Table of Contents

## Introduction

When I first learned SQL in college, it was through the “pyramid” pattern. We studied the basics of relational algebra and set theory for the first 3-4 weeks. I honestly don’t remember anything. I only learned SQL properly once I got a job and started using it.

If you pick up a typical SQL introductory book or course, it follows the same “pyramid” pattern. You start with the language basics, you learn the syntax, then you build up from there to increasingly complex concepts.

That way of learning rarely sticks

If you’ve ever gotten really good at something over the course of your career, you’ve built up a collection of patterns and best practices that go beyond the basics. You use these patterns, whether consciously or not, every day to solve complex problems with ease.

They exist in every field. Chefs don’t create recipes from scratch. They use common cooking patterns, like sautéing vegetables, browning meat, making dough, using spices, etc. to create delicious meals.

Likewise fiction writers use character patterns like: “antihero”, “sidekick”, “mad scientist”, “girl next door”; plot patterns like romantic comedy, drama, red herring, foreshadowing, cliffhangers, etc. to write novels, TV shows and movies.

These basic mental constructs act like LEGO pieces for your mind allowing you to build up and evolve a solution to a complex problem from base level components by mixing and matching.

I believe that studying and learning patterns is the fastest way to level up your skills in any field, especially that of data and SQL. But there’s a problem. These patterns are rarely codified and taught. Professors who’ve never work in industry don’t know them, so they teach you theoretical concepts while professionals aren’t even aware they’re using them.

You need to first spend years in the field and be exposed to many different data problems to even have a chance to learn them. Second you need to be aware of the fact that you’re using them so you can codify them.

I’ve been writing SQL for ~15 years. I’ve seen and written hundreds of thousands of lines of code. Over time I noticed a set of patterns and best practices I always come back to when writing queries. These patterns made my code more efficient, easier to understand and a breeze to maintain.

When looking at other people’s code, even though it was correct, I could easily spot how with a just few changes they could improve its the readability and maintainability. I really wanted to help them learn what I knew.

I looked around for a book or course that taught these patterns but couldn’t find one, so I decided to write it. I’ve codified and organized them into 5 categories to make it easy for you to learn them and start using them right away.

Those categories are: 1. Query Decomposition patterns teach you how to decompose large queries into smaller pieces making it a breeze to solve just about any complex problem. 2. Query Maintainability patterns teach you how to organize your code in ways that make it easier to read, understand and maintain in the future. 3. Query Performance patterns teach you ways to make your code more efficient and faster without sacrificing clarity. It’s a delicate balance because performant code can look really messy. 4. Query Robustness patterns teach you ways to make your code resistant to messy data, such as duplicates, missing values, NULLs, etc.

Once you study and learn these patterns you’ll be able to: - Get really good at advanced SQL really fast - Write efficient, production-ready SQL that’s easy to read and maintain - Learn best practices when querying so you can avoid common traps - Solve complex queries like an expert without having to wait decades to become one

### How is this book organized?

I’m a huge fan of project-based learning. The idea that you can learn anything if you can come up with an interesting project to use that thing in has proven incredibly useful in my own career. I taught myself data science in just a few months by focusing on a project to build a model for scoring leads based on their propensity to convert.

For this reason, I’ve organized the book around a complex and useful data project that not only will help you understand the patterns but will do so in context. Learning anything in context will help you retain the material much better the next time you need to apply it.

In this book we’ll be working with the Stackoverflow dataset that’s publicly available in BigQuery for free. You can access it [here](https://console.cloud.google.com/marketplace/product/stack-exchange/stack-overflow).

BigQuery also offers 1TB/month free of processing so even if you sign up for it with a credit card, you can complete this entire course for free. I’ve made sure that the queries we run are small and limited so you won’t have to worry about being charged.

Using this dataset we’re going to build a “user reputation” table which calculates reputation metrics per user. This type of table can be very useful if you want to build for example a customer 360 table or if you want to build a model for customer LTV calculation.

As we go through the project, we’ll cover each pattern when it arises. That will help you understand why we’re using the pattern at that exact moment. Each chapter will cover a select group of patterns while building on the previous chapters.

In Chapter 1 we introduce the project

We will get into the details of the project in the next chapter.

## Chapter 1: Introducing The Project

As discussed in the introduction, in this chapter we’re going to get into the details of the project that will help you learn the SQL Patterns in context.

Many books start by teaching you the basic concepts first and by the time you get to use them, you’ve already forgotten them. By taking a project based approach, we circumvent that problem entirely and you get the learn these patterns simply by following along.

So what is this project?

As you saw in the introduction, we’re using a real-world, public dataset from StackOverflow (SO). In case you’re not familiar, SO is a popular website where users can ask technical questions about any topic (programming, SQL, databases, data analysis, stats, etc.) and other users can answer these questions.

Based on the quality of the answers, as determined by the community upvotes and downvotes, the users who give them can gain reputation and badges which they can use as social proof both on the SO site and on other websites.

Using this dataset we’re going to build a “user reputation” table which calculates reputation metrics per user. This type of table can be very useful if you want to do customer engagement analysis or if you want to identify your best customers. It also happens to be quite perfect to demonstrate most of the patterns described in this book.

The schema of what it would look something like this:

user\_id  
user\_name  
posts  
answers  
questions  
streak\_in\_days  
posts\_per\_day  
answers\_per\_day  
questions\_per\_day  
upvotes\_per\_day  
downvotes\_per\_day  
comments\_on\_user\_posts\_per\_day  
comments\_by\_user\_per\_day  
answers\_per\_post\_ratio  
upvotes\_per\_post  
downvotes\_per\_post  
comments\_per\_post\_on\_user\_posts  
comments\_by\_user\_per\_per\_post

Why is this useful?

In many marketing campaigns it is very useful to segment your customers based on certain behavior and engagement criteria and a table like this is perfect. You’re basically aggregating various metrics assuming you can associate them with a user.

Because we have one row per user, it means we have to transform all user related data at the user\_id, date granularity. We’ll talk about how to do that in the next few chapters.

### Understanding the Data Model

In order to succeed in any SQL endeavor one of the first things we must do is to understand the data model we’re working with. This may already exist in the form of documentation but more often than not you’ll have to build the model as you go. You might even learn the hard way, like I did, by making mistakes. That’s ok.

The SO data model is quite complex but if you search for it online you’ll find the version corresponding to their internal database which doesn’t match BigQuery. That’s because BigQuery modifies the data in certain ways to avoid self joins. I’ve taken the liberty of drawing it up for you and we’ll cover it now.

Here’s a look at the Entity-RElationship (ER) diagram [[StackOverflow BQ ER Diagram.png]]

There are 8 tables that represent the various post types. You can get this result by using the INFORMATION\_SCHEMA views in BigQuery like this:

SELECT table\_name  
FROM `bigquery-public-data.stackoverflow.INFORMATION\_SCHEMA.TABLES`  
WHERE table\_name like 'posts\_%'  
  
table\_name |  
--------------------------+  
posts\_answers |  
posts\_orphaned\_tag\_wiki |  
posts\_tag\_wiki |  
posts\_questions |  
posts\_tag\_wiki\_excerpt |  
posts\_wiki\_placeholder |  
posts\_privilege\_wiki |  
posts\_moderator\_nomination|

We’ll be focusing on just two of them for our project: 1. posts\_questions contains all the question posts 2. posts\_answers contains all the answer posts

They both have the same schema:

SELECT column\_name, data\_type  
FROM `bigquery-public-data.stackoverflow.INFORMATION\_SCHEMA.COLUMNS`  
WHERE table\_name = 'posts\_answers'  
  
column\_name |data\_type|  
------------------------+---------+  
id |INT64 |  
title |STRING |  
body |STRING |  
accepted\_answer\_id |STRING |  
answer\_count |STRING |  
comment\_count |INT64 |  
community\_owned\_date |TIMESTAMP|  
creation\_date |TIMESTAMP|  
favorite\_count |STRING |  
last\_activity\_date |TIMESTAMP|  
last\_edit\_date |TIMESTAMP|  
last\_editor\_display\_name|STRING |  
last\_editor\_user\_id |INT64 |  
owner\_display\_name |STRING |  
owner\_user\_id |INT64 |  
parent\_id |INT64 |  
post\_type\_id |INT64 |  
score |INT64 |  
tags |STRING |  
view\_count |STRING |

Both tables have an id column that identifies a single post, creation\_date that identifies the timestamp when the post was created and a few other attributes like score for the upvotes and downvotes.

Note the parent\_id column which signifies a hierarchical structure. The parent\_id is a one-to-many relationship that links up an answer to the corresponding question. A single question can have multiple answers but an answer belongs to one and only one question.

As you can see there’s no user\_id in the table because posts and users have a many-to-many relationship. They’re connected via the post\_history table.

SELECT column\_name, data\_type  
FROM `bigquery-public-data.stackoverflow.INFORMATION\_SCHEMA.COLUMNS`  
WHERE table\_name = 'post\_history'  
  
column\_name |data\_type|  
--------------------+---------+  
id |INT64 |  
creation\_date |TIMESTAMP|  
post\_id |INT64 |  
post\_history\_type\_id|INT64 |  
revision\_guid |STRING |  
user\_id |INT64 |  
text |STRING |  
comment |STRING |

Both post types (question and answer) have a one-to-many relationship to the post\_history. A single post can have many types of activities identified by the post\_history\_type\_id column.

This id indicates the different types of activities a user can do on the site. We’re only concerned with the first 6. You can see the rest of them [here](https://meta.stackexchange.com/questions/2677/database-schema-documentation-for-the-public-data-dump-and-sede/2678#2678) if you’re curious.

1 = Initial Title - initial title *(questions only)* 2 = Initial Body - initial post raw body text 3 = Initial Tags - initial list of tags *(questions only)* 4 = Edit Title - modified title *(questions only)* 5 = Edit Body - modified post body (*raw markdown*) 6 = Edit Tags - modified list of tags \_(questions only)

The first 3 indicate when a post is first submitted and the next 3 when a post is edited.

This table also connects to the users table. A single user can perform multiple activities on a post. This is known as a bridge table between the users and posts which have a many-to-many relationship which cannot be modeled otherwise.

The users table has one row per user and contains user attributes such as name, reputation, etc. We’ll use some of these attributes in our final table.

SELECT column\_name, data\_type  
FROM `bigquery-public-data.stackoverflow.INFORMATION\_SCHEMA.COLUMNS`  
WHERE table\_name = 'users'  
  
column\_name |data\_type|  
-----------------+---------+  
id |INT64 |  
display\_name |STRING |  
about\_me |STRING |  
age |STRING |  
creation\_date |TIMESTAMP|  
last\_access\_date |TIMESTAMP|  
location |STRING |  
reputation |INT64 |  
up\_votes |INT64 |  
down\_votes |INT64 |  
views |INT64 |  
profile\_image\_url|STRING |  
website\_url |STRING |

Next we take a look at the comments table. It has a zero-to-many relationship with posts and with users, which means that both a user or a post could have 0 comments. The connection to the posts indicates comments on a post and the connection to the user indicates comments by a user.

SELECT column\_name, data\_type  
FROM `bigquery-public-data.stackoverflow.INFORMATION\_SCHEMA.COLUMNS`  
WHERE table\_name = 'comments'  
  
column\_name |data\_type|  
-----------------+---------+  
id |INT64 |  
text |STRING |  
creation\_date |TIMESTAMP|  
post\_id |INT64 |  
user\_id |INT64 |  
user\_display\_name|STRING |  
score |INT64 |

Finally the votes table represents the upvotes and downvotes on a post. Once we connect a post to a user, we can compute This is exactly what we need to compute the total vote count on a user’s post which will indicate how good the question or the answer is. This table has a granularity of one row per vote per post per date.

SELECT column\_name, data\_type  
FROM `bigquery-public-data.stackoverflow.INFORMATION\_SCHEMA.COLUMNS`  
WHERE table\_name = 'votes'  
  
column\_name |data\_type|  
-------------+---------+  
id |INT64 |  
creation\_date|TIMESTAMP|  
post\_id |INT64 |  
vote\_type\_id |INT64 |

Note that the votes table is connected to a post, so in order for us to get upvotes and downvotes on a user’s post, we’ll need to join it with the users table.

## Chapter 2: Core Concepts

### Granularity

Granularity (also known as the grain) is a measure of the level of detail that determines an individual row in a table or view. This is extremely important when it comes to joins or aggregating data. A low granularity table means a very low level of detail, like one row per transaction.

Granularity is usually expressed as the number of unique rows for each column or combination of columns.

For example the users table has one row per user. That is the lowest grain on it. The post\_history table, on the other hand, contains a log of all the changes that a user performs on a post on a given date and time. Therefore the granularity is one row per user, per post, per timestamp.

The comments table contains a log of all the comments on a post by a user on a given date so its granularity is also one row per user, per post, per date.

The votes table contains a log of all the upvotes and downvotes on a post on a given date. It has separate rows for upvotes and downvotes so its granularity is one row per post, per vote type, per date.

To find a table’s granularity you either read the documentation, or if that doesn’t exist, you make an educated guess and check. Trust but verify. Real world data is messy

How do you check? It’s easy.

For the post\_history table we can run the following query:

SELECT   
 creation\_date,  
 post\_id,  
 post\_history\_type\_id,  
 user\_id,  
 COUNT(\*) AS total\_rows  
FROM `bigquery-public-data.stackoverflow.post\_history`  
GROUP BY 1,2,3,4  
HAVING COUNT(\*) > 1;

So I’m aggregating by all the columns I expect to make up the unique row and filtering for any that invalidate my assumption. If my hunch is correct, I should get 0 rows from this query.

But we don’t! We get a bunch of duplicate rows:

creation\_date |post\_id |post\_history\_type\_id|user\_id |total\_rows|  
-----------------------+--------+--------------------+--------+----------+  
2020-07-20 05:00:26.413|62964197| 34| -1| 2|  
2020-08-05 16:31:15.220|63272171| 5|14038907| 2|  
2018-10-08 09:54:40.990|40921767| 5| 4826457| 2|  
2020-05-07 22:02:27.877|61637980| 34| -1| 2|  
2018-10-13 05:26:22.243|52784015| 5| 6599590| 2|  
2021-01-03 10:35:35.693|65550662| 5|12833166| 2|  
2018-12-02 14:28:12.947|53576317| 5|10732059| 2|  
2018-09-05 04:16:26.440|52140985| 4| 3623424| 3|  
2018-12-17 22:43:27.800|53826052| 8| 1863229| 2|  
2018-09-13 17:13:31.490|52321596| 5| 5455640| 2|

This means we have to be careful when joining with this table on post\_id, user\_id, creation\_date, post\_history\_type\_id and we’d have to deal with the duplicate issue first. Let’s see a couple of methods for doing that.

### Aggregating Data

When you aggregate data you’re moving from a level of low granularity to a level of higher granularity. Please note that this is a “one-way street.” By aggregating data you’re reducing the level of detail and by definition removing information. But if you store data at this aggregated level, you lose the details.

That’s why it’s very common in data warehouses to store data at the lowest possible grain you have it and then aggregate it up to whatever level is needed for reporting. You can also use aggregating to deal with duplicate rows, and we have some so let’s do it.

Let’s refer again to the previous example.

If I simply select the columns I want without aggregation, we get duplicates which as we mentioned earlier will mess up joins later. (Rows 2 and 3 are the same)

SELECT   
 creation\_date,  
 post\_id,  
 post\_history\_type\_id,  
 user\_id  
FROM   
 `bigquery-public-data.stackoverflow.post\_history`  
WHERE   
 post\_id = 63272171   
 AND user\_id = 14038907  
 AND post\_history\_type\_id = 5  
  
creation\_date |post\_id |post\_history\_type\_id|user\_id |  
-----------------------+--------+--------------------+--------+  
2020-08-05 15:42:25.130|63272171| 5|14038907|  
2020-08-05 16:31:15.220|63272171| 5|14038907|  
2020-08-05 16:31:15.220|63272171| 5|14038907|  
2020-08-05 16:37:23.983|63272171| 5|14038907|  
2020-08-05 15:34:38.187|63272171| 5|14038907|

By simply adding a GROUP BY we can easily solve this problem

SELECT   
 creation\_date,  
 post\_id,  
 post\_history\_type\_id,  
 user\_id  
FROM   
 `bigquery-public-data.stackoverflow.post\_history`  
WHERE   
 post\_id = 63272171   
 AND user\_id = 14038907  
 AND post\_history\_type\_id = 5  
GROUP BY 1,2,3,4;  
  
creation\_date |post\_id |post\_history\_type\_id|user\_id |  
-----------------------+--------+--------------------+--------+  
2020-08-05 16:37:23.983|63272171| 5|14038907|  
2020-08-05 16:31:15.220|63272171| 5|14038907|  
2020-08-05 15:34:38.187|63272171| 5|14038907|  
2020-08-05 15:42:25.130|63272171| 5|14038907|

Notice that for the purposes of removing duplicates we don’t need to use an aggregate function like COUNT() or MAX(), MIN() You can achieve the same effect by using DISTINCT

Of course using aggregate functions is the most common way to aggregate data. Summing up or counting multiple rows are still the workhorse of aggregation. We’ll use that a lot in our project.

Here’s a traditional application of it:

SELECT  
 user\_id,  
 CAST(creation\_date AS DATE) AS activity\_date,  
 COUNT(\*) as total\_comments  
FROM  
 `bigquery-public-data.stackoverflow.comments`  
WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
GROUP BY  
 1,2

### Pivoting Data

Here’s another pattern that’s very commonly used for aggregation:

SELECT  
 post\_id,  
 CAST(v.creation\_date AS DATE) AS activity\_date,  
 SUM(CASE WHEN vote\_type\_id = 2 THEN 1 ELSE 0 END) AS total\_upvotes,  
 SUM(CASE WHEN vote\_type\_id = 3 THEN 1 ELSE 0 END) AS total\_downvotes,  
FROM  
 `bigquery-public-data.stackoverflow.votes` v  
WHERE  
 TRUE  
 AND v.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND v.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
GROUP BY  
 1,2

This pattern is commonly known as **Pivoting** because we take data that looks like this

id |creation\_date |post\_id |vote\_type\_id|  
---------+-----------------------+--------+------------+  
239119706|2021-09-23 20:00:00.000|69301792| 2|  
239123009|2021-09-23 20:00:00.000|69301792| 3|  
239200936|2021-09-24 20:00:00.000|69301792| 2|  
239087730|2021-09-22 20:00:00.000|69301792| 3|  
239199214|2021-09-24 20:00:00.000|69301792| 2|  
239118872|2021-09-23 20:00:00.000|69301792| 3|  
239135887|2021-09-23 20:00:00.000|69301792| 2|  
239127938|2021-09-23 20:00:00.000|69301792| 2|  
239147153|2021-09-23 20:00:00.000|69301792| 3|  
239153591|2021-09-23 20:00:00.000|69301792| 2|  
239168079|2021-09-23 20:00:00.000|69301792| 2|  
239121664|2021-09-23 20:00:00.000|69301792| 3|  
239117803|2021-09-23 20:00:00.000|69301792| 2|  
239117878|2021-09-23 20:00:00.000|69301792| 3|  
239116194|2021-09-23 20:00:00.000|69301792| 2|  
239130104|2021-09-23 20:00:00.000|69301792| 2|  
239157135|2021-09-23 20:00:00.000|69301792| 2|  
239142497|2021-09-23 20:00:00.000|69301792| 3|  
239157729|2021-09-23 20:00:00.000|69301792| 2|  
239129111|2021-09-23 20:00:00.000|69301792| 3|

and turn it into this

post\_id |activity\_date|total\_upvotes|total\_downvotes|  
--------+-------------+-------------+---------------+  
69301792| 2021-09-24| 10| 7|  
69301792| 2021-09-25| 2| 0|  
69301792| 2021-09-23| 0| 1|

by “pivoting” on the vote type

You’ll notice that I manipulate the timestamp column creation\_date into just a date field without the time information. Date fields are special when it comes to aggregation because they have many layers of granularities built in.

Given a single timestamp, we can construct granularities for seconds, minutes, hours, days, weeks, months, quarters, years, decades. We do that by using one of the many date manipulation functions like CAST(), DATE\_TRUNC(), DATE\_PART(), etc. There’s way too many of them to mention here and nobody remembers the exact syntax so you just look it up in the documentation.

### Joining Data

Joining tables is one of the most basic functions in SQL since the databases are designed to minimize redundancy of information and the only to do that is to spread information out into multiple tables. This is called normalization. Joins then allow us to get all the information back in a single piece by combining these tables together.

I assume you’re familiar with them if you’re reading this book, so what I wanted to share with you are certain anti-patterns involving joins that always creep up and burn analysts and data scientists.

#### Granularity Multiplication Antipattern

If any of tables has duplicates for the columns being joined on, the final result set will be multiplied by the number of duplicates.

For example in our case the users table has a grain of one row per user:

SELECT  
 id,  
 display\_name,  
 creation\_date ,  
 reputation,  
 views  
FROM `bigquery-public-data.stackoverflow.users`  
WHERE id = 8974849;  
  
  
id |display\_name|creation\_date |reputation|views|  
-------+------------+-----------------------+----------+-----+  
8974849|neutrino |2017-11-20 18:16:46.653| 790| 107|

Whereas the post\_history table has multiple rows for the same user:

SELECT  
 id,  
 creation\_date,  
 post\_id,  
 post\_history\_type\_id,  
 user\_id   
FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
WHERE  
 TRUE  
 AND ph.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND ph.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 AND ph.user\_id = 8974849;  
  
  
id |creation\_date |post\_id |post\_history\_type\_id|user\_id|  
---------+-----------------------+--------+--------------------+-------+  
250199272|2021-07-14 00:54:58.127|68372251| 2|8974849|  
250199273|2021-07-14 00:54:58.127|68372251| 1|8974849|  
250199274|2021-07-14 00:54:58.127|68372251| 3|8974849|  
250263915|2021-07-15 00:01:07.497|68387743| 2|8974849|  
250263916|2021-07-15 00:01:07.497|68387743| 1|8974849|  
250263917|2021-07-15 00:01:07.497|68387743| 3|8974849|  
250316277|2021-07-15 16:32:44.163|68400451| 2|8974849|

If we join them on user\_id the granularity of the final result will be multiplied to have as many rows per user:

SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date,  
 post\_history\_type\_id  
FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 INNER JOIN `bigquery-public-data.stackoverflow.users` u ON u.id = ph.user\_id  
WHERE  
 TRUE  
 AND ph.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND ph.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 AND ph.user\_id = 8974849;  
  
post\_id |user\_id|user\_name|activity\_date |post\_history\_type\_id|  
--------+-------+---------+-----------------------+--------------------+  
68078326|8974849|neutrino |2021-06-22 02:03:45.830| 2|  
68078326|8974849|neutrino |2021-06-22 02:03:45.830| 1|  
68078326|8974849|neutrino |2021-06-22 02:03:45.830| 3|  
68273785|8974849|neutrino |2021-07-06 11:56:05.827| 2|  
68273785|8974849|neutrino |2021-07-06 11:56:05.827| 1|  
68273785|8974849|neutrino |2021-07-06 11:56:05.827| 3|  
68277148|8974849|neutrino |2021-07-06 16:40:53.003| 2|  
68277148|8974849|neutrino |2021-07-06 16:40:53.003| 1|  
68277148|8974849|neutrino |2021-07-06 16:40:53.003| 3|  
68273785|8974849|neutrino |2021-07-06 12:02:11.913| 5|

Notice how the user\_name repeats for each row.

So if the history table has 10 entries for the same user and the users table has 1, the final result will contain 10 x 1 entries for the same user. If for some reason the users contained 2 entries for the same user (messy real world data), we’d see 10 x 2 = 20 entries for that user in the final result.

This is extremely important when doing analysis because a single duplicate row will multiply all your results by a factor of n and all your numbers will be inflated.

#### Accidental Inner Join Antipattern

Did you know that SQL will ignore a LEFT JOIN clause and perform an INNER JOIN instead if you make this one simple mistake? This is one of those SQL hidden secrets which sometimes gets asked as a trick question in interviews so strap in.

When doing a LEFT JOIN you’re intending to show all the results on the table in the FROM clause but if you need to limit

Let’s take a look at the example query from above:

SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date  
FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 INNER JOIN `bigquery-public-data.stackoverflow.users` u ON u.id = ph.user\_id  
WHERE  
 TRUE  
 AND ph.post\_id = 4  
ORDER BY  
 activity\_date;

This query will produce 58 rows. Now let’s change the INNER JOIN to a LEFT JOINand rerun the query:

SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date  
FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 LEFT JOIN `bigquery-public-data.stackoverflow.users` u ON u.id = ph.user\_id  
WHERE  
 TRUE  
 AND ph.post\_id = 4  
ORDER BY  
 activity\_date;

Now we get 72 rows!! If you scan the results, you’ll notice several where both the user\_name and the user\_id are NULL which means they’re unknown. These could be people who made changes to that post and then deleted their accounts. Notice how the INNER JOIN was filtering them out? That’s what I mean by data reduction which we discussed previously.

Suppose we only want to see users with a reputation of higher than 50. That’s seems pretty straightforward just add the condition to the where clause

SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date  
FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 LEFT JOIN `bigquery-public-data.stackoverflow.users` u ON u.id = ph.user\_id  
WHERE  
 TRUE  
 AND ph.post\_id = 4  
 AND u.reputation > 50  
ORDER BY  
 activity\_date;

We only get 56 rows! What happened?

Adding filters on the where clause for tables that are left joined will ALWAYS perform an INNER JOIN except for one single condition where the left join is preserved. If we wanted to filter rows in the users table and still do a LEFT JOIN we have to add the filter in the join condition like so:

SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date  
FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 LEFT JOIN `bigquery-public-data.stackoverflow.users` u ON u.id = ph.user\_id  
 AND u.reputation > 50   
WHERE  
 TRUE  
 AND ph.post\_id = 4  
ORDER BY  
 activity\_date;

The ONLY time when putting a condition in the WHERE clause does NOT turn a LEFT JOIN into an INNER JOIN is when checking for NULL. This is very useful when you want to see the missing data on the table that’s being left joined. Here’s an example

SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date  
FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 LEFT JOIN `bigquery-public-data.stackoverflow.users` u ON u.id = ph.user\_id   
WHERE  
 TRUE  
 AND ph.post\_id = 4  
 AND u.id is NULL  
ORDER BY  
 activity\_date;

Now we only get the 12 missing users

### Appending Data

You can combine the rows from multiple tables in order to make a longer table by simply appending the rows from one table by using the UNION operator.

For example we can combine two of the posts tables like this:

SELECT  
 id AS post\_id,  
 'question' AS post\_type,  
FROM  
 `bigquery-public-data.stackoverflow.posts\_questions`  
WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
  
UNION ALL  
  
SELECT  
 id AS post\_id,  
 'answer' AS post\_type,  
FROM  
 `bigquery-public-data.stackoverflow.posts\_answers`  
WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)

There are two types of unions, UNION ALL and UNION (distinct)

UNION ALL will append two tables without checking if they have the same exact row. This might cause duplicates but it’s really fast. If you know for sure your tables don’t contain duplicates, this is the preferred way to append them.

UNION (distinct) will append the tables but remove all duplicates from the final result thus guaranteeing unique rows for the final result set. This of course is slower because of the extra operations to remove duplicates. Use this only when you’re not sure if the tables contain duplicates or you cannot remove duplicates beforehand.

Most SQL flavors only use UNION keyword for the distinct version, but BigQuery forces you to use UNION DISTINCT in order to make the query far more explicit

Appending rows to a table also has two requirements: 1. The number of the columns from all tables has to be the same 2. The data types of the columns from all the tables has to line up

One of the most annoying things when appending two or more tables with a lot of columns is lining up all the columns in the right order. There’s been many a time when I’ve had to use Excel to line up the columns. There’s no shame in admitting that.

As a rule of thumb, whenever you’re appending tables, it’s a good idea to add a constant column to indicate the source table or some kind of type. This is helpful when appending say activity tables to create a long, time-series table and you want to identify each activity type in the final result set.

You’ll notice in my query above I create a post\_type column indicating where the data is coming from.

#### De-Pivoting Data Pattern

We saw how to pivot data above, but can you reverse the process? Well, sort of. As I said before, aggregation is a “one-way street” meaning that once you aggregate, you lose important information, however it is possible to “de-pivot” data using the UNION operator like this:

WITH votes\_pivot AS (  
 SELECT  
 post\_id,  
 CAST(v.creation\_date AS DATE) AS activity\_date,  
 SUM(CASE WHEN vote\_type\_id = 2 THEN 1 ELSE 0 END) AS total\_upvotes,  
 SUM(CASE WHEN vote\_type\_id = 3 THEN 1 ELSE 0 END) AS total\_downvotes,  
 FROM  
 `bigquery-public-data.stackoverflow.votes` v  
 WHERE  
 TRUE  
 AND v.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND v.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 AND post\_id = 69301792  
 GROUP BY  
 1,2  
)  
SELECT   
 activity\_date,  
 2 AS vote\_type\_id,  
 total\_upvotes AS votes  
FROM  
 votes\_pivot  
  
UNION ALL   
  
SELECT   
 activity\_date,  
 3 AS vote\_type\_id,  
 total\_downvotes AS votes  
FROM   
 votes\_pivot  
ORDER BY  
 activity\_date;  
  
activity\_date|vote\_type\_id|votes|  
-------------+------------+-----+  
 2021-09-23| 2| 0|  
 2021-09-23| 3| 1|  
 2021-09-24| 3| 7|  
 2021-09-24| 2| 10|  
 2021-09-25| 3| 0|  
 2021-09-25| 2| 2|

The above query uses a CTE (Common Table Expression) which will be covered in the next chapter in more detail. You can see how we’ve “de-pivoted” the data but not quite recovered the original 20 rows.

## Chapter 4: Query Decomposition

### Introduction to CTEs

“The only way to write complex software that won’t fall on its face is to build it out of simple modules connected by well-defined interfaces, so that most problems are local and you can have some hope of fixing or optimizing a part without breaking the whole” -Eric S. Raymond

One of the core principles of software engineering is that the only way you can build a complex system is by building simple, self-contained modules and connecting them together. This is known as the **Modularity Principle**

Similarly. every complex query can and should be broken down into small, simple modules. These modules should have a single purpose or responsibility which allows them to be written, tested and debugged independently.

When I first started writing queries professionally, I wanted to show off my smarts. I wanted to get the entire query written in one fell swoop, one single, perfect, beautiful query that gave the correct answer. Reality, however, had other plans.

You see real world data is messy. From inconsistent field types, missing or duplicate rows, unexpected values, etc. I learned pretty quickly that complex queries, no matter how simple they might seem, needed to be broken down into smaller modules.

Initially I did this with temporary tables. This way I could test each query individually as I wrote it. Then, when I combined them together to solve the big complex query I knew that the results would be accurate. This also had the added benefit of making my code easier to read and maintain by others.

Later I learned how to use CTEs (Common Table Expressions) for the same purpose. CTEs or Common Table Expressions are temporary views whose scope is limited to the current query. They are not stored in the database; they only exist while the query is running and are only accessible in that query.

\_Side Note: Even though CTEs have been part of the definition of the SQL standard since 1999, it has taken many years for database vendors to implement them. Some versions of older databases (like MySQL before 8.0, PostgreSQL before 8.4, SQL Server before 2005) do not have support for CTEs. All the modern cloud vendors have support for CTEs

We define a single CTE using the WITH keyword and then use it in the main query like this:

-- Define CTE  
WITH <cte\_name> AS (  
 SELECT col1, col2  
 FROM table\_name  
)  
  
-- Main query  
SELECT \*  
FROM <cte\_name>

We can define multiple CTEs similarly using the WITH keyword like this:

-- Define CTE 1  
WITH <cte1\_name> AS (  
 SELECT col1  
 FROM table1\_name  
)  
  
-- Define CTE 2  
, <cte2\_name> AS (  
 SELECT col1  
 FROM table2\_name  
)  
  
-- Main query  
SELECT \*  
FROM <cte1\_name> AS cte1  
JOIN <cte2\_name> AS cte2 ON cte1.col1 = cte2.col1

Notice that you only use the WITH keyword once then you separate them using a comma in front of the name of the each one.

We can refer to a previous CTE in a new CTE thus chaining them together like this:

-- Define CTE 1  
WITH <cte1\_name> AS (  
 SELECT col1  
 FROM table1\_name  
)  
  
-- Define CTE 2 by referring to CTE 1  
, <cte2\_name> AS (  
 SELECT col1  
 FROM cte1\_name  
)  
  
-- Main query  
SELECT \*  
FROM <cte2\_name>

This pattern allows for a lot of flexibility with multi-step calculations. We’ll see that later.

When CTEs are used it lets us read a query top to bottom and easily understand what’s going on. When sub-queries are used, it’s a lot harder to trace the logic and figure out which column is defined where and what scope it has because you have to read the innermost subquery first.

Just because we can chain CTEs, it doesn’t mean we can do that infinitely. There are practical limitations on levels of chaining because after a while the query will end up becoming computationally complex. This depends on the database system you’re using.

### Query Decomposition

In order to understand how to break down a large, complex query into CTEs we need to think about what we want to achieve and map out a solution. We’re looking to build a table at the user\_id, date level starting from tables with user activity and date.

We know that a user can perform any of the following activities on any given date: 1. Post a question 2. Post an answer 3. Edit a question 4. Edit an answer 5. Comment on a post 6. Receive a comment on their post 7. Receive a vote (upvote or downvote) on their post

We can break this down into several subproblems and map out a solution like this:

Sub-problem 1 In order to get the first 4 activities at the user\_id, date granularity we first need to solve the problem of reducing the granularity of the post\_history to the user\_id, date, post\_id level. Then we’ll join that back to the posts (by combining questions and answers) so we can get the post types. Finally we will aggregate data to the user\_id, date level and calculate some of the metrics.

Sub-problem 2 We will apply the same granularity reduction pattern to comments and votes so that in the end we have 3-4 CTEs all at the same granularity of user\_id, date.

Sub-problem 3 Once we get all activity types on the same granularity, we will join them on user\_id and date in order to calculate all the final metrics per user.

### Chaining CTEs Pattern

We saw how we can define multiple CTEs above and we also saw how each CTE can use a previous CTE which allows us to chain them together to solve out complex query.

To solve the first sub-problem we have to define a CTE that gets the post activity for each user\_id, post\_id, activity\_type, date combination. We then need to restrict this activity to only creation and editing because we don’t care about the other kinds. That makes for a perfect small, self-contained CTE which can also be used later when we need to join in votes to users.

WITH post\_activity AS (  
 SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date,  
 CASE WHEN ph.post\_history\_type\_id IN (1,2,3) THEN 'created'  
 WHEN ph.post\_history\_type\_id IN (4,5,6) THEN 'edited'   
 END AS activity\_type  
 FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 INNER JOIN `bigquery-public-data.stackoverflow.users` u on u.id = ph.user\_id  
 WHERE  
 TRUE   
 AND ph.post\_history\_type\_id BETWEEN 1 AND 6  
 AND user\_id > 0 --exclude automated processes  
 AND user\_id IS NOT NULL --exclude deleted accounts  
 AND ph.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND ph.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2,3,4,5  
)  
SELECT \*  
FROM post\_activity  
WHERE user\_id = 16366214  
ORDER BY activity\_date   
  
post\_id |user\_id |user\_name |activity\_date |activity\_type|  
--------+--------+-----------+-----------------------+-------------+  
68226767|16366214|Tony Agosta|2021-07-02 10:18:42.410|created |  
68441160|16366214|Tony Agosta|2021-07-19 09:16:57.660|created |  
68469502|16366214|Tony Agosta|2021-07-21 08:29:22.773|created |  
68469502|16366214|Tony Agosta|2021-07-26 07:31:43.513|edited |  
68441160|16366214|Tony Agosta|2021-07-26 07:32:07.387|edited |

Notice that we’re performing an INNER JOIN which will eliminate any users that do not exist in both tables. For our purposes this is exactly what you want but remember that I recommended starting with a LEFT JOIN in the previous chapter. That’s only a recommendation not a rule. Check your data to be sure.

The astute reader would have also noticed the aggregation pattern to reduce granularity. Remember that we don’t need the use an aggregate function to actually aggregate our data, we can just use the GROUP BY keyword to reduce granularity and remove duplicates.

Now that we have the post\_activity CTE, we need to join it with the questions and answers and then aggregate the activity.

Since the schema of both post\_questions and post\_answers is identical, we can combine them into a single CTE using UNION ALL and then we join with post\_activity. This is a textbook example of **CTE chaining.**

WITH post\_activity AS (  
 SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date,  
 CASE ph.post\_history\_type\_id  
 WHEN 1 THEN 'created'  
 WHEN 4 THEN 'edited'   
 END AS activity\_type  
 FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 INNER JOIN `bigquery-public-data.stackoverflow.users` u on u.id = ph.user\_id  
 WHERE  
 TRUE   
 AND ph.post\_history\_type\_id IN (1,4)  
 AND user\_id > 0 --exclude automated processes  
 AND user\_id IS NOT NULL  
 AND ph.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND ph.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2,3,4,5  
)  
,post\_types as (  
 SELECT  
 id AS post\_id,  
 'question' AS post\_type,  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_questions`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 UNION ALL  
 SELECT  
 id AS post\_id,  
 'answer' AS post\_type,  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_answers`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 )  
SELECT  
 pt.user\_id,  
 pt.user\_name,  
 DATE\_TRUNC(pt.activity\_date, DAY) AS date,  
 SUM(CASE WHEN activity\_type = 'created'  
 AND post\_type = 'question' THEN 1 ELSE 0 END) AS question\_created,  
 SUM(CASE WHEN activity\_type = 'created'  
 AND post\_type = 'answer' THEN 1 ELSE 0 END) AS answer\_created,  
 SUM(CASE WHEN activity\_type = 'edited'  
 AND post\_type = 'question' THEN 1 ELSE 0 END) AS question\_edited,  
 SUM(CASE WHEN activity\_type = 'edited'  
 AND post\_type = 'answer' THEN 1 ELSE 0 END) AS answer\_edited   
FROM post\_types pt  
 JOIN post\_activity pa ON pt.post\_id = pa.post\_id  
WHERE user\_id = 16366214  
GROUP BY 1,2,3

You’ll notice that I’m using a DATE\_TRUNC() function on the activity\_date field. What does it do? As it turns out, a date or timestamp field contains multiple levels of granularity embedded all of which are accessible via date functions.

Let’s review what we’ve done so far. We created two CTEs, one for post types and one for the post activity by user. We joined these two CTEs and pivoted the data at the user\_id, date level in order to create 4 new metrics.

You might ask why didn’t we join the post\_types as a subquery in the first CTE and then aggregate everything? Well that’s the idea behind single purpose. If we need to use the first CTE later on, which we do, then by joining to smaller CTE, we ensure that the query is more efficient. Yes a modern database might optimize by saving the results somewhere instead of running the query again, but this way we don’t assume.

Also the nice thing about using CTEs vs sub-queries is that you can read the query top to bottom and understand exactly what’s happening. With sub-queries you typically have to read from the inside out. You read the innermost subquery first then you work your way out. It can become pretty tedious to keep it all in your head.

Also if we wanted to test each CTE we can highlight the portions of the code we care about and run just that.

### Important Notes

Before we go further I want to highlight a few things regarding CTEs. First like I said earlier, not all databases support them. You’d have to be on the latest version in order to get all the benefits. We’re focusing on cloud data warehouses here so that’s not really an issue.

## Chapter 5: Query Maintainability

### DRY Pattern (Don’t Repeat Yourself )

In the previous section we saw how we can decompose a large complex query into multiple smaller components which can be chained together to give us the final result. We said that an added benefit to doing this is that it makes the query more readable. In that same vein, the DRY principle ensures that your query is clean from unnecessary repetition.

The DRY principle states that if you find yourself copy-pasting the same chunk of code in multiple locations, it’s probably a good idea to put that code in a single CTE and reference that CTE where it’s needed.

To illustrate I’ll rewrite the query from the previous chapter so that it still produces the same result but it clearly shows repeating code

WITH post\_activity AS (  
 SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date,  
 CASE WHEN ph.post\_history\_type\_id IN (1,2,3) THEN 'created'  
 WHEN ph.post\_history\_type\_id IN (4,5,6) THEN 'edited'   
 END AS activity\_type  
 FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 INNER JOIN `bigquery-public-data.stackoverflow.users` u on u.id = ph.user\_id  
 WHERE  
 TRUE   
 AND ph.post\_history\_type\_id BETWEEN 1 AND 6  
 AND user\_id > 0 --exclude automated processes  
 AND user\_id IS NOT NULL --exclude deleted accounts  
 AND ph.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND ph.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2,3,4,5  
)  
, questions AS (  
 SELECT  
 id AS post\_id,  
 'question' AS post\_type,  
 pa.user\_id,  
 pa.user\_name,  
 pa.activity\_date,  
 pa.activity\_type  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_questions` q  
 INNER JOIN post\_activity pa ON q.id = pa.post\_id  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
)  
, answers AS (  
 SELECT  
 id AS post\_id,  
 'answer' AS post\_type,  
 pa.user\_id,  
 pa.user\_name,  
 pa.activity\_date,  
 pa.activity\_type  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_answers` q  
 INNER JOIN post\_activity pa ON q.id = pa.post\_id  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
)

This is definitely another valid solution to our query, if we then calculate the aggregates later on and combine them. The CTEs are small and single purpose, abiding by the modularity principle, however you’ll see that most of the code repeats.

The DRY principle says we should try and remove as much repeating code as possible, and since in our case the question and answer table have the same exact schema, that’s a perfect candidate for appending rows.

### Appending Rows Pattern

In the previous section we combined the two posts tables using the UNION ALL operator to make a single post\_types CTE like this:

post\_types as (  
 SELECT  
 id AS post\_id,  
 'question' AS post\_type,  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_questions`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 UNION ALL  
 SELECT  
 id AS post\_id,  
 'answer' AS post\_type,  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_answers`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 )

Let’s take a moment to see how this pattern works. Just like a JOIN adds columns to a result set the UNION operator appends rows to it by combining two or more tables length-wise. There are two types of unions, UNION ALL and UNION (distinct)

UNION ALL will append two tables without checking if they have the same exact row. This might cause duplicates but it’s really fast. If you know for sure your tables don’t contain duplicates, as in our case this is the preferred way to append two tables.

UNION (distinct) will append the tables but remove all duplicates from the final result thus guaranteeing unique rows for the final result set. This of course is slower because of the extra operations to remove duplicates. Use this only when you’re not sure if the tables contain duplicates or you cannot remove duplicates beforehand.

Most SQL flavors only use UNION keyword for the distinct version, but BigQuery forces you to use UNION DISTINCT in order to make the query far more explicit

Appending rows to a table also has two requirements: 1. The number of the columns from all tables has to be the same 2. The data types of the columns from all the tables has to line up

One of the most annoying things when appending two or more tables with a lot of columns is lining up all the columns in the right order. There’s been many a time when I’ve had to use Excel to line up the columns. There’s no shame in admitting that.

As a rule of thumb, whenever you’re appending tables, it’s a good idea to add a constant column to indicate the source table or some kind of type. This is helpful when appending say activity tables to create a long, time-series table and you want to identify each activity type in the final result set.

You’ll notice in my query above I create a post\_type column indicating where the data is coming from.

### Creating Views Pattern

There are many cases where a piece of code can be useful outside of the query you’re writing because it encapsulates something in a neat little package. In cases like these it makes a lot of sense to make a view with that snippet of code. This view can also be materialized so that querying it is fast and efficient.

You won’t know what that piece of code could be ahead of time but if you find yourself copying and pasting something in multiple files, that’s a great opportunity to create a view. This goes back to the DRY principle but in this case applied across multiple files.

Creating a view is easy:

CREATE OR REPLACE VIEW <view\_name> AS  
 SELECT col1  
 FROM table1  
 WHERE col1 > x;

Once created you can run:

SELECT col1  
FROM <view\_name>

This view is now stored in the database but it doesn’t take up any space (unless it’s materialized) It only stores the query which is executed each time you select from the view or join the view in a query.

\*Note: In BigQuery views are considered like CTEs so they count towards the maximum level of nesting. That is if you call a view from inside a CTE, that’s two levels of nesting and if you then join that CTE in another CTE that’s three levels of nesting. BigQuery has a hard limitation on how deep nesting can go beyond which you can no longer run your query. At that point, perhaps the view is best materialized into a table.

## Chapter 6: Query Performance

The query performance principle states that your queries should as fast as possible while still accurate.

It’s not just about speed. Yes it’s important to have reports that execute in seconds, but these days performance is directly tied to cost, whether through compute resources spent to run the query or the overall amount of data scanned so performance considerations are directly tied to your bottom line.

As we continue building up our complex query, we now need to solve the second sub-problem, dealing with comments. If you recall from the ER diagram chapter, the comments table contains a log of all the comments on a post by a user on a given date so its granularity is also one row per user, per post, per date.

In order to calculate user level metrics from this table we’ll need to split up the work into a couple of CTEs, one to get comments by a user on a given date and the other to get comments on a user’s post on a given date.

Here’s a snippet that explains the approach: (this won’t run by itself btw because of the CTE reference)

, comments\_by\_user AS (  
 SELECT  
 user\_id,  
 CAST(DATE\_TRUNC(creation\_date, DAY) AS DATE) AS activity\_date,  
 COUNT(\*) as total\_comments  
 FROM  
 `bigquery-public-data.stackoverflow.comments`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2  
)  
, comments\_on\_user\_post AS (  
 SELECT  
 pa.user\_id,  
 CAST(DATE\_TRUNC(c.creation\_date, DAY) AS DATE) AS activity\_date,  
 COUNT(\*) as total\_comments\_on\_post  
 FROM  
 `bigquery-public-data.stackoverflow.comments` c  
 INNER JOIN post\_activity pa ON pa.post\_id = c.post\_id  
 WHERE  
 TRUE  
 AND pa.activity\_type = 'created'  
 AND c.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND c.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2  
)

Throughout the book we’ve been using a pattern for improving query performance that I’ll highlight now, but you’ll soon notice in all the other pieces of code.

In every CTE, I’m adding the condition

AND c.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
AND c.creation\_date <= CAST('2021-09-30' as TIMESTAMP)

This condition filters data to only 3 months from the entire history and demonstrates one of the core principles of query performance:

### Reducing Data Pattern

By reducing the number of rows you’re accessing upfront in a CTE, you ensure that the final result is smaller and the query runs faster.

For example the following two queries are technically equivalent in that you’ll get the same exact result

WITH comments\_by\_user AS (  
 SELECT  
 user\_id,  
 CAST(DATE\_TRUNC(creation\_date, DAY) AS DATE) AS activity\_date,  
 COUNT(\*) as total\_comments  
 FROM  
 `bigquery-public-data.stackoverflow.comments`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2  
)  
SELECT \*  
FROM comments\_by\_user   
WHERE user\_id = 16366214

WITH comments\_by\_user AS (  
 SELECT  
 user\_id,  
 CAST(DATE\_TRUNC(creation\_date, DAY) AS DATE) AS activity\_date,  
 COUNT(\*) as total\_comments  
 FROM  
 `bigquery-public-data.stackoverflow.comments  
 GROUP BY  
 1,2  
)  
SELECT \*  
FROM comments\_by\_user   
WHERE user\_id = 16366214

However in the second query, if I were to join that CTE with another table or CTE in the query it would join with a much larger table, many more rows which would make the final query really slow.

### SELECT \* Antipattern

It’s very tempting to always do SELECT \* in your queries or CTEs, especially if you don’t know which columns you need later. While this may be ok in a traditional RDBMS, in fact many introduction courses suggest to use this to explore data, cloud warehouse platforms are different.

This means that each column you select increases the amount of data you scan and how much compute resources you use. This in turn directly affects the performance of your queries and your bottom line. Platforms like BigQuery charge based o the amount of data you scan, even if you limit the rows. So a SELECT \* LIMIT 10 will still scan the entire table!

Throughout this book you’ve seen that my code only selects the columns that I need and restrict the data inside a CTE before I use that CTE in a join. We will continue this pattern while we add the final element to our query, the votes.

You can see here that despite all the columns available in the post\_questions and post\_answers tables we only get the post\_id here since the column post\_type has a static value and doesn’t affect the performance.

,post\_types AS (  
 SELECT  
 id AS post\_id,  
 'question' AS post\_type,  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_questions`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 UNION ALL  
 SELECT  
 id AS post\_id,  
 'answer' AS post\_type,  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_answers`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 )

### Premature Ordering Antipattern

So far we’ve created CTEs for all the post activity and the comments. The only piece remaining is the upvotes and downvotes. The votes table is only attached to a post, meaning it only tracks the votes at the post level not the user level. In order to get this at the user\_id, date level we’ll have to join it with the posts\_activity CTE like this:

, votes\_on\_user\_post AS (  
 SELECT  
 pa.user\_id,  
 CAST(DATE\_TRUNC(v.creation\_date, DAY) AS DATE) AS activity\_date,  
 SUM(CASE WHEN vote\_type\_id = 2 THEN 1 ELSE 0 END) AS total\_upvotes,  
 SUM(CASE WHEN vote\_type\_id = 3 THEN 1 ELSE 0 END) AS total\_downvotes,  
 FROM  
 `bigquery-public-data.stackoverflow.votes` v  
 INNER JOIN post\_activity pa ON pa.post\_id = v.post\_id  
 WHERE  
 TRUE  
 AND pa.activity\_type = 'created'  
 AND v.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND v.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2  
)

With this final section in place we can finally write the query that calculates all the metrics:

SELECT  
 user\_id,  
 user\_name,  
 total\_posts\_created,   
 total\_answers\_created,  
 total\_answers\_edited,  
 total\_questions\_created,  
 total\_questions\_edited,  
 total\_upvotes,  
 total\_comments\_by\_user,  
 total\_comments\_on\_post,  
 streak\_in\_days,  
 ROUND(IFNULL(SAFE\_DIVIDE(total\_posts\_created,   
 streak\_in\_days), 0), 1) AS posts\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_posts\_edited, streak\_in\_days), 0) AS NUMERIC), 1) AS edits\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_answers\_created, streak\_in\_days), 0) AS NUMERIC), 1) AS answers\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_questions\_created, streak\_in\_days), 0) AS NUMERIC), 1) AS questions\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_comments\_by\_user, streak\_in\_days), 0) AS NUMERIC), 1) AS comments\_by\_user\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_answers\_created,   
 total\_posts\_created), 0) AS NUMERIC), 1) AS answers\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_questions\_created, total\_posts\_created), 0) AS NUMERIC), 1) AS questions\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_upvotes, total\_posts\_created), 0) AS NUMERIC), 1) AS upvotes\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_downvotes, total\_posts\_created), 0) AS NUMERIC), 1) AS downvotes\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_comments\_by\_user, total\_posts\_created), 0) AS NUMERIC), 1) AS user\_comments\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_comments\_on\_post, total\_posts\_created), 0) AS NUMERIC), 1) AS comments\_on\_post\_per\_post  
FROM  
 total\_metrics\_per\_user  
ORDER BY   
 total\_questions\_created DESC;

You can see we’re finally ordering the results by total posts created. We could have been sorting data at any point in the query but it would have been unnecessary and a performance drain. So leave sorting at the very end if absolutely necessary or better yet leave it out and let the reporting tool handle it.

### Functions in WHERE Antipattern

## Chapter 7: Query Robustness

### Defensive Programming Patterns

From NULLs, to missing data, duplicate rows and random values, real world data is messy. A well-written query is robust enough to handle many of these cases without crashing or giving inaccurate results.

Real world data is not static. As companies push their development processes to release early and often, applications are in constant flux and their data is constantly changing. Bugs and other issues are always present so your queries need to be robust enough to handle these changes without breaking.

Below are some common patterns of what I like to call Defensive Programming, protecting against bad data.

### Type Conversion Defeneses

Type conversion is very important core principle of SQL. Tables can store many different types and the reason for this is that different types use up different storage and at the same time allow for more flexibility in calculations.

SQL mainly built support for primitive types such as strings, integers and dates.

By definition strings can be any length of characters (numbers, letters or symbols) but because of limitations of storage in the early days of computing, in many databases strings are stored as either CHAR(n) which represents a fixed-length string of n characters or VARCHAR(n) which represents a variable-length string of characters.

Strings can be considered “universal” data types because anything can be stored as a string. Doing this is very useful when loading data into a table from a text file like a comma-delimited CSV or tab-delimited TSV. If you try to load data in at the correct type and there are errors in the file, you’ll be dealing with a lot of anguish, so load as string first.

Once data is loaded in a table as strings, we can convert it to a more appropriate type and handle the errors. The standard function for converting data in SQL is CAST() Some other database implementations like SQL Server also use a custom function called CONVERT(). We can use CAST() to both convert between types (like string to date) or within the same type (like a timestamp to date)

Here’s an example of how type conversion works:

SELECT CAST('2021-12-01' as DATE);  
  
dt |  
----------+  
2021-12-01|

Suppose that for whatever reason the date was bad:

SELECT CAST('2021-12-01' as DATE);  
  
Error: Could not cast literal "2021-13-01" to type DATE at [1:13]

Obviously there’s no 13th month so BigQuery throws an error.

Same thing happens if the formatting was bad:

SELECT CAST('2021-12--01' as DATE);  
  
Message: Could not cast literal "2021-12--01" to type DATE at [1:13]

The extra dash in this case messes up conversion.

Same types of things happen if you try to convert a string to a number and the formatting is malformed or the data is not a number. So how do you deal with these issues?

### Ignore the Error Pattern

One of the easiest ways to deal with these issues is to simply ignore the malformed data. However the CAS() function will fail if it encounters an issue and we want our query to be robust.

To deal with this problem many databases introduce “safe” casting functions like SAFE\_CAST() in BigQuery or TRY\_CAST() in SQL Server. These functions will not fail when the formatting is unexpected but rather return NULL which then allows us to use IFNULL() or COALESCE() to replace NULL with a sensible value.

Here’s how that works:

SELECT SAFE\_CAST('2021-12--01' as DATE) AS dt;  
  
 dt|  
------+  
 NULL |

Now we can apply any of the functions that deal with NULL and replace it or just leave it.

SELECT IFNULL(SAFE\_CAST('2021-' as INTEGER), 0) AS num;  
  
num|  
---+  
 0|

### Force Formatting Pattern

While ignoring incorrect data is easy, you can’t always get away with it. Sometimes you need to extract the valuable data from the incorrect format. This is the time when you need to look for repeating patterns in the incorrect data and force the formatting.

Here’s a few examples: Suppose that all dates had extra dashes like this:

2021-12--01  
2021-12--02  
2021-12--03  
2021-12--04

Since this is a regular pattern, we can extract the meaningful numbers and force the formatting like this:

WITH dates AS (  
 SELECT '2021-12--01' AS dt  
 UNION ALL   
 SELECT '2021-12--02' AS dt  
 UNION ALL   
 SELECT '2021-12--03' AS dt  
 UNION ALL   
 SELECT '2021-12--04' AS dt  
 UNION ALL   
 SELECT '2021-12--05' AS dt  
)  
SELECT CAST(SUBSTRING(dt, 1, 4) || '-' ||   
 SUBSTRING(dt, 6, 2) || '-' ||   
 SUBSTRING(dt, 10, 2) AS DATE) AS date\_field   
FROM dates;  
  
date\_field  
----------  
2021-12-01  
2021-12-02  
2021-12-03  
2021-12-04  
2021-12-05

So as you can see in this example, we took advantage of the regularity of the incorrect formatting to extract the important information (the year, month and day) and reconstruct the correct formatting by concatenating strings via the || operator.

What if you have multiple types of regularities in your data? In some cases if information is aggregated from multiple sources you might have to deal with multiple types of formatting.

Let’s take a look at an example:

dt |  
-----------+  
2021-12--01|  
2021-12--02|  
2021-12--03|  
12/04/2021 |  
12/05/2021 |

Obviously we can’t force the same format for all the dates here so we’ll have to split this up and apply the force formatting pattern separately as long as we can detect the right patterns:

WITH dates AS (  
 SELECT '2021-12--01' AS dt  
 UNION ALL   
 SELECT '2021-12--02' AS dt  
 UNION ALL   
 SELECT '2021-12--03' AS dt  
 UNION ALL   
 SELECT '12/04/2021' AS dt  
 UNION ALL   
 SELECT '12/05/2021' AS dt  
)  
SELECT CAST(CASE WHEN dt LIKE '%-%--%'  
 THEN SUBSTRING(dt, 1, 4) || '-' ||  
 SUBSTRING(dt, 6, 2) || '-' ||  
 SUBSTRING(dt, 10, 2)  
 WHEN dt LIKE '%/%/%'  
 THEN SUBSTRING(dt, 7, 4) || '-' ||  
 SUBSTRING(dt, 1, 2) || '-' ||  
 SUBSTRING(dt, 4, 2)  
 END AS DATE) AS date\_field   
FROM dates;

As you can see in this example what we’re doing is separating each pattern via a CASE statement and handling each one differently. You can repeat this pattern as many times as you want to handle each case.

### Expect NULLs

This pattern can and should be used at any time even when you think the data is clean. Basically whenever you’re doing a LEFT JOIN or type conversion you should be expecting NULLs and protecting against them as a defensive measure. This is done to make sure that even if your data ever gets messy your query will not fail. It’s simply the use of IFNULL() or COALESCE() everywhere in your select.

NULLs in SQL represent unknown values. While the data may appear to be blank or empty, it’s not the same as an empty string or white space. You cannot compare NULLs to anything directly, for example you cannot say:

SELECT col1  
FROM table  
WHERE col2 = NULL;

You get NULL when you try to perform any type of calculation with NULL like adding or subtracting, multiplying or dividing because adding anything to an unknown value is still unknown. SQL deals with NULLs by using the IS keyword. IS NULL literally means is unknown. IFNULL() then means if this is unknown.

So in order to protect against unexpected NULLs it’s often a good idea for your production queries to wrap IFNUL() around all the fields.

WITH dates AS (  
 SELECT '2021-12--01' AS dt  
 UNION ALL   
 SELECT '2021-12--02' AS dt  
 UNION ALL   
 SELECT '2021-12--03' AS dt  
 UNION ALL   
 SELECT '12/04/2021' AS dt  
 UNION ALL   
 SELECT '12/05/2021' AS dt  
 UNION ALL   
 SELECT '13/05/2021' AS dt  
)  
SELECT IFNULL(SAFE\_CAST(  
 CASE WHEN dt LIKE '%-%--%'  
 THEN SUBSTRING(dt, 1, 4) || '-' ||  
 SUBSTRING(dt, 6, 2) || '-' ||  
 SUBSTRING(dt, 10, 2)  
 WHEN dt LIKE '%/%/%'  
 THEN SUBSTRING(dt, 7, 4) || '-' ||  
 SUBSTRING(dt, 1, 2) || '-' ||  
 SUBSTRING(dt, 4, 2)  
 END AS DATE), '1900-01-01') AS date\_field   
FROM dates;

This is the same query as above but implemented using “defensive coding” where we expect junk dates (like 13/05/2021) and we replace with a fixed date 1900-01-01 This way our query will not fail and afterwards we can investigate why the data was junk.

### Start With a LEFT JOIN

One of the ways you’ll get NULLs in your results is when you use a LEFT JOIN. Now there are legitimate reasons to use a LEFT JOIN, like when you know for sure the data will be missing on the right table but in this case we’re using it deliberately to avoid restricting the final results.

Whenever we use INNER JOIN the final result is always reduced down to just the matching rows from both tables. This means that if the history table has some strange user\_id that doesn’t exist in the users table, they will not show up in the final result. The same happens with the users that have no activity in post\_history

For the purposes of our project, we only want the active users so an INNER JOIN is very appropriate here. If we wanted everyone, we’d have to user a LEFT JOIN So why am I saying you should start with a LEFT JOIN? Get burned too many times and you eventually learn your lesson.

The mantra I keep repeating here is “real world data is messy” There are missing rows, duplicate rows, incorrect types and so on. Unless you know your data well and it’s being carefully monitored for these things, you should consider them in your joins.

### Dealing with divide by zero

Whenever you need to calculate ratios you always have to worry about division by zero. Going back to our principle of defensive programming, it makes sense to explicitly handle cases where the denominator can be zero.

The easiest way to handle this is by excluding zero values in the where clause as we do in our query

SELECT  
 ROUND(CAST(total\_comments\_on\_post /  
 total\_posts\_created AS NUMERIC), 1) AS comments\_on\_post\_per\_post  
FROM  
 total\_metrics\_per\_user  
WHERE  
 total\_posts\_created > 0  
ORDER BY   
 total\_questions\_created DESC;

This will work fine in most cases but what if you’re calculating multiple ratios and you don’t want to restrict the data for each one? One way to handle this is by using a CASE statement like this:

SELECT  
 CASE  
 WHEN total\_posts\_created > 0  
 THEN ROUND(CAST(total\_comments\_on\_post /  
 total\_posts\_created AS NUMERIC), 1)  
 ELSE 0  
 END AS comments\_on\_post\_per\_post,  
 CASE  
 WHEN streak\_in\_days > 0  
 THEN ROUND(CAST(total\_posts\_created /  
 streak\_in\_days AS NUMERIC), 1)  
 END AS posts\_per\_day  
FROM  
 total\_metrics\_per\_user  
ORDER BY   
 total\_questions\_created DESC;

This looks good and is pretty clean but not as elegant. BigQuery offers another way we can do this more cleanly. Just like the SAFE\_CAST() function, it has a SAFE\_DIVIDE() function which returns NULL in the case of divide-by-zero error. Then you can simply deal with the NULL value using IFNULL()

SELECT  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_posts\_created,   
 streak\_in\_days), 0) AS NUMERIC), 1) AS posts\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_comments\_by\_user,   
 total\_posts\_created), 0) AS NUMERIC), 1) AS user\_comments\_per\_post  
FROM  
 total\_metrics\_per\_user  
ORDER BY   
 total\_questions\_created DESC;

Now that’s far more elegant isn’t it? Snowflake also implements a similar function they call DIV() which automatically returns 0 if there’s a division by zero error eschewing the need for IFNULL() If your database has these functions, I highly recommend you use them.

### Dealing with messy strings

I said earlier that strings are the easiest way to store any kind of data (numbers, dates, strings) but strings also have their own issues, especially when you’re trying to join on a string field.

Here are some issues you’ll undoubtedly run into with strings. 1. Inconsistent casing 2. Space padding 3. Non-ASCII characters

Many databases are case sensitive so if the same string is stored with different cases it will not match when doing a join. Let’s see an example:

SELECT 'string' = 'String' AS test;  
  
test |  
-----+  
false|

As you can see, a different case causes the test to show as FALSE The only way to deal with this problem when joining on strings or matching patterns on a string is to convert all fields to upper or lower case.

SELECT lower('string') = lower('String') AS test;  
  
test|  
----+  
true|

Space padding is the other common issue you deal with strings.

SELECT 'string' = ' string' AS test;  
  
test |  
-----+  
false|

You deal with this by using the TRIM() function which removes all the leading and trailing spaces.

SELECT trim('string') = trim(' string') AS test;  
  
test|  
----+  
true|

We’ve now explored all the sections of the query so let’s see the whole thing in one place so we can see all the patterns in action.

-- Get the user name and collapse the granularity of post\_history to the user\_id, post\_id, activity type and date  
WITH post\_activity AS (  
 SELECT  
 ph.post\_id,  
 ph.user\_id,  
 u.display\_name AS user\_name,  
 ph.creation\_date AS activity\_date,  
 CASE WHEN ph.post\_history\_type\_id IN (1,2,3) THEN 'created'  
 WHEN ph.post\_history\_type\_id IN (4,5,6) THEN 'edited'   
 END AS activity\_type  
 FROM  
 `bigquery-public-data.stackoverflow.post\_history` ph  
 INNER JOIN `bigquery-public-data.stackoverflow.users` u on u.id = ph.user\_id  
 WHERE  
 TRUE   
 AND ph.post\_history\_type\_id BETWEEN 1 AND 6  
 AND user\_id > 0 --exclude automated processes  
 AND user\_id IS NOT NULL --exclude deleted accounts  
 AND ph.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND ph.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2,3,4,5  
)  
-- Get the post types we care about questions and answers only and combine them  
,post\_types AS (  
 SELECT  
 id AS post\_id,  
 'question' AS post\_type,  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_questions`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 UNION ALL  
 SELECT  
 id AS post\_id,  
 'answer' AS post\_type,  
 FROM  
 `bigquery-public-data.stackoverflow.posts\_answers`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 )  
 -- Finally calculate the post metrics at the user, date granularity  
, user\_post\_metrics AS (  
 SELECT  
 user\_id,  
 user\_name,  
 CAST(activity\_date AS DATE) AS activity\_date ,  
 SUM(CASE WHEN activity\_type = 'created'   
 AND post\_type = 'question' THEN 1 ELSE 0 END) AS questions\_created,  
 SUM(CASE WHEN activity\_type = 'created'   
 AND post\_type = 'answer' THEN 1 ELSE 0 END) AS answers\_created,  
 SUM(CASE WHEN activity\_type = 'edited'   
 AND post\_type = 'question' THEN 1 ELSE 0 END) AS questions\_edited,  
 SUM(CASE WHEN activity\_type = 'edited'   
 AND post\_type = 'answer' THEN 1 ELSE 0 END) AS answers\_edited,  
 SUM(CASE WHEN activity\_type = 'created' THEN 1 ELSE 0 END) AS posts\_created,  
 SUM(CASE WHEN activity\_type = 'edited' THEN 1 ELSE 0 END) AS posts\_edited  
 FROM post\_types pt  
 JOIN post\_activity pa ON pt.post\_id = pa.post\_id  
 GROUP BY 1,2,3  
)  
-- Calculate the comments metics at the user, date granularity  
, comments\_by\_user AS (  
 SELECT  
 user\_id,  
 CAST(creation\_date AS DATE) AS activity\_date,  
 COUNT(\*) as total\_comments  
 FROM  
 `bigquery-public-data.stackoverflow.comments`  
 WHERE  
 TRUE  
 AND creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2  
)  
, comments\_on\_user\_post AS (  
 SELECT  
 pa.user\_id,  
 CAST(c.creation\_date AS DATE) AS activity\_date,  
 COUNT(\*) as total\_comments  
 FROM  
 `bigquery-public-data.stackoverflow.comments` c  
 INNER JOIN post\_activity pa ON pa.post\_id = c.post\_id  
 WHERE  
 TRUE  
 AND pa.activity\_type = 'created'  
 AND c.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND c.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2  
)  
-- Calculate the votes metrics at the user, date granularity  
, votes\_on\_user\_post AS (  
 SELECT  
 pa.user\_id,  
 CAST(v.creation\_date AS DATE) AS activity\_date,  
 SUM(CASE WHEN vote\_type\_id = 2 THEN 1 ELSE 0 END) AS total\_upvotes,  
 SUM(CASE WHEN vote\_type\_id = 3 THEN 1 ELSE 0 END) AS total\_downvotes,  
 FROM  
 `bigquery-public-data.stackoverflow.votes` v  
 INNER JOIN post\_activity pa ON pa.post\_id = v.post\_id  
 WHERE  
 TRUE  
 AND pa.activity\_type = 'created'  
 AND v.creation\_date >= CAST('2021-06-01' as TIMESTAMP)   
 AND v.creation\_date <= CAST('2021-09-30' as TIMESTAMP)  
 GROUP BY  
 1,2  
)  
-- Combine all the above metrics in one CTE  
, total\_metrics\_per\_user AS (  
 SELECT  
 pm.user\_id,  
 pm.user\_name,  
 SUM(pm.posts\_created) AS total\_posts\_created,   
 SUM(pm.posts\_edited) AS total\_posts\_edited,  
 SUM(pm.answers\_created) AS total\_answers\_created,  
 SUM(pm.answers\_edited) AS total\_answers\_edited,  
 SUM(pm.questions\_created) AS total\_questions\_created,  
 SUM(pm.questions\_edited) AS total\_questions\_edited,  
 SUM(vu.total\_upvotes) AS total\_upvotes,  
 SUM(vu.total\_downvotes) AS total\_downvotes,  
 SUM(cu.total\_comments) AS total\_comments\_by\_user,  
 SUM(cp.total\_comments) AS total\_comments\_on\_post,  
 COUNT(DISTINCT pm.activity\_date) AS streak\_in\_days   
 FROM  
 user\_post\_metrics pm  
 JOIN votes\_on\_user\_post vu  
 ON pm.activity\_date = vu.activity\_date  
 AND pm.user\_id = vu.user\_id  
 JOIN comments\_on\_user\_post cp   
 ON pm.activity\_date = cp.activity\_date  
 AND pm.user\_id = cp.user\_id  
 JOIN comments\_by\_user cu  
 ON pm.activity\_date = cu.activity\_date  
 AND pm.user\_id = cu.user\_id  
 GROUP BY  
 1,2  
)  
------------------------------------------------  
---- Main Query - Calculate all derived metrics  
SELECT  
 user\_id,  
 user\_name,  
 total\_posts\_created,   
 total\_answers\_created,  
 total\_answers\_edited,  
 total\_questions\_created,  
 total\_questions\_edited,  
 total\_upvotes,  
 total\_comments\_by\_user,  
 total\_comments\_on\_post,  
 streak\_in\_days,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_posts\_created, streak\_in\_days), 0) AS NUMERIC), 1) AS posts\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_posts\_edited, streak\_in\_days), 0) AS NUMERIC), 1) AS edits\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_answers\_created, streak\_in\_days), 0) AS NUMERIC), 1) AS answers\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_questions\_created, streak\_in\_days), 0) AS NUMERIC), 1) AS questions\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_comments\_by\_user, streak\_in\_days), 0) AS NUMERIC), 1) AS comments\_by\_user\_per\_day,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_answers\_created, total\_posts\_created), 0) AS NUMERIC), 1) AS answers\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_questions\_created, total\_posts\_created), 0) AS NUMERIC), 1) AS questions\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_upvotes, total\_posts\_created), 0) AS NUMERIC), 1) AS upvotes\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_downvotes, total\_posts\_created), 0) AS NUMERIC), 1) AS downvotes\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_comments\_by\_user, total\_posts\_created), 0) AS NUMERIC), 1) AS user\_comments\_per\_post,  
 ROUND(CAST(IFNULL(SAFE\_DIVIDE(total\_comments\_on\_post, total\_posts\_created), 0) AS NUMERIC), 1) AS comments\_on\_post\_per\_post  
FROM  
 total\_metrics\_per\_user  
ORDER BY   
 total\_questions\_created DESC;

There’s one final pattern we use in the final CTE. We pre-calculate all the aggregates at the user level and then add a few more ratio-based metrics. You’ll notice that we use two functions to shape the results: CAST() is used because SQL performs integer division and for the ratios we want to show the remainder, and then ROUND() is used to round the remainder to a single decimal point.

Now that you have all these wonderful metrics you can sort it by any of the metrics. For example you can sort by questions\_per\_post to see everyone who posts mostly questions or answers\_by\_post to see those who post mostly answers. You can also create new metrics that indicate who your best users are.

Some of the best uses of this type of table are for customer segmentation or as a feature table for data science.

Could we have written this query in fewer lines using subqueries? Of course! The power of SQL is that there’s many ways to solve a problem, especially one as complex as this. But, by splitting our query up into multiple CTEs, aligning the granularity on the CTEs, chaining them carefully together we can achieve a solution that’s cleaner, easier to read and understand, and easier to maintain.

In this chapter I want to go over all the patterns we’ve learned so far one more time using simple examples