Sololearn SQL intermediate

Lun-Hsien Chang

2024-12-26

SQL Intermediate

Data Manipulations **1**
- Introduction . . . 1
- String Functions . . . 2
- Math & Aggregate Functions . . . 3
- CASE . . . 3
- Module 1 Quiz . . . 4

Table Constraints **4**
- Identity . . . 4
- PRACTICE EXERCISE Adding Data . . . 4
- Primary & Foreign Keys . . . 5
- Unique . . . 6
- Module 2 Quiz . . . 7

Working with Data **7**
- Multiple Tables . . . 7
- Joins . . . 8
- Union . . . 10
- Find the Average . . . 11
- Module 3 Quiz . . . 12

Certificate **13**

Data Manipulations

Introduction

SQL databases are used everywhere, from e-commerce websites to social networks and games. In this course we will learn how to perform calculations, store and manipulate data in multiple tables, create relations between tables and write advanced SQL queries to analyze data.

To recap:

- A **SELECT** statement is used to select data from a table.
- You can use the **ORDER BY** clause to order the results of the query by a specified column or columns.
- The **LIMIT** keyword allows you to select only a subset of the data, by specifying the start index and the number of rows with **OFFSET**.

String Functions

SQL provides a number of useful functions to work with text data (or strings, as they are called in programming).

For example, the CONCAT function allows you to combine text from multiple columns into one. Let's select the full name of the customers as a new column called name:

```
SELECT CONCAT(firstname, lastname) AS name
FROM Customers
```

The **CONCAT** function can take any number of arguments, combining them together in the result. So we can simply add a space between the columns to get the result we want:

```
SELECT CONCAT(firstname, ' ', lastname) AS name
FROM Customers
```

The **LOWER** function converts the text in the provided column to lowercase.

The **SUBSTRING** function allows you to extract part of the text in a column. It takes the starting position and the number of characters we want to extract
`SUBSTRING(string, start_position, length)`.
For example, let's take the first 3 characters of the **firstname:**

```
/*Extract 3 characters, starting from the first character of firstname*/
SELECT SUBSTRING(firstname, 1, 3)
FROM Customers

/*Extract the last 3 characters

LENGTH(firstname) returns the total number of characters in the string firstname
(LENGTH(firstname) - 2) moves the starting position 2 characters before the end of the string,
FOR 3 means that starting from the position calculated by LENGTH(firstname) - 2, 3 characters
SELECT SUBSTRING(firstname FROM LENGTH(firstname)-2 FOR 3)
FROM Customers
```

The **REPLACE** function replaces all occurrences of the given string with another one. The syntax form: `REPLACE(string, old_substring, new_substring)` For example, let's replace **New York** with **NY** in the city column:

```
SELECT firstname, lastname, REPLACE(city, 'New York', 'NY')
FROM Customers
```

We can combine multiple functions into a single query. For example, let's create a name column that has the first letter of the **firstname**, followed by a dot and the **lastname** in all uppercase:

```
SELECT CONCAT(SUBSTRING(firstname, 1, 1), '. ', UPPER(lastname)) AS name
FROM Customers
```

Math & Aggregate Functions

We can use mathematical operators on multiple columns to get a result. Let's consider that our table includes **weight** and **height** columns for our customers. We can calculate the **BMI** based on these values:

```
SELECT firstname, lastname, weight/(height*height) AS bmi
FROM Customers
```

The BMI is equal to the weight divided by the square of the height (the values are in metric units). Remember, these new columns exist only in the result table of the query, and not in the original table.

CASE

We want to set the category columns value to 'Senior', in case the age value is greater than 65, 'Adult' in case it's in the range of 25 to 64, and 'Youth', if it's under 25. This is done using the CASE statement. Here is one condition:

```
SELECT firstname, lastname, CASE WHEN age >= 65 THEN 'Senior' END AS category
FROM Customers
```

We can add multiple conditions using multiple **WHEN** clauses. Here is the second condition:

```
SELECT firstname
       ,lastname
       , CASE WHEN age >= 65 THEN 'Senior'
              WHEN age >= 25 AND age < 65 THEN 'Adult' END AS category
FROM Customers
```

For all other cases, we can set a value using the **ELSE** keyword:

```
SELECT firstname
      ,lastname
      ,CASE WHEN age >= 65 THEN 'Senior'
            WHEN age >= 25 AND age < 65 THEN 'Adult'
            ELSE 'Youth'
            END AS category
FROM Customers
```

Module 1 Quiz

Table Constraints

Identity

As we have seen, most tables we work with have a column called id, which contains a number. That column is often called the identity column and is similar to a row number in Excel. Here is the id column for our customers:

```
SELECT id, firstname FROM Customers
```

Often, the id is an integer (a whole number) which is incremented with each new row. SQL allows you to create a column that gets automatically incremented with each new row. That is done using the **AUTO_INCREMENT** keyword.

Create a table called 'Products' with an auto increment column called 'number':

```
CREATE TABLE Products (number int AUTO_INCREMENT);
```

Now when inserting a new row, we do not need to specify the value of the id column, as it will automatically be set. For example:

```
INSERT INTO Customers (firstname, lastname, city, age)
VALUES ('demo', 'demo', 'Paris', 52), ('test', 'test', 'London', 21); SELECT * FROM Customers;
```

By default, the **AUTO_INCREMENT** column starts with the value **1**. This can be changed if needed, using the following:

```
ALTER TABLE Customers AUTO_INCREMENT=555
```

```
INSERT INTO Customers (firstname, lastname, city, age)
VALUES ('test', 'test', 'London', 21);
SELECT * FROM Customers;
```

PRACTICE EXERCISE Adding Data

There are new employees that need to be added to the Employees table. Here is their data:

Firstname: Wang

Lastname: Lee

Salary: 1900

Firstname: Greta

Lastname: Wu

Salary: 1200

The Employees table has an identity column called id, which is set to AUTO_INCREMENT.

Add the data to the table, then select the id, firstname, lastname and salary columns sorted by the id column in descending order.

```
INSERT INTO Employees (firstname, lastname, salary)
VALUES ('Wang','Lee', 1900), ('Greta', 'Wu', 1200);

SELECT id, firstname, lastname, salary
FROM Employees
ORDER BY id DESC
```

Primary & Foreign Keys

Another important concept in databases are keys. They are used to define relationships between tables. Let's say we need to store multiple phone numbers for each of our Customers. Storing it in the same table would be problematic, as each customer can have a variable number of phone numbers.

Before looking at how the data will look in these tables, let's first create the relationship between them using keys! The **primary key** constraint is used to uniquely identify rows of a table. In most cases, the primary key is the **auto_increment** column. So, for our **Customers** and **PhoneNumbers** tables, it's the **id** column. It is set when creating the table:

```
CREATE TABLE Customers (
id int NOT NULL AUTO_INCREMENT
,firstname varchar(255)
,lastname varchar(255)
,PRIMARY KEY (id)
);
```

Here are some rules for primary keys:

- A primary key must contain **unique** values.
- A primary key column cannot have **NULL** values.
- A table can have **only one** primary key.

Another type of constraint is the **Foreign Key.** A **Foreign Key** is a column in one table that refers to the **Primary Key** in another table.

In our case, the **customer_id** column in the PhoneNumbers table is the foreign key, which refers to the primary key **id** in the Customers table.

```
CREATE TABLE PhoneNumbers (
id int NOT NULL AUTO_INCREMENT
,customer_id int NOT NULL
,number varchar(55)
,type varchar(55)
,PRIMARY KEY (id)
,FOREIGN KEY (customer_id) REFERENCES Customers(id));
```

Here is how some example data in the Customers and PhoneNumbers table would look:

Customers:

id	firstname	lastname	city	age
1	John	Smith	New York	24
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65

PhoneNumbers:

id	customer_id	number	type
1	1	(555) 123456	mobile
2	1	(943)554545	home
3	2	(331) 111111	mobile
4	2	(88) 11 22 33	work
5	2	(999) 00 11 33	emergency

This is how relationships between tables are created. The foreign key column is referencing the primary key column of another table, thus linking the data in these tables.

This way, a customer can have any number of phone numbers associated with them.

Lesson Takeaways

Awesome! Now you know how to create keys in tables, linking the data.

- The primary key is used to uniquely identify each row of a table. It is usually the identity column.
- The foreign key is used to reference an identity column in another table. This allows you to link the data between multiple tables and prevent actions that would break the relationship.

Unique

The **UNIQUE** constraint ensures that all values in a column are different. A **PRIMARY KEY** constraint automatically has a **UNIQUE** constraint.

Let's make the **lastname** column of our Customers unique:

```
ALTER TABLE Customers
ADD UNIQUE (lastname)
```

Now when we try to insert a Customer with a lastname that is already present in the table, we will get an error:

```
INSERT INTO Customers (firstname, lastname, city, age)
VALUES ('demo', 'Anderson', 'London', 24);
SELECT * FROM Customers;
```

'ERROR: duplicate key value violates unique constraint "customers_lastname_key" DETAIL: Key (lastname)=(Anderson) already exists.'

True or False: You can have multiple NULL values in a unique column. True

Summary

Let's summarze what we have learned about keys:

- The Primary key uniquely identifies each record of a table. It is usually set as an auto increment integer.
- Foreign keys are used to create relationships between tables. They refer to the primary key in other tables.
- A table can have multiple foreign keys, but only one single primary key.
- The UNIQUE constraint is used to make values in a column unique.

Module 2 Quiz

Create a table with an auto incremented column called id, which is the primary key.

```
CREATE TABLE Users (
id int AUTO_INCREMENT)
,name varchar(255) NOT NULL
PRIMARY KEY (id)
);
```

Working with Data

Multiple Tables

True or False: The foreign key column should be unique. False

We can select data from multiple tables by comma separating them in a **SELECT** statement:

```
SELECT firstname, lastname, city, number, type
FROM Customers, PhoneNumbers
WHERE Customers.id = PhoneNumbers.customer_id
```

When working with multiple tables, it's common practice to define the columns by their full name – the table name, followed by a dot and the column name. For example: Customers.id is the id column of the Customers table, while PhoneNumbers.id is the id column of the PhoneNumbers table.

So, here is what our query would look like with the full column names:

```
SELECT Customers.firstname, Customers.lastname, Customers.city, PhoneNumbers.number, PhoneNumbe
FROM Customers, PhoneNumbers
WHERE Customers.id = PhoneNumbers.customer_id
```

PRACTICE EXERCISE Books and Authors

You are working with a library database that stores data on books. The **Books** table has the columns **id, name, year, author_id.** The **author_id** column connects to the **Authors** table, which stores the **id, name** columns for the book authors. You need to select all the books with their authors, ordered by the author name alphabetically, then by the year in ascending order. The result set should contain only 3 columns: the book **name, year** and its **author** (name the column **author).**

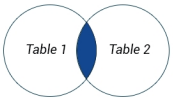
```
SELECT Books.name, Books.year, Authors.name as author
FROM Books, Authors
WHERE Books.author_id= Authors.id
ORDER BY author, Books.year
```

Joins

A better way of combining data is the JOIN clause. It allows you to combine multiple tables based on a condition. For example,

```
SELECT firstname, lastname, city, number, type
FROM Customers JOIN PhoneNumbers
ON Customers.id = PhoneNumbers.customer_id
```

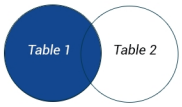
The image below demonstrates how **JOIN** works:



Because you use the full column names when joining tables, the query can get really long. To make it easier and shorter, we can provide nicknames for our tables:

```
SELECT C.firstname, C.lastname, C.city, PN.number, PN.type
FROM Customers AS C JOIN PhoneNumbers AS PN
ON C.id = PN.customer_id
```

Another type of JOIN is the LEFT JOIN. The LEFT JOIN returns all rows from the left table (first table), even if there are no matches in the right table (second table). This means that if there are no matches for the ON clause in the table on the right, the join will still return the rows from the first table in the result. The image below demonstrates how LEFT JOIN works:



For example, in our case, the **Customers** table includes customers that do not have any records in the **PhoneNumbers** table:

```
SELECT C.firstname, C.lastname, C.city, PN.number, PN.type
FROM Customers AS C LEFT JOIN PhoneNumbers AS PN
ON C.id = PN.customer_id
```

The table A contains 3 rows in the id column with the values 1, 2, 3. The B table has an id column containing 3 rows with the values 1, 2, 1. How many rows will the following query return?

```
SELECT A.id, B.id
FROM A LEFT JOIN B
ON A.id = B.id
```

4

Similarly, the **RIGHT JOIN** returns all the rows from the right table, even if there are no matches in the left table. For example, we could rewrite the previous query this way:

```
SELECT C.firstname, C.lastname, C.city, PN.number, PN.type
FROM PhoneNumbers AS PN RIGHT JOIN Customers AS C
ON C.id = PN.customer_id
ORDER BY C.id
```

Lesson Takeaways

Joins allow you to combine data from multiple tables based on conditions.

The LEFT JOIN returns all rows from the left table (first table), even if there are no matches in the right table (second table).

Similarly, RIGHT JOIN returns all the rows from the right table, even if there are no matches in the left table.

You will learn how to combine results of SELECT statements into one single data set in the next lesson, so stay tuned!

PRACTICE EXERCISE Number of Books

You are working on the library database, which contains the Books and Authors tables. Columns of the Books table: **id**, **name**, **year**, **author_id**. Columns of the Authors table: **id**, **name**. Write a query to get the author names and the number of books they have in the Books table.

Note that some authors do not have any books associated with them. In this case, the result needs to include their names and have 0 as the count. The count column should be called books in the result. Sort the result by the number of books, from highest to lowest.

```
SELECT a.name, COUNT(b.name) AS books
FROM Authors AS a LEFT JOIN Books AS b
ON a.id = b.author_id
GROUP BY a.name
ORDER BY books DESC
```

Union

Occasionally, you might need to combine data from multiple similar tables into one comprehensive dataset. For example, you might have multiple tables storing Customers data and you want to combine them into one result set. This can be done using the **UNION** statement.

The UNION operator is used to combine the result-sets of two or more SELECT statements. Consider having a Customers and Contacts tables, both having firstname, lastname and age columns:

```
SELECT firstname, lastname, age FROM Customers
UNION
SELECT firstname, lastname, age FROM Contacts
```

All **SELECT** statements within the **UNION** must have the same number of columns. The columns must also have the same data types. Also, the columns in each **SELECT** statement must be in the same order.

UNION removes the duplicate records.

UNION ALL is similar to **UNION,** but does not remove the duplicates:

```
SELECT firstname, lastname, age FROM Customers
UNION ALL
SELECT firstname, lastname, age FROM Contacts
```

Remember, the **SELECT** statements need to have the same columns for the **UNION** to work. In case one of the tables has extra columns that we need to select, we can simply add them to the second select as **NULL**:

```
SELECT firstname, lastname, age, city FROM Customers
UNION
SELECT firstname, lastname, age, NULL FROM Contacts
```

We can also set conditions for each select in the UNION. For example:

```
SELECT firstname, lastname, age FROM Customers WHERE age > 30
UNION
SELECT firstname, lastname, age FROM Contacts WHERE age < 25
```

Lesson Takeaways

To summarize this lesson:

- UNION allows you to combine records from multiple SELECT statements into one dataset.
- For UNION to work, each SELECT statement needs to have the same number of columns and matching data types.
- UNION removes duplicate records, while UNION ALL does not remove them.
- Each SELECT statement in a UNION can have its own conditions.

Next you will learn how to solve a real-life SQL challenge!

PRACTICE EXERCISE New Arrivals

You are working with the library books database. The Books table has the columns **id**, **name**, **year**. The library has new books whose information is stored in another table called "New", however they do not have a year column.

Write a query to select the books from both tables, Books and New, combining their data. For the year column of the New books use the value 2022.

Also, select only the books that are released after the year 1900.

The result set should contain the name and year columns only, ordered by the name column alphabetically.

```
SELECT name, year FROM Books WHERE year > 1900
UNION
SELECT name, 2022 AS year FROM NewORDER BY name
```

Find the Average

In this lesson we will learn how to solve a slightly more complex problem - we need to find the average number of phone numbers the Customers in our table have.

To calculate the average, we need to find the number of phone numbers that each customer has, then use the **AVG** function over that result set.

First, let's join the tables:

```
SELECT C.id, C.firstname, C.lastname, PN.number, PN.type
FROM Customers AS C LEFT JOIN PhoneNumbers AS PN
ON C.id = PN.customer_id ORDER BY C.id
```

Now we can group the data based on our customers and find the number of phone numbers each of them has:

```
SELECT C.id, COUNT(PN.number) AS count
FROM Customers AS C LEFT JOIN PhoneNumbers AS PN
ON C.id = PN.customer_id
GROUP BY C.id
```

Now, we need to find the average of these values. For that, we need another SELECT query over the data of the join:

```
SELECT AVG(count) FROM(
  SELECT C.id, COUNT(PN.number) AS count
  FROM Customers AS C LEFT JOIN PhoneNumbers AS PN
  ON C.id = PN.customer_id
  GROUP BY C.id
  ) AS Numbers
```

To treat a SELECT statement as a table and give it a name, we need to enclose it in: parentheses

Lesson Takeaways

You learned how to solve a real-life problem!

The key takeaway from this lesson is that you are able to enclose a query into parentheses and give it a name using the AS keyword. This enables us to use the query as a table: select from it, use it in JOINS, run aggregate functions, etc.

Module 3 Quiz

Table A contains 5 rows, while table B contains 3 rows. How many rows will the following query result?
`SELECT * FROM A, B;`

15

Certificate

[Credential URL](https://www.sololearn.com/certificates/CC-XKPIXCFT)