arm TREASURE DATA

Learn SQL by Calculating Customer Lifetime Value

Setup, Counting and Filtering

CONTENTS

SQL (Structured Query Language) is the most widely used query language for retrieving data from a database. Product managers, marketers, and related professionals rely on data for their decision making, but often have to turn to IT staff to perform these data retrievals. However, by adapting their existing Excel formula skills, such professionals can develop a SQL skillset which empowers them to perform their own data analysis on large databases.

Getting Started

Many SQL learning guides rely on showing syntax without giving a context. The approach in this tutorial is different. You will learn SQL through a simple but typical data analysis scenario. Each step introduces a new SQL concept that's tied to a common business question. To further aid explanation, each query is explained in Excel equivalent terms. By the end of this tutorial, you should feel comfortable getting read-only access to your company's databases and running your own queries from a command prompt.

The Excel concepts that correlate to SQL include the following:

EXCEL	SQL
Table	Table
Row	Record
Column	Field
Sort	ORDER BY
Filter	WHERE
Pivot Table	GROUP BY
VLOOKUP	JOIN
Array Functions	Aggregate Functions

The SQL concepts covered in this tutorial include:

SQL	Description
SELECT	Statement
GROUP BY	Statement
ORDER BY	Keyword
DESC	Optional ORDER BY keyword
NOT	Keyword
AND, OR, IN, LIKE	Operator
WHERE	Clause
JOIN	Clause
AS	Alias

Scenario

Imagine that you are a product manager at an e-commerce or SaaS startup. For such a business, the critical metric is Customer Lifetime Value (CLV or CLTV). If you know how much revenue each customer brings to your business over their lifetime with the product of service, you have a much better idea of how to meet revenue goals and develop your offering.

Knowing the CLV for each customer leads to more efficient business decisions. Which customer acquisition channel has high CLVs? Which one is losing money? Are there regional differences?

This tutorial teaches SQL by walking through the steps of computing the average Customer Lifetime Value from two database tables: user data (“users”) and transactional records (“payments”).

Setup

This tutorial uses SQLite3, a free, widely deployed, embedded SQL database engine that is included with Mac OSX. A free download for Windows is available at www.sqlite.org.

SQLite requires a command line interface. (I promise that you won’t have to do much here besides entering SQL commands.) On the Mac, the command line interface is Terminal, which can be found by searching for it in your applications folder. Windows users will need to open a CMD window, which can be found through a search for "command prompt". If you're brand new to the console, take a moment to do a web search for "terminal commands ________", inserting your operating system (Windows, Mac, Linux, etc.) in the blank, and familiarize yourself with simple navigational commands like Change Directory ("cd") to move around your file system.

To run the SQLite3 interactive shell, download **this file** as bootstrap.sql. Next, start Terminal, change to the directory containing the bootstrap.sql file you just downloaded, and execute the following command:

```
$sqlite3 -initbootstrap.sql
```

Note that the dollar sign ($) symbolizes the start of a new command line in your terminal; you do not need to type it as part of the command. This starts the SQLite interactive shell and displays the prompt:

```
sqlite>
```

The boostrap.sql file creates two pre-populated tables. To display the table names, enter the command ".tables" at the prompt:

```
sqlite>.tables payments users
```

Now that we know the names of the tables (payments, users), we can enter a command to view the table contents. Note that unlike Excel, you need to actually query a database to view the data in a table.

The simplest SQL query fetches all the columns and rows in a table. DO NOT DO THIS for very large tables as it may cause your computer to hang. However, for this tutorial, the table is small (just 10 rows), so entering the statement "SELECT * FROM users;" is safe. Note the statement syntax requires a trailing semicolon.

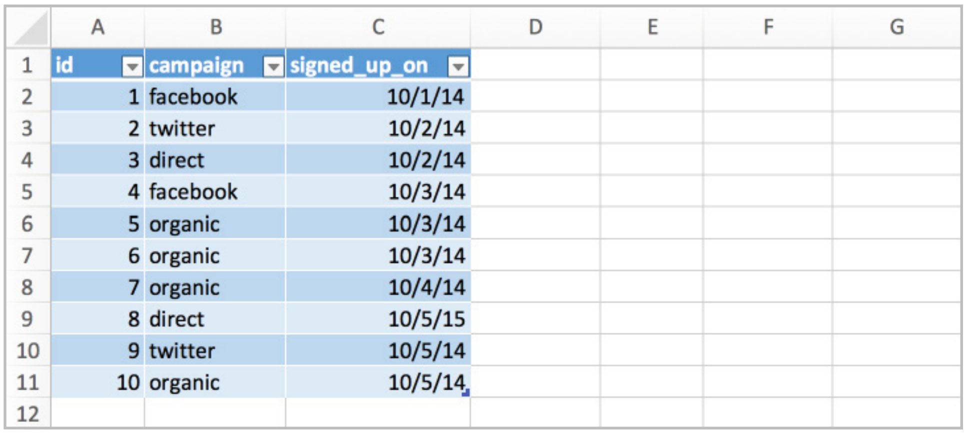
```
sqlite> SELECT * FROM users;

id          campaign    signed_up_on
----------  ----------  ------------
1           facebook    2014-10-01
2           twitter     2014-10-02
3           direct      2014-10-02
4           facebook    2014-10-03
5           organic     2014-10-03
6           organic     2014-10-03
7           organic     2014-10-04
8           direct      2014-10-05
9           twitter     2014-10-05
10          organic     2014-10-05
```

Note that in SQL, "*" is a wild card meaning "grab everything". In this case, SELECT * means "grab every column". You'll see later on that in real world queries, we usually only SELECT the columns we need to create a report or perform a computation.

As you can see, the users table has three columns, or fields:

• "id" is the user's ID. This is a unique identifier that is also referenced in the payments table
• "campaign" is the campaign used to acquire that user.
• "signed_up_on" is the date when the user signed up for the campaign.

Below is an Excel equivalent table. By comparison, each field is represented with a column:



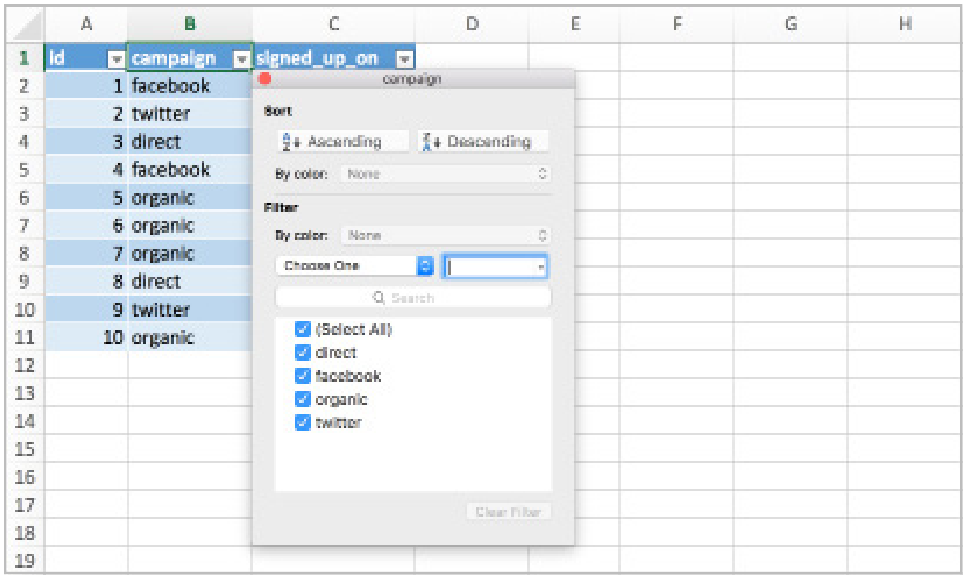
	A	B	C	D	E	F	G
1	id	campaign	signed_up_on				
2	1	facebook	10/1/14				
3	2	twitter	10/2/14				
4	3	direct	10/2/14				
5	4	facebook	10/3/14				
6	5	organic	10/3/14				
7	6	organic	10/3/14				
8	7	organic	10/4/14				
9	8	direct	10/5/15				
10	9	twitter	10/5/14				
11	10	organic	10/5/14				
12							

Sorting with ORDER BY

Just like Excel, SQL lets you sort data by one or more fields (columns). To do so, we add the ORDER BY keyword to the SELECT statement:

```
sqlite> SELECT * FROM users ORDER BY
campaign;

id          campaign    signed_up_on
----------  ----------  ------------
3           direct      2014-10-02
8           direct      2014-10-05
1           facebook    2014-10-01
4           facebook    2014-10-03
5           organic     2014-10-03
6           organic     2014-10-03
7           organic     2014-10-04
10          organic     2014-10-05
2           twitter     2014-10-02
9           twitter     2014-10-05
```

The ORDER BY keyword correlates to the Excel Sort command:



id	campaign	signed_up_on
1	facebook	
2	twitter	
3	direct	
4	facebook	
5	organic	
6	organic	
7	organic	
8	direct	
9	twitter	
10	organic	

Just like Excel, SQL can sort by more than one field:

```
sqlite> SELECT * FROM users ORDER BY campaign;

id          campaign    signed_up_on
----------  ----------  ------------
1           facebook    10/1/2014
3           direct      10/2/2014
2           twitter     10/2/2014
4           facebook    10/3/2014
5           organic     10/3/2014
6           organic     10/3/2014
8           direct      10/5/2014
7           organic     10/5/2014
10          organic     10/5/2014
9           twitter     10/5/2014
```

In the above query, the users table was sorted by the signed_up_on date and then at a second level by campaign.
To sort in reverse order, add DESC to ORDER BY:

```
sqlite> SELECT * FROM users ORDER BY campaign;

id          campaign    signed_up_on
----------  ----------  ------------
2           twitter     2014-10-02
9           twitter     2014-10-05
5           organic     2014-10-03
6           organic     2014-10-03
7           organic     2014-10-04
10          organic     2014-10-05
1           facebook    2014-10-01
4           facebook    2014-10-03
3           direct      2014-10-02
8           direct      2014-10-05
```

To sort records in descending order by multiple fields, use DESC with each field:

```
sqlite> SELECT * FROM users ORDER BY
campaign DESC, signed_up_onDESC;

id          campaign    signed_up_on
----------  ----------  ------------
9           twitter     10/5/2014
7           organic     10/5/2014
10          organic     10/5/2014
8           direct      10/5/2014
5           organic     10/3/2014
6           organic     10/3/2014
4           facebook    10/3/2014
2           twitter     10/2/2014
3           direct      10/2/2014
1           facebook    10/1/2014
```

Filtering with WHERE

In Excel, we display rows that meet certain criteria by using a filter. In SQL, we achieve this with a WHERE clause, which can include an even broader range of conditions than an Excel filter.

For example, below is a query that fetches all the users who signed up organically. Note that the syntax for a WHERE clause requires criteria in single quotation marks:

```
sqlite> SELECT * FROM users WHERE campaign = 'organic';

id            campaign      signed_up_on
----------    ----------    ------------
5             organic       2014-10-03
6             organic       2014-10-03
7             organic       2014-10-04
10            organic       2014-10-05
```

As you can see, we have WHERE campaign = 'organic' in the above query. An analog in Excel is going to the "campaign" column and selecting just 'organic'.

What if you want to fetch multiple values? No problem, use the SQL IN operator:

```
sqlite> SELECT * FROM users WHERE campaign IN
('facebook', 'twitter');

id            campaign      signed_up_on
----------    ----------    ------------
1             facebook      2014-10-01
2             twitter       2014-10-02
4             facebook      2014-10-03
9             twitter       2014-10-05
```

The above query fetches all the users that signed up through Facebook or Twitter. As you can see, multiple criteria are separated by commas inside the parentheses.

What if you wanted to fetch all the users EXCEPT those that came from Facebook or Twitter? This requires the NOT keyword:

```
sqlite> SELECT * FROM users WHERE campaign NOT IN
('facebook', 'twitter');

id           campaign     signed_up_on
----------   ----------   ------------
3            direct       2014-10-02
5            organic      2014-10-03
6            organic      2014-10-03
7            organic      2014-10-04
8            direct       2014-10-05

10           organic      2014-10-05
```

It's significant to note that it's not easy to "exclude" particular values for a given column in Excel.

Okay, but all the filtering thus far involved a single field. Can SQL filter by multiple fields? The answer is yes, by using the AND operator to query for more than one condition. The following query fetches all the users that signed up for Facebook or Twitter campaigns on Oct. 1, 2014.

```
sqlite> SELECT * FROM users WHERE campaign IN
('facebook', 'twitter') AND signed_up_on = '2014-10-01';

id           campaign     signed_up_on
----------   ----------   ------------
1            facebook     2014-10-01
```

Now it's time to show that the SQL WHERE clause is more powerful than Excel filters. In addition to AND for more than one condition, SQL includes the OR operator, which queries for one condition OR another. So, the query "Get me all the users who signed up before 2014- 10-04 OR who came in organically" looks like:

```
sqlite> SELECT * FROM users WHERE campaign = 'organic'
OR signed_up_on < '2014-10-04';

id            campaign      signed_up_on
----------    ----------    ------------
1             facebook      2014-10-01
2             twitter       2014-10-02
3             direct        2014-10-02
4             facebook      2014-10-03
5             organic       2014-10-03
6             organic       2014-10-03
7             organic       2014-10-04
10            organic       2014-10-05
```

This is not easy to do in Excel. Most likely you would need to create an additional column.

Filtering and Sorting

As you might have guessed by now, SQL allows you to filter and sort in one pass. The syntax is simple: Have both WHERE and ORDER BY in the SELECT statement, but make sure WHERE comes before ORDER BY. Here is a query that fetches all the Facebook and Twitter sourced users, sorted by campaign.

```
sqlite> SELECT * FROM users WHERE campaign IN
('facebook', 'twitter') ORDER BY campaign;

id     campaign      signed_up_on
---    ---------     ------------
1      facebook      2014-10-01
4      facebook      2014-10-03
2      twitter       2014-10-02
9      twitter       2014-10-05
```

GROUP BY: SQL's PivotTable

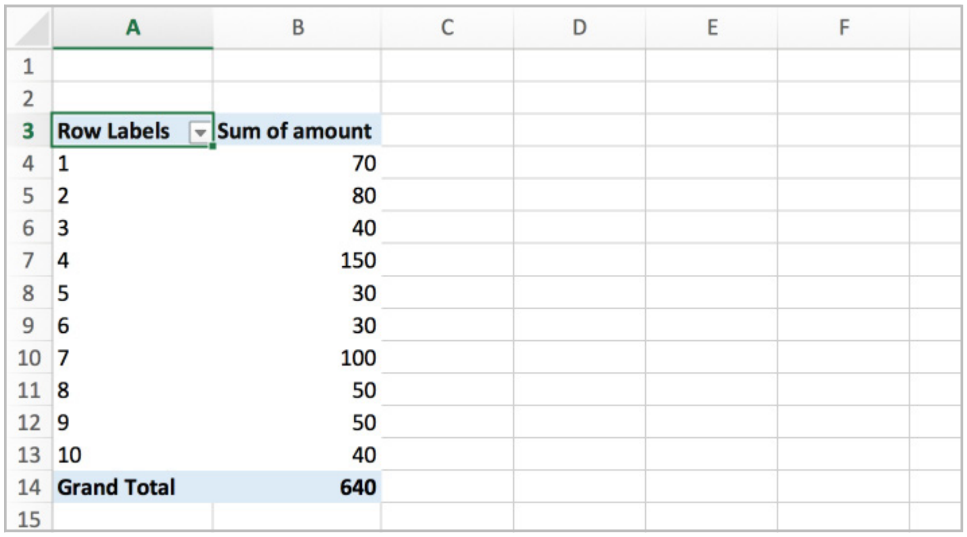
Now that we understand how to do basic sorting and filtering with SQL, we are ready to learn how to calculate CLV. For our simple model of an e-commerce website, we'll consider CLV to be the sum of all the purchases a customer has made to date. For this we'll need the GROUP BY statement.

The simplest way to describe the GROUP BY statement is: SQL's version of a PivotTable. To explain what this means, let's query the payments table, which stores transaction data (e.g. for an ecommerce site):

```
sqlite> SELECT * FROM payments;

id      amount    paid_on       user_id
----    -------   ----------    -------
1       40        2014-10-02    1
2       30        2014-10-03    1
3       30        2014-10-03    2
4       50        2014-10-03    4
5       100       2014-10-04    4
6       30        2014-10-05    5
7       30        2014-10-06    6
8       50        2014-10-07    8
9       50        2014-10-08    2
10      50        2014-10-09    9
11      40        2014-10-10    10
12      100       2014-10-11    7
13      40        2014-10-12    3
```

A typical question to ask here is how much money did each of the 10 users spend on our site?

In Excel, this is as simple as creating a PivotTable:



Row Labels	Sum of amount
1	70
2	80
3	40
4	150
5	30
6	30
7	100
8	50
9	50
10	40
Grand Total	**640**

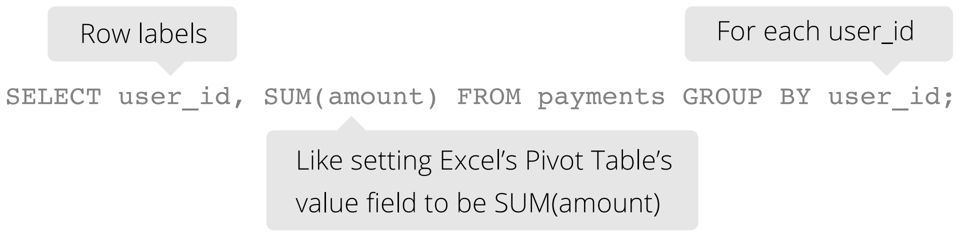
The equivalent operation in SQL uses the GROUP BY statement:

```
sqlite> SELECT user_id, SUM(amount) FROM payments GROUP
BY user_id;

user id      SUM (amount)
-------      ------------
1            70
2            80
3            40
4            150
5            30
6            30
7            100
8            50
9            50
10           40
```

As the name suggests, the GROUP BY statement groups the table's rows based on a column name. In the payments query, GROUP BY user_id groups the payments table based on its user_id column.

In the query above, we're also using an aggregation function SUM() to tell our query how it should combine the values from multiple rows with the same user_id. Without this aggregation, the query would just return the value of the last row for each user_id (not useful).

Correlating the query to Excel looks like:



```
SELECT user_id, SUM(amount) FROM payments GROUP BY user_id;
```

Row labels
For each user_id
Like setting Excel's Pivot Table's value field to be SUM(amount)

Needless to say, GROUP BY can be combined with WHERE (filtering) and ORDER BY (sorting). Syntactically, WHERE comes before GROUP BY, which in turn comes before ORDER BY.

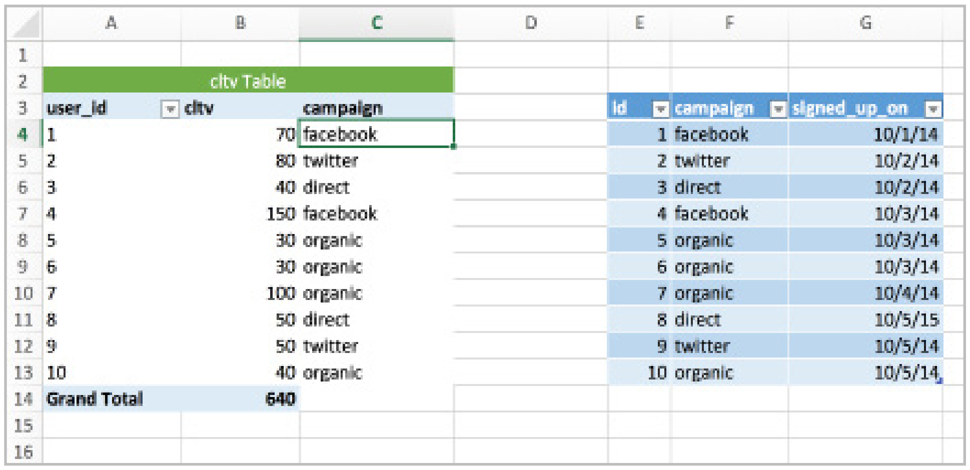
For example, the query below calculates CLV for users with user_id > 5, sorted by CLV amount. Note the use of the SQL AS for giving the SUM(amount) field an alias (new name) in the output

```
sqlite> SELECT user_id, SUM(amount) AS clv FROM
payments WHERE user_id > 5 GROUP BY user_id ORDER BY
clv;

user id      clv
-------      ------------
6            30
10           40
8            50
9            50
7            100
```

JOIN:ConnectingMultipleSources of Information

We have learned several SQL concepts by first querying the users table and then by querying the payments table. But what if we want our query to consider fields from both tables? Let's say you wish to determine which campaign (organic/Facebook/Twitter/direct) yields the highest CLV? In this case, we need to cross reference data from the payments table with the campaign field from the users table.

In Excel, this is where VLOOKUP comes in. Namely, you VLOOKUP the campaign column in the users table for the CLV PivotTable:



	A	B	C	D	E	F	G
1							
2	cltv Table						
3	user_id	cltv	campaign		id	campaign	signed_up_on
4	1	70	facebook		1	facebook	10/1/14
5	2	80	twitter		2	twitter	10/2/14
6	3	40	direct		3	direct	10/2/14
7	4	150	facebook		4	facebook	10/3/14
8	5	30	organic		5	organic	10/3/14
9	6	30	organic		6	organic	10/3/14
10	7	100	organic		7	organic	10/4/14
11	8	50	direct		8	direct	10/5/15
12	9	50	twitter		9	twitter	10/5/14
13	10	40	organic		10	organic	10/5/14
14	Grand Total	640					
15							
16							

This is all well and good, but what if you have more than 10 users? What if, say, you have 100,000 users? Excel won't be able to handle the data, or even if it can, the UI begins to lag. And if you have 10 million users (which happens with decently sized e-commerce websites), Excel is definitely not going to be sufficient.

SQL databases (e.g., MySQL, PostgreSQL, etc.) are far more scalable than Excel—even more so with proper indices—and can perform more complex computations in a more automated manner. (Indexing is a fascinating and deep topic in databases, but it's beyond the scope of this tutorial. Just know that grouping by indexed columns is much faster than grouping by unindexed columns.)

To perform the same operation in SQL, a JOIN clause must first combine tables based on the common field between them and then use the combined rows of data to select the output:

```
sqlite> SELECT joined.user_id, joined.clv,
users.campaign FROM (SELECT user_id, SUM(amount) AS clv
FROM payments GROUP BY user_id) AS
clv_table
JOIN users
ON clv_table.user_id = users.id;

user_id    clv     campaign
---------  -----   ----------
1          70      facebook
2          80      twitter
3          40      direct
4          150     facebook
5          30      organic
6          30      organic
7          100     organic
8          50      direct
9          50      twitter
10         40      organic
```

In the query above, the first line chooses columns from the joined table. Note the field names begin with their table name followed by a period. For example, users.campaign.

In the FROM line, we use the original CLV query in parentheses to create a new table we aliased as 'clv_table'. This aggregate table is created at the time of the query, stored in memory, and then destroyed after the result is output to our terminal. We can use as many nested SELECT statements as we want to create temporary aggregate tables to answer more complex questions.

The JOIN and ON lines show that we are joining the users table onto the joined table that we just aliased. But how do you join two tables?

This is answered on the last line: it matches the rows of the joined table with the rows of the users table so that the clv_table's user_id field equals the users table's id field. There is no strictly equivalent process for this in VLOOKUP because VLOOKUP forces you to JOIN by the leftmost columns. In SQL, you can JOIN by any desired columns!

The Other CLV: Campaign Lifetime Value

Now that we have a single view of the user IDs, campaign sources, and CLVs, we can calculate which campaign has the highest return thus far. To do so, we simply run one more GROUP BY, grouping per-user CLV by campaign:

```
sqlite> SELECT campaign, SUM(clv) AS campaign_value
FROM (SELECT clv_table.user_id,
clv_table.clv, users.campaign FROM (SELECT user_id,
SUM(amount) AS clv FROM payments GROUP
BY user_id) AS clv_table
JOIN users
ON clv_table.user_id = users.id) GROUP
BY campaign ORDER BY campaign_value DESC;

campaign   campaign_value
---------  --------------
facebook   220
organic    200
twitter    130
direct     90
```

From this we discover that Facebook is the winner! It looks like there is a healthy amount of organic traffic, comparable to Twitter. Note that this data only accounts for how much revenue is generated for each campaign. Different campaigns have different costs, so if you wish to calculate the ROI of different campaigns (and I hope you do!), you need the data for how much money is spent on each campaign.

Conclusion

• GROUP BY in SQL is like PivotTable in Excel, except it scales better with larger datasets (especially with proper indices).

• JOIN in SQL is like VLOOKUP in Excel, except JOIN is more flexible.

• You can query against an output of another query to ask more complex questions against your data.

If you want to process massive datasets using SQL, check out Treasure Data's cloud analytics platform. For more use-case specific SQL query templates, check out our library!

Feel free to contact me on Twitter @kiyototamura if you have any questions.

Kiyoto Tamura

Kiyoto began his career in quantitative finance before making a transition into the startup world. A math nerd turned software engineer turned developer marketer, he enjoys postmodern literature, statistics, and a good cup of coffee.

Sample Queries

1.

```
SELECT_____FROM_____WHERE____ORDER BY____;
```

2. When using an aggregation function:

```
SELECT Agg.Func.(______) FROM_____GROUP BY ______;
```

3. When using JOINs:

```
SELECT table1.column2, table2.column3 FROM table1
JOIN table2
ON table1.column1=table2.column1;
```

Logical Operators

1. AND is used to select rows from a column that satisfy both the conditions in a conditional statement

2. OR is used to select rows from a column that satisfy one of the conditions in a conditional statement

3. NOT is used before any conditional statement to select the rows that do not satisfy the condition

Comparison Operators:

1. =
2. <
3. >
4. <=
5. >=
6. != means NOT EQUAL
7. IN is used to specify the values we need in a column
8. BETWEEN is used to specify the range and select only the rows in that range
9. IS NULL is used to select the rows with no values in a column

Aggregation Functions:

1. SUM is used to add all the rows in a column
2. AVG is used to average all the rows in a column
3. MIN is used to find the minimum value in all the rows in a column
4. MAX is used to find the maximum value in all the rows in a column
5. COUNT is used to count all the rows in a column
6. DISTINCT is used to count all unique the values in a column