb table with no twitter_handle:

```
DELETE FROM celebs
WHERE twitter_handle IS NULL;
```

1. DELETE FROM is a clause that lets you delete rows from a table.
2. celebs is the name of the table we want to delete rows from.
3. WHERE is a clause that lets you select which rows you want to delete. Here we want to delete all of the rows where the twitter_handle column IS NULL.
4. IS NULL is a condition in SQL that returns true when the value is NULL and false otherwise.

1.10 Constraints

Constraints that add information about how a column can be used are invoked after specifying the data type for a column. They can be used to tell the database to reject inserted data that does not adhere to a certain restriction. The statement below sets *constraints* on the celebs table.

```
CREATE TABLE celebs (
  id INTEGER PRIMARY KEY,
  name TEXT UNIQUE,
  date_of_birth TEXT NOT NULL,
  date_of_death TEXT DEFAULT "Not Applicable"
);
```

1. `PRIMARY KEY` columns can be used to uniquely identify the row. Attempts to insert a row with an identical value to a row already in the table will result in a *constraint violation* which will not allow you to insert the new row.

2. `UNIQUE` columns have a different value for every row. This is similar to `PRIMARY KEY` except a table can have many different `UNIQUE` columns.

3. `NOT NULL` columns must have a value. Attempts to insert a row without a value for a `NOT NULL` column will result in a constraint violation and the new row will not be inserted.

4. `DEFAULT` columns take an additional argument that will be the assumed value for an inserted row if the new row does not specify a value for that column.

1.11 Review

We have learned six commands commonly used to manage data stored in a relational database and how to set constraints on such data.

SQL is a programming language designed to manipulate and manage data stored in relational databases.

- A *relational database* is a database that organizes information into one or more tables.
- A *table* is a collection of data organized into rows and columns.

A *statement* is a string of characters that the database recognizes as a valid command.

- `CREATE TABLE` creates a new table.
- `INSERT INTO` adds a new row to a table.
- `SELECT` queries data from a table.
- `ALTER TABLE` changes an existing table.
- `UPDATE` edits a row in a table.
- `DELETE FROM` deletes rows from a table.

Constraints add information about how a column can be used.

2 Queries

Now we learn the most commonly used SQL commands to query a table in a database

2.1 Introduction

Now, we will be learning different SQL commands to query a single table in a database.

One of the core purposes of the SQL language is to retrieve information stored in a database. This is commonly referred to as querying. Queries allow us to communicate with the database by asking questions and returning a result set with data relevant to the question.

We will be querying a database with one table named movies.

Fun fact: IBM started out SQL as SEQUEL (**S**tructured **E**nglish **QUE**ry **L**anguage) in the 1970's to query databases.

2.2 Select

Previously, we learned that SELECT is used every time you want to query data from a database and * means all columns.

Suppose we are only interested in two of the columns. We can select individual columns by their names (separated by a comma):

```sql
SELECT column1, column2
FROM table_name;
```

To make it easier to read, we moved FROM to another line.

Line breaks do not mean anything specific in SQL. We could write this entire query in one line, and it would run just fine.

2.3 As

Knowing how SELECT works, suppose we have the code below:

```sql
SELECT name AS "Titles"
FROM movies;
```

AS is a keyword in SQL that allows you to rename a column or table using an alias. The new name can be anything you want as long as you put it inside of single quotes. Here we renamed the name column as Titles.

- Although it is not always necessary, it's best practice to surround your aliases with single quotes.
- When using AS, the columns are not being renamed in the table. The aliases only appear in the result.

2.4 Distinct

When we are examining data in a table, it can be helpful to know what distinct values exist in a particular column.

DISTINCT is used to return unique values in the output. It filters out all duplicate values in the specified column(s).

For instance,

```sql
SELECT tools
FROM inventory;
```

might produce:

tools
Hammer
Nails
Nails
Nails

By adding DISTINCT before the column name,

```sql
SELECT DISTINCT tools
FROM inventory;
```

the result would now be:

tools
Hammer
Nails

Filtering the results of a query is an important skill in SQL. It is easier to see the different possible genres in the movie table after the data has been filtered than to scan every row in the table.

```sql
SELECT DISTINCT genre
FROM movies;
```

2.5 Where

We can restrict our query results using the WHERE clause in order to obtain only the information we want.

Following this format, the statement below filters the result set to only include top rated movies (IMDb ratings greater than 8):

```sql
SELECT *
FROM movies
WHERE imdb_rating > 8;
```

1. WHERE clause filters the result set to only include rows where the following *condition* is true.
2. imdb_rating > 8 is the condition. Here, only rows with a value greater than 8 in the imdb_rating column will be returned.

The > is an operator. Operators create a condition that can be evaluated as either *true* or *false*.

Comparison operators used with the WHERE clause are:

- = equal to

- != not equal to
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

2.6 Like

LIKE can be a useful operator when you want to compare similar values.

The movies table contains two films with similar titles, 'Se7en' and 'Seven'.

In order to select all movies that start with 'Se' and end with 'en' and have exactly one character in the middle,

```sql
SELECT *
FROM movies
WHERE name LIKE "Se_en";
```

- LIKE is a special operator used with the WHERE clause to search for a specific pattern in a column.
- name LIKE 'Se_en' is a condition evaluating the name column for a specific pattern.
- Se_en represents a pattern with a wildcard character.

The _ means you can substitute any individual character here without breaking the pattern. The names Seven and Se7en both match this pattern.

The percentage sign % is another wildcard character that can be used with LIKE.

This statement below filters the result set to only include movies with names that begin with the letter 'A':

```sql
SELECT *
FROM movies
WHERE name LIKE "A%";
```

% is a wildcard character that matches zero or more missing letters in the pattern. For example:

- A% matches all movies with names that begin with letter 'A'
- %a matches all movies that end with 'a'

We can also use % both before and after a pattern:

```sql
SELECT *
FROM movies
WHERE name LIKE "%man%";
```

Here, any movie that contains the word 'man' in its name will be returned in the result.

LIKE is not case sensitive. 'Batman' and 'Man of Steel' will both appear in the result of the query above.

2.7 Is Null

More often than not, the data you encounter will have missing values. Unknown values are indicated by NULL.

It is not possible to test for NULL values with comparison operators, such as = and !=.

Instead, we will have to use these operators:

- IS NULL
- IS NOT NULL

To filter for all movies *with* an IMDb rating:

```sql
SELECT name
FROM movies
WHERE imdb_rating IS NOT NULL;
```

2.8 Between

The BETWEEN operator is used in a WHERE clause to filter the result set within a certain *range*. It accepts two values that are either numbers, text or dates.

For example, this statement filters the result set to only include movies with years from 1990 up to, and including 1999.

```sql
SELECT *
FROM movies
WHERE year BETWEEN 1990 AND 1999;
```

When the values are text, BETWEEN filters the result set for within the alphabetical range.

In this statement, BETWEEN filters the result set to only include movies with names that begin with the letter ‘A’ up to, but not including ones that begin with ‘J’.

```sql
SELECT *
FROM movies
WHERE name BETWEEN "A" AND "J";
```

However, if a movie has a name of simply ‘J’, it would actually match. This is because BETWEEN goes *up to* the second value — up to ‘J’. So the movie named ‘J’ would be included in the result set but not ‘Jaws’.

2.9 And

Sometimes we want to combine multiple conditions in a WHERE clause to make the result set more specific and useful.

One way of doing this is to use the AND operator. Here, we use the AND operator to only return 90’s romance movies.

```sql
SELECT *
FROM movies
WHERE year BETWEEN 1990 AND 1999
   AND genre = "romance";
```

- year BETWEEN 1990 AND 1999 is the 1st condition.
- genre = 'romance' is the 2nd condition.
- AND combines the two conditions.

With AND, both conditions must be true for the row to be included in the result.

2.10 Or

Similar to AND, the OR operator can also be used to combine multiple conditions in WHERE, but there is a fundamental difference:

- AND operator displays a row if *all* the conditions are true.
- OR operator displays a row if *any* condition is true.

Suppose we want to check out a new movie or something action-packed:

```sql
SELECT *
FROM movies
WHERE year > 2014
   OR genre = "action";
```

- year > 2014 is the first condition.
- genre = "action" is the second condition.
- OR combines the two conditions.

With OR, if any of the conditions are true, then the row is added to the result.

2.11 Order By

It is often useful to list the data in our result set in a particular order.

We can *sort* the results using ORDER BY, either alphabetically or numerically. Sorting the results often makes the data more useful and easier to analyze.

For example, if we want to sort everything by the movie's title from A through Z:

```sql
SELECT *
FROM movies
ORDER BY name;
```

- ORDER BY is a clause that indicates you want to sort the result set by a particular column.
- name is the specified column.

Sometimes we want to sort things in a decreasing order. For example, if we want to select all of the well-received movies, sorted from highest to lowest by their year:

```sql
SELECT *
FROM movies
WHERE imdb_rating > 8
ORDER BY year DESC;
```

- DESC is a keyword used in ORDER BY to sort the results in *descending order* (high to low or Z-A).
- ASC is a keyword used in ORDER BY to sort the results in *ascending order* (low to high or A-Z).

The column that we ORDER BY does not even have to be one of the columns that we are displaying.

Note: ORDER BY always goes after WHERE (if WHERE is present).

2.12 Limit

Most SQL tables contain hundreds of thousands of records. In those situations, it becomes important to cap the number of rows in the result.

For instance, imagine that we just want to see a few examples of records.

```sql
SELECT *
FROM movies
LIMIT 10;
```

LIMIT is a clause that lets you specify the maximum number of rows the result set will have. This saves space on our screen and makes our queries run faster.

Here, we specify that the result set cannot have more than 10 rows.

LIMIT always goes at the very end of the query. Also, it is not supported in all SQL databases.

2.13 Case

A CASE statement allows us to create different outputs (usually in the SELECT statement). It is SQL's way of handling if-then logic.

Suppose we want to condense the ratings in movies to three levels:

- *If the rating is above 8, then it is Fantastic.*
- *If the rating is above 6, then it is Poorly Received.*
- *Else, Avoid at All Costs.*

```sql
SELECT name,
 CASE
  WHEN imdb_rating > 8 THEN "Fantastic"
  WHEN imdb_rating > 6 THEN "Poorly Received"
  ELSE "Avoid at All Costs"
 END
FROM movies;
```

- Each WHEN tests a condition and the following THEN gives us the string if the condition is true.
- The ELSE gives us the string if all the above conditions are false.
- The CASE statement must end with END.

In the result, you have to scroll right because the column name is very long. To shorten it, we can rename the column to 'Review' using AS:

```sql
SELECT name,
 CASE
  WHEN imdb_rating > 8 THEN "Fantastic"
  WHEN imdb_rating > 6 THEN "Poorly Received"
  ELSE "Avoid at All Costs"
 END AS "Review"
FROM movies;
```

2.14 Review

We just learned how to query data from a database using SQL. We also learned how to filter queries to make the information more specific and useful. Let us summarize:

- SELECT is the clause we use every time we want to query information from a database.
- AS renames a column or table.
- DISTINCT return unique values.
- WHERE is a popular command that lets you filter the results of the query based on conditions that you specify.
- LIKE and BETWEEN are special operators.
- AND and OR combines multiple conditions.
- ORDER BY sorts the result.
- LIMIT specifies the maximum number of rows that the query will return.
- CASE creates different outputs.

3 Aggregate Functions

Now we learn how to use SQL to perform calculations during a query.

3.1 Introduction

We have learned how to write queries to retrieve information from the database. Now, we are going to learn how to perform calculations using SQL.

Calculations performed on multiple rows of a table are called **aggregates**.

In this chapter, we will use a table named `fake_apps` which is made up of fake mobile applications data.

Here is a quick preview of some important aggregates that we will cover in this chapter:

- `COUNT()`: count the number of rows
- `SUM()`: the sum of the values in a column
- `MAX()`/`MIN()`: the largest/smallest value
- `AVG()`: the average of the values in a column
- `ROUND()`: round the values in the column

3.2 Count

The fastest way to calculate how many rows are in a table is to use the `COUNT()` function.

`COUNT()` is a function that takes the name of a column as an argument and counts the number of non-empty values in that column.

```sql
SELECT COUNT(*)
FROM table_name;
```

Here, we want to count every row, so we pass `*` as an argument inside the parenthesis.

3.3 Sum

SQL makes it easy to add all values in a particular column using `SUM()`.

`SUM()` is a function that takes the name of a column as an argument and returns the sum of all the values in that column.

The following code computes the total number of downloads for all of the apps combined

```sql
SELECT SUM(downloads)
FROM fake_apps;
```

This adds all values in the `downloads` column.

3.4 Max / Min

The `MAX()` and `MIN()` functions return the highest and lowest values in a column, respectively.

The following code computes the number of downloads the most popular app have?

```sql
SELECT MAX(downloads)
FROM fake_apps;
```

MAX() takes the name of a column as an argument and returns the largest value in that column. Here, we returned the largest value in the downloads column.

MIN() works the same way but it does the exact opposite; it returns the smallest value.

3.5 Average

SQL uses the AVG() function to quickly calculate the average value of a particular column.

The statement below returns the average number of downloads for an app in our database:

```sql
SELECT AVG(downloads)
FROM fake_apps;
```

The AVG() function works by taking a column name as an argument and returns the average value for that column.

3.6 Round

By default, SQL tries to be as precise as possible without rounding. We can make the result table easier to read using the ROUND() function.

ROUND() function takes two arguments inside the parenthesis:

1. a column name

2. an integer

It rounds the values in the column to the number of decimal places specified by the integer.

```sql
SELECT ROUND(price, 0)
FROM fake_apps;
```

Here, we pass the column price and integer 0 as arguments. SQL rounds the values in the column to 0 decimal places in the output.

3.7 Group By

Oftentimes, we will want to calculate an aggregate for data with certain characteristics.

For instance, we might want to know the mean IMDb ratings for all movies each year. We could calculate each number by a series of queries with different WHERE statements, like so:

```sql
SELECT AVG(imdb_rating)
FROM movies
WHERE year = 1999;

SELECT AVG(imdb_rating)
FROM movies
WHERE year = 2000;

SELECT AVG(imdb_rating)
```

```
FROM movies
WHERE year = 2001;
and so on.
```

However, we can use GROUP BY to do this in a single step:

```
SELECT year,
   AVG(imdb_rating)
FROM movies
GROUP BY year
ORDER BY year;
```

GROUP BY is a clause in SQL that is used with aggregate functions. It is used in collaboration with the SELECT statement to arrange identical data into groups.

The GROUP BY statement comes after any WHERE statements, but before ORDER BY or LIMIT.

Sometimes, we want to GROUP BY a calculation done on a column.

For instance, we might want to know how many movies have IMDb ratings that round to 1, 2, 3, 4, 5. We could do this using the following syntax:

```
SELECT ROUND(imdb_rating),
   COUNT(name)
FROM movies
GROUP BY ROUND(imdb_rating)
ORDER BY ROUND(imdb_rating);
```

However, this query may be time-consuming to write and more prone to error.

SQL lets us use column reference(s) in our GROUP BY that will make our lives easier.

- 1 is the first column selected
- 2 is the second column selected
- 3 is the third column selected

and so on.

The following query is equivalent to the one above:

```
SELECT ROUND(imdb_rating),
   COUNT(name)
FROM movies
GROUP BY 1
ORDER BY 1;
```

Here, the 1 refers to the first column in our SELECT statement, ROUND(imdb_rating).

3.8 Having

In addition to being able to group data using GROUP BY, SQL also allows you to filter which groups to include and which to exclude.

For instance, imagine that we want to see how many movies of different genres were produced each year, but we only care about years and genres with at least 10 movies.

We cannot use WHERE here because we do not want to filter the rows; we want to filter groups.

This is where HAVING comes in.

HAVING is very similar to WHERE. In fact, all types of WHERE clauses you learned about thus far can be used with HAVING.

We can use the following for the problem:

```sql
SELECT year ,
   genre ,
   COUNT(name)
FROM movies
GROUP BY 1, 2
HAVING COUNT(name) > 10;
```

- When we want to limit the results of a query based on values of the individual rows, use WHERE.
- When we want to limit the results of a query based on an aggregate property, use HAVING.

HAVING statement always comes after GROUP BY, but before ORDER BY and LIMIT.

3.9 Review

We just learned how to use aggregate functions to perform calculations on our data.

- COUNT(): count the number of rows
- SUM(): the sum of the values in a column
- MAX()/MIN(): the largest/smallest value
- AVG(): the average of the values in a column
- ROUND(): round the values in the column

Aggregate functions combine multiple rows together to form a single value of more meaningful information.

- GROUP BY is a clause used with aggregate functions to combine data from one or more columns.
- HAVING limit the results of a query based on an aggregate property.

4 Multiple Tables

Now we learn how to query multiple tables using joins.

4.1 Introduction

In order to efficiently store data, we often spread related information across multiple tables.

For instance, imagine that we are running a magazine company where users can have different types of subscriptions to different products. Different subscriptions might have many different properties. Each customer would also have lots of associated information.

We could have one table with all of the following information:

- `order_id`
- `customer_id`
- `customer_name`
- `customer_address`
- `subscription_id`
- `subscription_description`
- `subscription_monthly_price`
- `subscription_length`
- `purchase_date`

However, a lot of this information would be repeated. If the same customer has multiple subscriptions, that customer's name and address will be reported multiple times. If the same subscription type is ordered by multiple customers, then the subscription price and subscription description will be repeated. This will make our table big and unmanageable.

So instead, we can split our data into three tables:

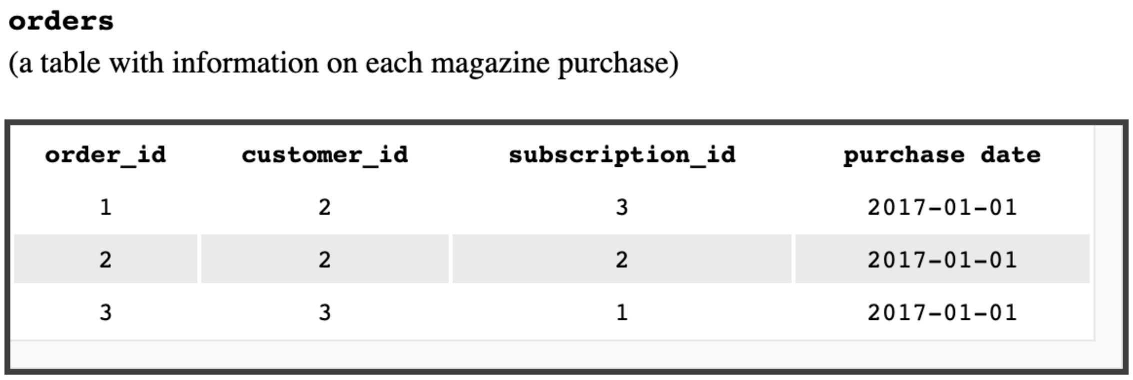
1. `orders` would contain just the information necessary to describe what was ordered:
   - `order_id`, `customer_id`, `subscription_id`, `purchase_date`
2. subscriptions would contain the information to describe each type of subscription:
   - `subscription_id`, `description`, `price_per_month`, `subscription_length`
3. customers would contain the information for each customer:
   - `customer_id`, `customer_name`, `address`

4.2 Combining Tables Manually

Let us return to our magazine company. Suppose we have the three tables described previously and are shown below:

- orders

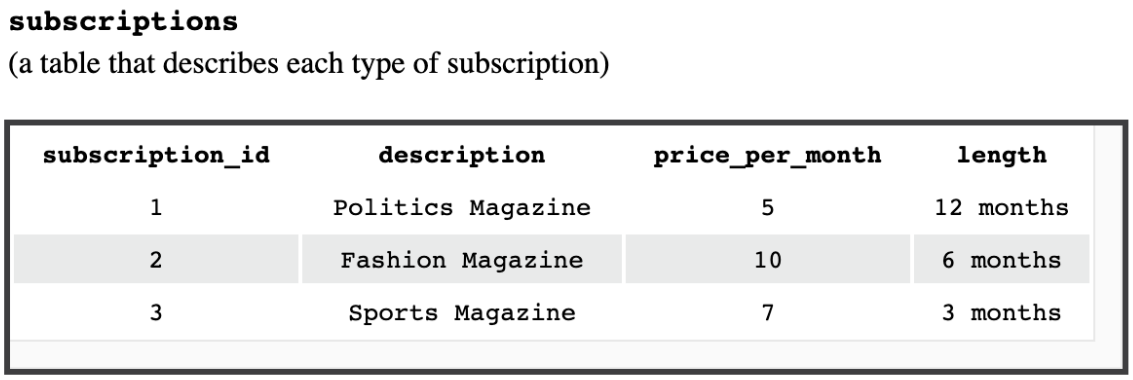
- subscriptions
- customers



orders

(a table with information on each magazine purchase)

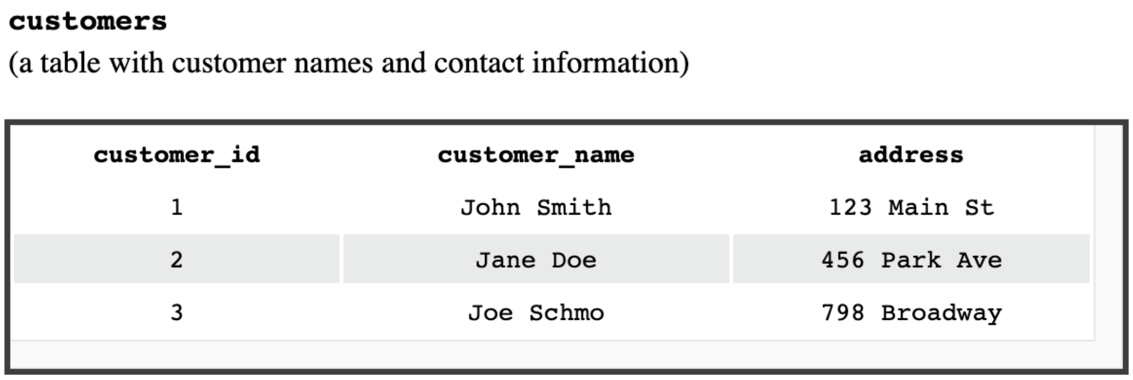
order_id	customer_id	subscription_id	purchase date
1	2	3	2017-01-01
2	2	2	2017-01-01
3	3	1	2017-01-01



subscriptions

(a table that describes each type of subscription)

subscription_id	description	price_per_month	length
1	Politics Magazine	5	12 months
2	Fashion Magazine	10	6 months
3	Sports Magazine	7	3 months



customers

(a table with customer names and contact information)

customer_id	customer_name	address
1	John Smith	123 Main St
2	Jane Doe	456 Park Ave
3	Joe Schmo	798 Broadway

If we just look at the orders table, we cannot really tell what has happened in each order. However, if we refer to the other tables, we can get a complete picture.

Let us examine the order with an order_id of 2. It was purchased by the customer with a customer_id of 2.

To find out the customer's name, we look at the customers table and look for the item with a customer_id value of 2. We can see that Customer 2's name is 'Jane Doe' and that she lives at '456 Park Ave'.

Doing this kind of matching is called **joining** two tables.

4.3 Combining Tables with SQL

Combining tables manually is time-consuming. Luckily, SQL gives us an easy sequence for this: it is called a `JOIN`.

If we want to combine orders and customers, we would type:

```sql
SELECT *
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id;
```

Let us break down this command:

1. The first line selects all columns from our combined table. If we only want to select certain columns, we can specify which ones we want.
2. The second line specifies the first table that we want to look in, `orders`
3. The third line uses `JOIN` to say that we want to combine information from `orders` with `customers`.
4. The fourth line tells us how to combine the two tables. We want to match `orders` table's `customer_id` column with `customers` table's `customer_id` column.

Because column names are often repeated across multiple tables, we use the syntax `table_name.column_name` to be sure that our requests for columns are unambiguous. In our example, we use this syntax in the `ON` statement, but we will also use it in the `SELECT` or any other statement where we refer to column names.

For example: Instead of selecting all the columns using `*`, if we only wanted to select `orders` table's `order_id` column and `customers` table's `customer_name` column, we could use the following query:

```sql
SELECT orders.order_id,
   customers.customer_name
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id;
```

4.4 Inner Joins

Let us revisit how we joined `orders` and `customers`. For every possible value of `customer_id` in `orders`, there was a corresponding row of customers with the same `customer_id`.

But there are cases that it is not true. For instance, imagine that our `customers` table was out of date, and was missing any information on customer 11. If that customer had an `order` in orders, then when we perform a simple `JOIN` (often called an *inner join*) our result only includes rows that match our `ON` condition.

4.5 Left Joins

If we want to combine two tables and keep some of the un-matched rows, we can do this through a command called `LEFT JOIN` (or `RIGHT JOIN`). A *left join* will keep all rows from the first table, regardless of whether there is a matching row in the second table. The final result will keep all rows of the first table but will omit the un-matched row from the second table.

Consider the following code

```sql
SELECT *
FROM table1
LEFT JOIN table2
  ON table1.c2 = table2.c2;
```

- The first line selects all columns from both tables.
- The second line selects table1 (the "left" table).
- The third line performs a LEFT JOIN on table2 (the "right" table).
- The fourth line tells SQL how to perform the join (by looking for matching values in column c2).

4.6 Primary Key vs Foreign Key

Let us return to our example of the magazine subscriptions. Recall that we had three tables: orders, subscriptions, and customers.

Each of these tables has a column that uniquely identifies each row of that table:

- order_id for orders
- subscription_id for subscriptions
- customer_id for customers

These special columns are called **primary keys**.

Primary keys have a few requirements:

- None of the values can be NULL.
- Each value must be unique (i.e., you cannot have two customers with the same customer_id in the customers table).
- A table can not have more than one primary key column.

Let us reexamine the orders table:

order_id	**customer_id**	**subscription_id**	**purchase_date**
1	2	3	2017-01-01
2	2	2	2017-01-01
3	3	1	2017-01-01

Note that customer_id (the primary key for customers) and subscription_id (the primary key for subscriptions) both appear in this.

When the primary key for one table appears in a different table, it is called a **foreign key**.

So customer_id is a primary key when it appears in customers, but a foreign key when it appears in orders.

In this example, our primary keys all had somewhat descriptive names. Generally, the primary key will just be called id. Foreign keys will have more descriptive names.

This is important because the most common types of joins will be joining a foreign key from one table with the primary key from another table. For instance, when we join orders and customers, we join on customer_id, which is a foreign key in orders and the primary key in customers.

4.7 Cross Join

So far, we have focused on matching rows that have some information in common.

Sometimes, we just want to combine all rows of one table with all rows of another table. For instance, if we had a table of shirts and a table of pants, we might want to know all the possible combinations to create different outfits. Our code might look like this:

```sql
SELECT shirts.shirt_color,
   pants.pants_color
FROM shirts
CROSS JOIN pants;
```

- The first two lines select the columns shirt_color and pants_color.
- The third line pulls data from the table shirts.
- The fourth line performs a CROSS JOIN with pants.

Notice that cross joins do not require an ON statement because we are not joining on any columns.

If we have 3 different shirts (white, grey, and olive) and 2 different pants (light denim and black), the results might look like this:

shirt_color	**pants_color**
white	light denim
white	black
grey	light denim
grey	black
olive	light denim
olive	black

3 shirts $\times$ 2 pants = 6 combinations

A more common usage of CROSS JOIN is when we need to compare each row of a table to a list of values.

Let us return to our newspaper subscriptions. This table contains two columns that we have not discussed yet:

- start_month: the first month where the customer subscribed to the print newspaper (i.e., 2 for February)
- end_month: the final month where the customer subscribed to the print newspaper

Suppose we wanted to know how many users were subscribed during each month of the year. For each month (1, 2, 3) we would need to know if a user was subscribed.

```sql
SELECT month, COUNT(*)
FROM newspaper
CROSS JOIN months
WHERE start_month <= month AND end_month >= month
GROUP BY month;
```

4.8 Union

Sometimes we just want to stack one dataset on top of the other. The UNION operator allows us to do that.

Suppose we have two tables and they have the same columns.

table1:

pokemon	type
Bulbasaur	Grass
Charmander	Fire
Squirtle	Water

table2:

pokemon	type
Snorlax	Normal

If we combine these two with UNION:

```sql
SELECT *
FROM table1
UNION
SELECT *
FROM table2;
```

The result would be:

pokemon	type
Bulbasaur	Grass
Charmander	Fire
Squirtle	Water
Snorlax	Normal

SQL has strict rules for appending data:

- Tables must have the same number of columns.
- The columns must have the same data types in the same order as the first table.

4.9 With

Often times, we want to combine two tables, but one of the tables is the result of another calculation.

Let us return to our magazine order example. Our marketing department might want to know a bit more about our customers. For instance, they might want to know how many magazines each customer subscribes to. We can easily calculate this using our orders table:

```sql
SELECT customer_id,
   COUNT(subscription_id) AS "subscriptions"
FROM orders
GROUP BY customer_id;
```

This query is good, but a customer_id is not terribly useful for our marketing department, they probably want to know the customer's name.

We want to be able to join the results of this query with our customers table, which will tell us the name of each customer. We can do this by using a WITH clause.

```sql
WITH previous_results AS (
   SELECT ...
   ...
   ...
   ...
)
SELECT *
FROM previous_results
JOIN customers
  ON _____ = _____;
```

- The WITH statement allows us to perform a separate query (such as aggregating customer's subscriptions)
- previous_results is the alias that we will use to reference any columns from the query inside of the WITH clause
- We can then go on to do whatever we want with this temporary table (such as join the temporary table with another table)

Essentially, we are putting a whole first query inside the parentheses () and giving it a name. After that, we can use this name as if it is a table and write a new query using the first query.

4.10 Review

We learned about relationships between tables in relational databases and how to query information from multiple tables using SQL.

- JOIN will combine rows from different tables if the join condition is true.
- LEFT JOIN will return every row in the left table, and if the join condition is not met, NULL values are used to fill in the columns from the right table.
- Primary key is a column that serves a unique identifier for the rows in the table.
- Foreign key is a column that contains the primary key to another table.
- CROSS JOIN lets us combine all rows of one table with all rows of another table.
- UNION stacks one dataset on top of another.
- WITH allows us to define one or more temporary tables that can be used in the final query.

5 Analytic Functions

5.1 Introduction

Now we will learn how to define *analytic functions*, which also operate on a set of rows like *aggregate functions*. However, unlike aggregate functions, analytic functions return a (potentially different) value for each row in the original table.

Analytic functions allow us to perform complex calculations with relatively straightforward syntax. For instance, we can quickly calculate moving averages and running totals, among other quantities.

5.2 Syntax

To understand how to write analytic functions, we will work with a small table containing data from two different people who are training for a race. The id column identifies each runner, the date column holds the day of the training session, and time shows the time (in minutes) that the runner dedicated to training.

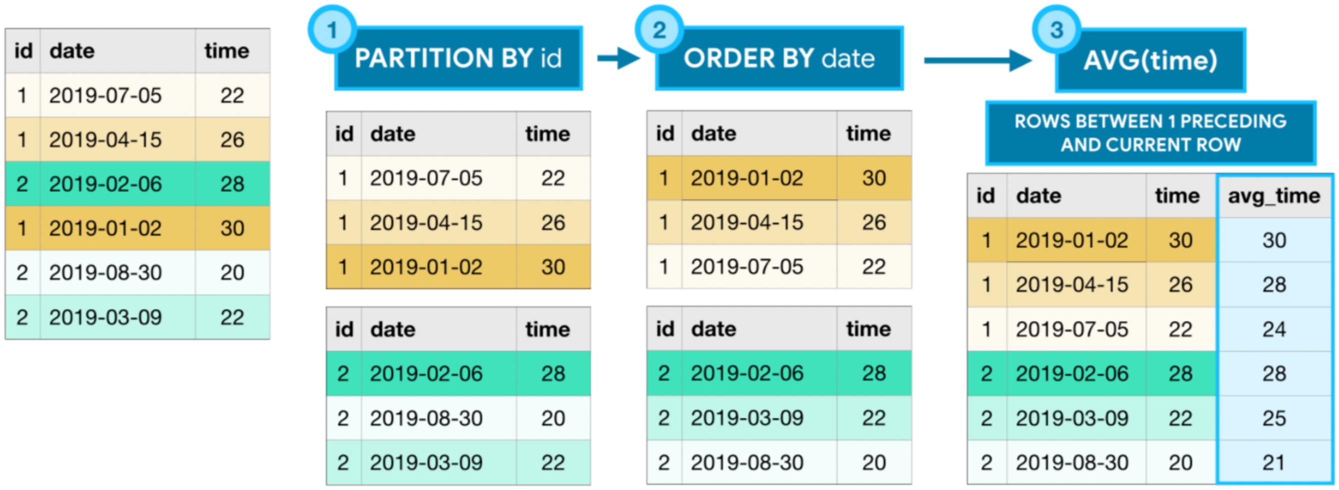
id	**date**	**time**
1	2019-07-05	22
1	2019-04-15	26
2	2019-02-06	28
1	2019-01-02	30
1	2019-08-30	30
1	2019-03-09	22

Say we would like to calculate a *moving average* of the training times for each runner, where we always take the average of the current and previous training sessions. We can do this with the following query:

```sql
SELECT *,
  AVG(time) OVER(
           PARTITION BY id
           ORDER BY date
           ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
           ) as avg_time
FROM table;
```

All analytic functions have an OVER clause, which defines the sets of rows used in each calculation. The OVER clause has three (optional) parts:

- The PARTITION BY clause divides the rows of the table into different groups. In the query above, we divide by id so that the calculations are separated by runner.
- The ORDER BY clause defines an ordering within each partition. In the sample query, ordering by the date column ensures that earlier training sessions appear first.
- The final clause (ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) is known as a **window frame** clause. It identifies the set of rows used in each calculation. We can refer to this group of rows as a **window**. (Actually, analytic functions are sometimes referred to as **analytic window functions** or **window functions**.)



id	date	time
1	2019-07-05	22
1	2019-04-15	26
2	2019-02-06	28
1	2019-01-02	30
2	2019-08-30	20
2	2019-03-09	22

id	date	time
1	2019-07-05	22
1	2019-04-15	26
1	2019-01-02	30

id	date	time
2	2019-02-06	28
2	2019-08-30	20
2	2019-03-09	22

id	date	time
1	2019-01-02	30
1	2019-04-15	26
1	2019-07-05	22

id	date	time
2	2019-02-06	28
2	2019-03-09	22
2	2019-08-30	20

id	date	time	avg_time
1	2019-01-02	30	30
1	2019-04-15	26	28
1	2019-07-05	22	24
2	2019-02-06	28	28
2	2019-03-09	22	25
2	2019-08-30	20	21

5.3 (More on) window frame clauses

There are many ways to write window frame clauses:

- `ROWS BETWEEN 1 PRECEDING AND CURRENT ROW` - the previous row and the current row.
- `ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING` - the 3 previous rows, the current row, and the following row.
- `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` - all rows in the partition.

Of course, this is not an exhaustive list, and you can imagine that there are many more options. In the code below, you will see some of these clauses in action.

5.4 Three types of analytic functions

The example above uses only one of many analytic functions. BigQuery supports a wide variety of analytic functions, and we will explore a few here. For a complete listing, you can take a look at the documentation.

1. Analytic aggregate functions As you might recall, `AVG()` (from the example above) is an aggregate function. The `OVER` clause is what ensures that it is treated as an analytic (aggregate) function. **Aggregate functions** take all of the values within the window as input and return a single value.
   - `MIN()` (or `MAX()`) - Returns the minimum (or maximum) of input values
   - `AVG()` (or `SUM()`) - Returns the average (or sum) of input values
   - `COUNT()` - Returns the number of rows in the input
2. Analytic navigation functions **Navigation functions** assign a value based on the value in a (usually) different row than the current row.
   - `FIRST_VALUE()` (or `LAST_VALUE()`) - Returns the first (or last) value in the input
   - `LEAD()` (and `LAG()`) - Returns the value on a subsequent (or preceding) row

3. Analytic numbering functions **Numbering functions** assign integer values to each row based on the ordering.
    - ROW_NUMBER() - Returns the order in which rows appear in the input (starting with 1)
    - RANK() - All rows with the same value in the ordering column receive the same rank value, where the next row receives a rank value which increments by the number of rows with the previous rank value.

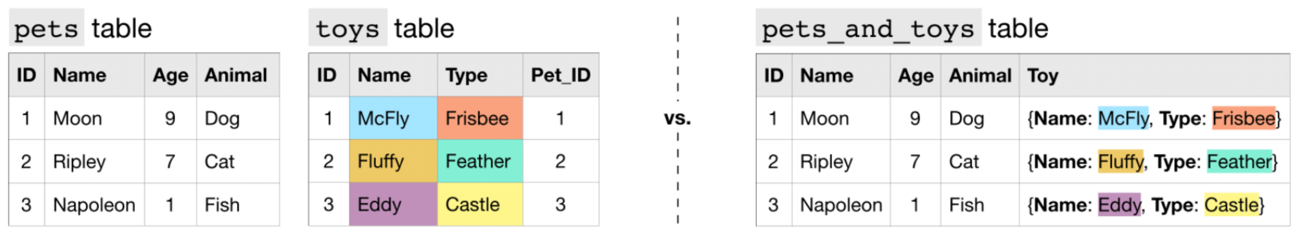
6 Nested and Repeated Data

Now we will learn how to query nested and repeated data. These are the most complex data types that you can find in BigQuery datasets.

6.1 Nested Data

Consider a hypothetical dataset containing information about pets and their toys. We could organize this information in two different tables (a `pets` table and a `toys` table). The `toys` table could contain a `Pet_ID` column that could be used to match each toy to the pet that owns it.

Another option in BigQuery is to organize all of the information in a single table, similar to the `pets_and_toys` table below.



`pets` table

ID	Name	Age	Animal
1	Moon	9	Dog
2	Ripley	7	Cat
3	Napoleon	1	Fish

`toys` table

ID	Name	Type	Pet_ID
1	McFly	Frisbee	1
2	Fluffy	Feather	2
3	Eddy	Castle	3

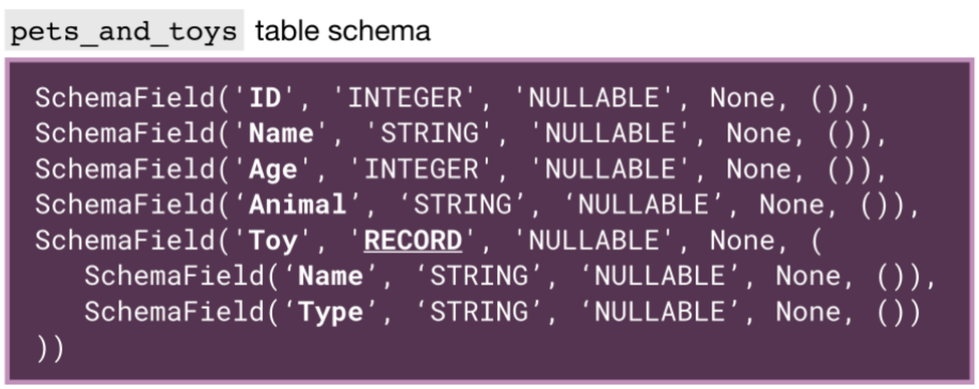
vs.

`pets_and_toys` table

ID	Name	Age	Animal	Toy
1	Moon	9	Dog	{**Name**: McFly, **Type**: Frisbee}
2	Ripley	7	Cat	{**Name**: Fluffy, **Type**: Feather}
3	Napoleon	1	Fish	{**Name**: Eddy, **Type**: Castle}

In this case, all of the information from the `toys` table is collapsed into a single column (the `Toy` column in the `pets_and_toys` table). We refer to the `Toy` column in the `pets_and_toys` table as a **nested column**, and say that the `Name` and `Type` fields are nested inside of it.

Nested columns have type **STRUCT** (or type **RECORD**). This is reflected in the table schema below.



`pets_and_toys` table schema

```
SchemaField('ID', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
SchemaField('Age', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Animal', 'STRING', 'NULLABLE', None, ()),
SchemaField('Toy', 'RECORD', 'NULLABLE', None, (
   SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
   SchemaField('Type', 'STRING', 'NULLABLE', None, ())
))
```

To query a column with nested data, we need to identify each field in the context of the column that contains it:

- `Toy.Name` refers to the `Name` field in the `Toy` column
- `Toy.Type` refers to the `Type` field in the `Toy` column.

```sql
SELECT Name AS Pet_Name,
        Toy.Name AS Toy_Name,
        Toy.Type AS Toy_Type
FROM pets_and_toys;
```

The code above yields:

Pet_Name	Toy_Name	Toy_Type
Moon	McFly	Frisbee
Ripley	Fluffy	Feather
Napoleon	Eddy	Castle

Otherwise, our usual rules remain the same - we need not change anything else about our queries.

6.2 Repeated Data

Now consider the case where each pet can have multiple toys. In this case, to collapse this information into a single table, we need to leverage a different datatype.

pets table

ID	Name	Age	Animal
1	Moon	9	Dog
2	Ripley	7	Cat
3	Napoleon	1	Fish

toys_type table

ID	Type	Pet_ID
1	Frisbee	1
2	Bone	1
3	Rope	1
4	Feather	2
5	Ball	2
6	Castle	3

vs.

pets_and_toys_type table

ID	Name	Age	Animal	Toys
1	Moon	9	Dog	[Frisbee, Bone, Rope]
2	Ripley	7	Cat	[Feather, Ball]
3	Napoleon	1	Fish	[Castle]

We say that the Toys column contains **repeated data**, because it permits more than one value for each row. This is reflected in the table schema below, where the mode of the Toys column appears as REPEATED.

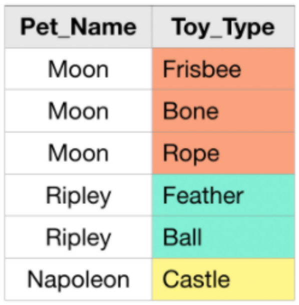
pets_and_toys_type table schema

```
SchemaField('ID', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
SchemaField('Age', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Animal', 'STRING', 'NULLABLE', None, ()),
SchemaField('Toys', 'STRING', 'REPEATED', None, ())
```

Each entry in a repeated field is an ARRAY, or an ordered list of (zero or more) values with the same datatype. For instance, the entry in the Toys column for Moon the Dog is **[Frisbee, Bone, Rope]**, which is an ARRAY with three values.

When querying repeated data, we need to put the name of the column containing the repeated data inside an UNNEST() function.

```sql
SELECT Name AS PetName,
          Toy_Type
FROM pets_and_toys_type
  UNNEST(Toys) AS Toy_Type;
```

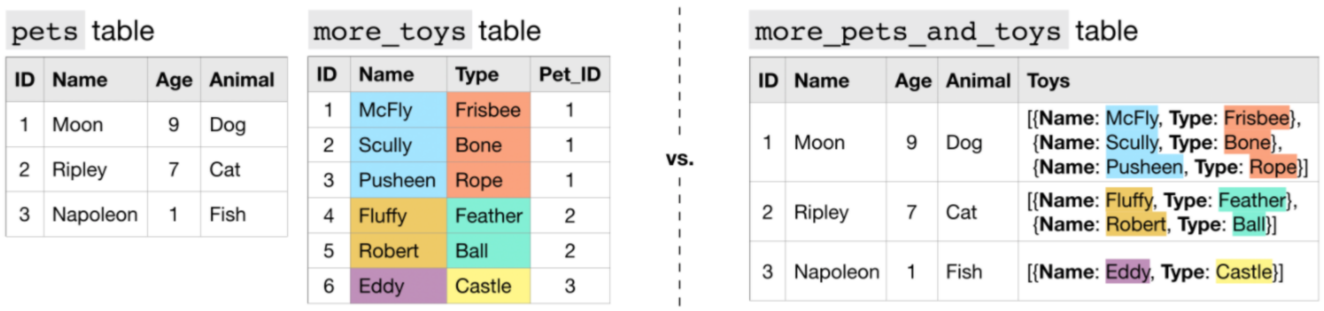


Pet_Name	Toy_Type
Moon	Frisbee
Moon	Bone
Moon	Rope
Ripley	Feather
Ripley	Ball
Napoleon	Castle

This essentially flattens the repeated data (which is then appended to the right side of the table) so that we have one element on each row.

6.3 Nested and repeated data

If pets can have multiple toys, and we would like to keep track of both the name and type of each toy, we can make the Toys column both **nested** and **repeated**.



pets table

ID	Name	Age	Animal
1	Moon	9	Dog
2	Ripley	7	Cat
3	Napoleon	1	Fish

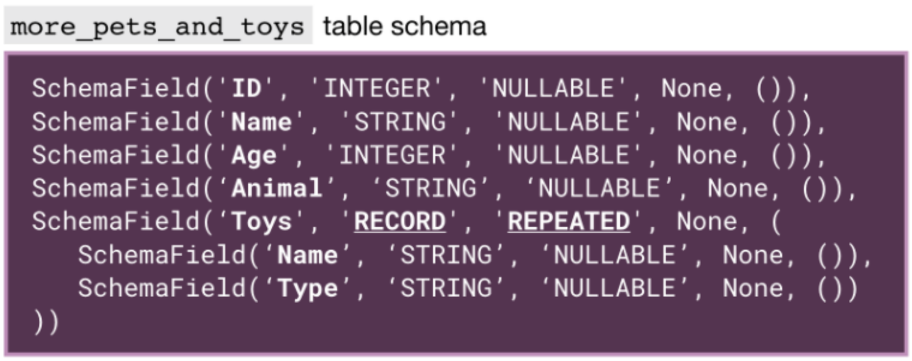
more_toys table

ID	Name	Type	Pet_ID
1	McFly	Frisbee	1
2	Scully	Bone	1
3	Pusheen	Rope	1
4	Fluffy	Feather	2
5	Robert	Ball	2
6	Eddy	Castle	3

more_pets_and_toys table

ID	Name	Age	Animal	Toys
1	Moon	9	Dog	[{**Name**: McFly, **Type**: Frisbee}, {**Name**: Scully, **Type**: Bone}, {**Name**: Pusheen, **Type**: Rope}]
2	Ripley	7	Cat	[{**Name**: Fluffy, **Type**: Feather}, {**Name**: Robert, **Type**: Ball}]
3	Napoleon	1	Fish	[{**Name**: Eddy, **Type**: Castle}]

In the more_pets_and_toys table above, Name and Type are both fields contained within the Toys STRUCT, and each entry in both Toys.Name and Toys.Type is an ARRAY.



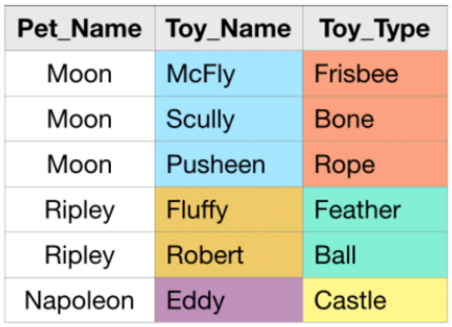
more_pets_and_toys table schema

```
SchemaField('ID', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
SchemaField('Age', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Animal', 'STRING', 'NULLABLE', None, ()),
SchemaField('Toys', 'RECORD', 'REPEATED', None, (
    SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
    SchemaField('Type', 'STRING', 'NULLABLE', None, ())
))
```

Let us look at a sample query.

```
SELECT Name AS Pet_Name,
         t.Name AS Toy_Name,
         t.Type AS Toy_Type
FROM more_pets_and_toys
  UNNEST(Toys) AS t;
```

The code above generates:



Pet_Name	Toy_Name	Toy_Type
Moon	McFly	Frisbee
Moon	Scully	Bone
Moon	Pusheen	Rope
Ripley	Fluffy	Feather
Ripley	Robert	Ball
Napoleon	Eddy	Castle

Since the Toys column is repeated, we flatten it with the UNNEST() function. And, since we give the flattened column an alias of t, we can refer to the Name and Type fields in the Toys column as t.Name and t.Type, respectively.