Contents

List of Figures

Introduction

This is a book about SQL Patterns. Patterns describe problems that occur over and over in our professional settings. A pattern is a like a template that you can apply to different problems. Once you learn each one, you can apply them to solve problems faster and make your code more readable.

We can illustrate this with an example. In fiction writing, authors rarely write from scratch. They use character patterns like: “antihero”, “sidekick”, “mad scientist”, “girl next door”. They also use plot patterns like romantic comedy, drama, red herring, foreshadowing, cliffhangers. This helps them write better books, movies and TV shows.

Each pattern consists of four elements:

1. The **pattern name** is a handle that describes the problem and potential solutions
2. The **problem** describes when you should apply the pattern and in what context
3. The **solution** describes the elements of the design for the solution to the problem
4. The **tradeoffs** are the consequences of applying that specific solution

Who am I

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.

Why did I write this book

I have a background in computer science. As part of the curriculum we learn how to make our code more efficient, more readable and easy to debug. As I started to write SQL, I applied many of these lessons to my own code.

When reviewing other people's code I would often spot the same mistakes. There were chunks of code that would repeat everywhere. The queries were long, complex and slow. I would often have rewrite them so I could understand what they were doing.

I looked around for a book or course that taught these patterns but couldn't find one, so I decided to write it myself.

Who this book is for

This book is for anyone who is familiar with SQL and wants to take their skills to the next level. We won't cover any of the basic syntax here so make sure you have that down pat. I expect you to already know how to join tables and do basic filtering and aggregation.

If you find that your SQL code is often inefficient and slow and you want to make it faster, this book is for you.

If you find that your SQL code is long, messy and hard to understand and you want to make it cleaner, this book is for you

If you find that your SQL code breaks easily when data changes and you want to make it more resilient, this book is for you.

What you'll learn in this book

I'm a huge fan of project-based learning. You can learn anything if you can come up with an interesting project to use it thing in. I used a project when I taught myself data science.

That's why for this book I wanted to come up with an interesting and useful data project to organize it around. I explain each pattern as I walk you through the project.

This will ensure that you learn the material better and remember it the next time you need to apply it.

We'll be working with the Stackoverflow dataset that's available in BigQuery for free. You can access it here.

BigQuery offers 1TB/month free of processing so you can complete this entire course for free. I've made sure that the queries are small and limited in scope so you won't have to worry about running out.

Using this dataset we're going to build a table which calculates reputation metrics. You can use this same type of table to calculate a customer engagement score or a customer 360 table.

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.

How this book is organized

In **Chapter 1** we introduce the project we'll be working on throughout the book. We'll make sure you have access to the dataset and can run the queries. We also get a basic understanding of the dataset by looking at the ER diagram

In **Chapter 2** we cover *Core Concepts and patterns*. These patterns act as our basic building blocks that will serve us throughout the book. I explain each one using the StackOverflow dataset since we'll be using every one in our final query.

The remaining patterns are grouped into four categories and each has its own chapter.

In **Chapter 3** we cover *Query Decomposition* patterns. We start off by learning how to decompose large queries into smaller pieces to make it easy to solve just about any complex problem. They are important to learn first because all the other patterns flow from them.

In **Chapter 4** we cover *Query Maintainability* patterns. These patterns teach you how to decompose a query and organize your code in ways that make it efficient. This will ensure your code is easier to read, understand and maintain in the future.

In **Chapter 5** we cover *Query Performance* patterns. They teach you ways to make your code more faster without sacrificing clarity. It's a delicate balance because performant code can sometimes look really messy.

In **Chapter 6** we cover *Query Robustness* patterns. They teach you ways to make your code resistant to messy data, such as duplicate rows, missing values, unexpected NULLs, etc.

The project is interwoven throughout the book. I make sure that each chapter covers some section of the final query.

In **Chapter 7** we wrap up our project and you get to see the entire query. By now you should be able to understand it and know exactly how it was designed. I recap the entire project so that you get another chance to review all the patterns. The goal here is to allow you to see all the patterns together and give you ideas on how to apply them in your day-to-day work.

In **Chapter 8** we cover a few special case patterns. We dive deeper into window functions and string manipulation, such as regular expressions and JSON parsing. Even though this is not related to our project, I wanted to make sure I enrich your vocabulary of patterns beyond what's in the project.

With that out of the way, let's dive into the project.

Chapter 1: Introducing the Project

In this chapter we're going to get into the details of the project that will help you learn the SQL Patterns. As you saw in the introduction, we're using a real-world, public dataset from StackOverflow.

StackOverflow is a popular website where users can post technical questions about any technical topic and others can post answers to these questions. They can also vote on the answers or comment on them.

Based on the quality of the answers, users 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 table that calculates reputation metrics for every user. This type of table is sometimes called a "feature table" and can be used in other applications in data science and analytics. You simply replace the user_id with a customer id or any other entity.

Since the query to build it is complex, it's the perfect tool to illustrate some of the patterns described in this book.

The schema of what it would look something like this:

column_name	type
user_id	INT64
user_name	STRING
total_posts_created	NUMERIC
total_answers_created	NUMERIC
total_answers_edited	NUMERIC
total_questions_created	NUMERIC
total_upvotes	NUMERIC
total_comments_by_user	NUMERIC
total_questions_edited	NUMERIC
streak_in_days	NUMERIC
total_comments_on_post	NUMERIC
posts_per_day	NUMERIC
edits_per_day	NUMERIC
answers_per_day	NUMERIC

```
| questions_per_day          | NUMERIC |
| comments_by_user_per_day   | NUMERIC |
| answers_per_post           | NUMERIC |
| questions_per_post         | NUMERIC |
| upvotes_per_post           | NUMERIC |
| downvotes_per_post         | NUMERIC |
| user_comments_per_post     | NUMERIC |
| comments_on_post_per_post | NUMERIC |
```

As you can see, we need to transform the source data model to a new model that has one row per `user_id` Before we do that, we need to understand the source data first.

Understanding the Data Model

Writing accurate and efficient SQL begins with understanding the data model we're starting with. This may already exist in the form of documentation and diagrams but more often than not you'll have to learn it as you go.

The original StackOverflow (SO) data model is different from the one loaded in BigQuery. When the engineers loaded it, they modified the mode somewhat. For example the SO model contains a single `Posts` table for all the different post types whereas BigQuery split each one into a separate table.

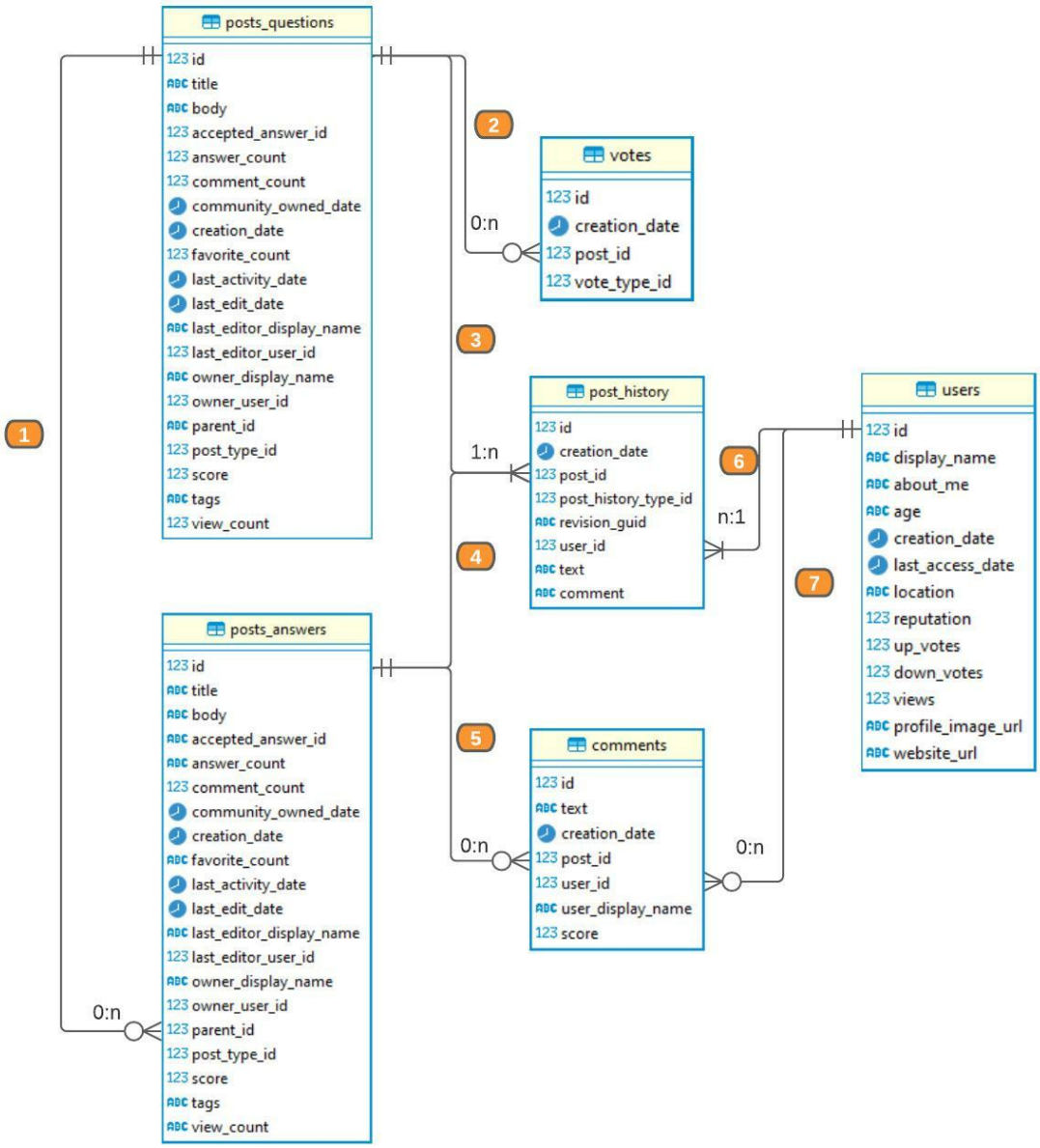

Figure 1.1 - StackOverflow ER diagram

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_%'
```

Here's the result of the query

```
|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 so I've left the other ones out: 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'
```

Here's the result of the query

```
|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. This is relation 1 in the **Figure 1.1** above.

Both post types (question and answer) have a one-to-many relationship to the `post_history`. These are relations 3 and 4 in the diagram above.

```
SELECT column_name, data_type
FROM bigquery-public-data.stackoverflow.INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'post_history'
```

Here's the result of the query

```
|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   |
```

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 perform on the site. We're only concerned with the first 6. You can see the rest of them here 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.

The `post_history` table also connects to the `users` table via the `user_id` in a one-to-many relationship shown in the diagram as number 6. A single user can perform multiple activities on a post.

In database lingo this is known as a bridge table because it connects two tables (user and posts) that 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'
```

Here's the result of the query

```
|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 shown in the diagram as number 5 and number 7, since both a user or a post could have 0 or many comments.

```
SELECT column_name, data_type
FROM bigquery-public-data.stackoverflow.INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'comments'
```

Here's the result of the query

```
|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. We'll need this 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.

```sql
SELECT column_name, data_type
FROM bigquery-public-data.stackoverflow.INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'votes'
```

Here's the result of the query

```
|column_name  |data_type|
|-------------|---------|
|id           |INT64    |
|creation_date|TIMESTAMP|
|post_id      |INT64    |
|vote_type_id |INT64    |
```

The `votes` table is connected to a post in a 0-to-many relationship shows in the diagram as number 2. In order for us to get upvotes and downvotes on a user's post, we'll need to join it with the `users` table.

Alright, now that we've familiarized ourselves with the source data model, next let's look at some core concepts.

Chapter 2: Core Concepts

In this chapter we're going to cover some of the core concepts of querying data and building tables for analysis and data science. We'll start with the most important but underrated concept in SQL, granularity.

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.

Granularity comes in two flavors: *fine grain* and *coarse grain*.

A finely grained table means a high level of detail, like one row per transaction.

A coarse grained table means a low level of detail like count of all transactions in a day.

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 finest 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 AS type_id,
    user_id,
    COUNT(*) AS total
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 |type_id|user_id |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.

What does this mean for our project?

Our final table will have a grain of one row per user. Only the `users` table has that same granularity. In order to build it we'll have to manipulate the granularity of the source tables so that's what we focus on next.

Granularity Manipulation

Now that you have a grasp of the concept of granularity the next thing to learn is how to manipulate it. What I mean by manipulation is specifically going from a fine grain to a coarser grain.

For example an e-commerce website might store each transaction it performs as a single row on a table. This gives us a very fine-grained table (i.e. a very high level of detail) If we wanted to know how much revenue you got on a given day, you have to reduce that level of detail to a single row.

This is done via aggregation.

Aggregation

Aggregation is a way of reducing the level of detail by grouping (aka rolling up) data to a coarser grain. You do that by reducing the number of columns in the output and applying GROUP BY to the remaining columns. The more columns you remove, the coarser the grain gets.

I call this *collapsing the granularity.*

This is a very common pattern of storing data in a data warehouse. You keep the table at the finest possible grain (i.e. one transaction per row) and then aggregate it up to whatever level is needed for reporting. This way you can always look up the details when you need to.

Let's look at an example.

The `post_history` table has too many rows for each `post_history_type_id` and we only need the ones representing post creation and editing. To do this, we can "collapse" them into custom categories via a CASE statement as shown below:

```
SELECT
    ph.post_id,
    ph.user_id,
    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
WHERE
    TRUE
```

```
    AND ph.post_history_type_id BETWEEN 1 AND 6
    AND ph.user_id > 0 --exclude automated processes
    AND ph.user_id IS NOT NULL --exclude deleted accounts
    AND ph.creation_date >= '2021-06-01'
    AND ph.creation_date <= '2021-09-30'
    AND ph.post_id = 69301792
GROUP BY
    1,2,3,4

post_id |user_id |activity_date          |activity_type|
--------+--------+-----------------------+-------------+
69301792|  331024|2021-09-23 21:11:44.957|edited       |
69301792|   63550|2021-09-24 10:38:36.220|edited       |
69301792|  331024|2021-09-23 10:17:11.763|created      |
69301792|  331024|2021-09-23 18:48:31.387|edited       |
69301792|14251221|2021-09-23 22:38:04.863|edited       |
69301792|  331024|2021-09-23 20:13:05.727|edited       |
```

Notice that didn't use an aggregation function like `COUNT()` or `SUM()` and that's perfectly ok since we don't need it.

You can see now how we're going to manipulate the granularity to get one row per user. We need the date in order o calculate all the date related metrics.

Date Granularity

The timestamp column `creation_date` is a rich field with both the date and time information (hour, minute, second, microsecond). Timestamp fields are special when it comes to aggregation because they have many levels of granularities built in.

Given a single timestamp, we can construct granularities for seconds, minutes, hours, days, weeks, months, quarters, years, decades, etc. We do that by using one of the many date manipulation functions like `CAST()`, `DATE_TRUNC()`, `DATE_PART()`, etc.

For example if I wanted to remove the time information, I could reduce all activities on a given date to a single row like this:

```
SELECT
    ph.post_id,
    ph.user_id,
    CAST(ph.creation_date AS 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,
    COUNT(*) AS total
FROM
    bigquery-public-data.stackoverflow.post_history ph
WHERE
    TRUE
    AND ph.post_history_type_id BETWEEN 1 AND 6
    AND ph.user_id > 0 --exclude automated processes
    AND ph.user_id IS NOT NULL --exclude deleted accounts
    AND ph.creation_date >= '2021-06-01'
    AND ph.creation_date <= '2021-09-30'
    AND ph.post_id = 69301792
GROUP BY
    1,2,3,4

post_id |user_id |activity_date|activity_type|total|
--------+--------+-------------+-------------+-----+
69301792|  331024|   2021-09-24|edited       |    3|
69301792|14251221|   2021-09-24|edited       |    1|
69301792|  331024|   2021-09-23|created      |    3|
69301792|   63550|   2021-09-24|edited       |    2|
69301792|  331024|   2021-09-23|edited       |    1|
```

In our case we only need to aggregate up to the day level, so we remove the time components by using CAST(AS DATE)

Pivoting Data

Pivoting is another form of granularity manipulation where you change the shape of aggregated data by “pivoting” rows into columns. Let’s look at the above

example and try to pivot the activity type into separate columns for `created` and `edited`

Note that the counts here don't make sense since we already know that there are 3 different `post_history_type_id` for creation and editing. This is simply shown for demonstration purposes.

This is the query:

```
SELECT
    ph.post_id,
    ph.user_id,
    CAST(ph.creation_date AS DATE) AS activity_date,
    SUM(CASE WHEN ph.post_history_type_id IN (1,2,3)
        THEN 1 ELSE 0 END) AS created,
    SUM(CASE WHEN ph.post_history_type_id IN (4,5,6)
        THEN 1 ELSE 0 END) AS edited
FROM
    bigquery-public-data.stackoverflow.post_history ph
WHERE
    TRUE
    AND ph.post_history_type_id BETWEEN 1 AND 6
    AND ph.user_id > 0 --exclude automated processes
    AND ph.user_id IS NOT NULL --exclude deleted accounts
    AND ph.creation_date >= '2021-06-01'
    AND ph.creation_date <= '2021-09-30'
    AND ph.post_id = 69301792
GROUP BY
    1,2,3
```

It will take this type of output

```
post_id |user_id |activity_date|activity_type|total|
--------+--------+-------------+-------------+-----+
69301792|  331024|   2021-09-24|edited       |    3|
69301792|14251221|   2021-09-24|edited       |    1|
69301792|  331024|   2021-09-23|created      |    3|
69301792|   63550|   2021-09-24|edited       |    2|
69301792|  331024|   2021-09-23|edited       |    1|
```

and turn it into this

```
post_id |user_id |activity_date|created|edited|
--------+--------+-------------+-------+------+
69301792|   63550|   2021-09-24|      0|     2|
69301792|  331024|   2021-09-24|      0|     3|
69301792|  331024|   2021-09-23|      3|     1|
69301792|14251221|   2021-09-24|      0|     1|
```

Pivoting is how we're going to calculate all the metrics for users, so this is an important concept to learn.

Granularity Multiplication

Joining tables is one of the most basic functions in SQL. Databases are designed to minimize redundancy of information and they do that by a process known as normalization. Joins then allow us to get all the information back in a single piece by combining these tables together.

Granularity multiplication will happen if the tables you're joining have different levels of detail for the columns being joined on. This will cause the resulting number of rows to multiply.

Let's look at an example:

The `users` table has a grain of one row per user:

```
SELECT
    id,
    display_name,
    creation_date,
    reputation
FROM bigquery-public-data.stackoverflow.users
```

```
WHERE id = 8974849;

id     |user_name|creation_date          |reputation|
-------+---------+-----------------------+----------+
8974849|neutrino |2017-11-20 18:16:46.653|       790|
```

Whereas the `post_history` table has multiple rows for the same user:

```
SELECT
    id,
    creation_date,
    post_id,
    post_history_type_id AS type_id,
    user_id
FROM
    bigquery-public-data.stackoverflow.post_history ph
WHERE
    TRUE
    AND ph.creation_date >= '2021-06-01'
    AND ph.creation_date <= '2021-09-30'
    AND ph.user_id = 8974849;


id       |creation_date      |post_id |type_id|user_id|
---------+-------------------+--------+-------+-------+
250199272|2021-07-14 00:54:58|68372251|      2|8974849|
250199273|2021-07-14 00:54:58|68372251|      1|8974849|
250199274|2021-07-14 00:54:58|68372251|      3|8974849|
250263915|2021-07-15 00:01:07|68387743|      2|8974849|
250263916|2021-07-15 00:01:07|68387743|      1|8974849|
250263917|2021-07-15 00:01:07|68387743|      3|8974849|
250316277|2021-07-15 16:32:44|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 AS 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 >= '2021-06-01'
    AND ph.creation_date <= '2021-09-30'
    AND ph.user_id = 8974849;

post_id |user_id|user_name|activity_date      |type_id|
--------+-------+---------+-------------------+-------+
68078326|8974849|neutrino |2021-06-22 02:03:45|      2|
68078326|8974849|neutrino |2021-06-22 02:03:45|      1|
68078326|8974849|neutrino |2021-06-22 02:03:45|      3|
68273785|8974849|neutrino |2021-07-06 11:56:05|      2|
68273785|8974849|neutrino |2021-07-06 11:56:05|      1|
68273785|8974849|neutrino |2021-07-06 11:56:05|      3|
68277148|8974849|neutrino |2021-07-06 16:40:53|      2|
68277148|8974849|neutrino |2021-07-06 16:40:53|      1|
68277148|8974849|neutrino |2021-07-06 16:40:53|      3|
68273785|8974849|neutrino |2021-07-06 12:02:11|      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 and each row would repeat twice.

Accidental INNER JOIN

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 JOIN`and 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;
```

This query gives us the 12 missing users

Starting with a LEFT JOIN

Since we're on the subject of LEFT JOINS, one of my most used rules of thumb is to always use a `LEFT JOIN` when I'm not sure if one table is a subset of the other. For example in the query above, there's definitely users that have a valid `user_id` in the `users` table but have never had any activity.

This often happens in the real world when data is deleted from a table and there's no foreign key constraints to ensure referential integrity (i.e. the database ensures you can't delete a row if it's referenced in another table. These types of constraints don't exist in data warehouses hence my general rule of thumb of always starting with a `LEFT JOIN`

Now that we have covered the basic concepts, it's time to dive into the patterns.

Chapter 4: Query Decomposition

In this chapter we're going to learn one of the most important patterns in SQL. This pattern will help you solve very complex queries by systematically decomposing them into simpler ones. Before we go that far, first we need to talk about CTEs

Common Table Expressions (CTEs)

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. They act like subqueries but are easier to understand and use.

CTEs allow you to break down complex queries into simpler, smaller self-contained modules. By connecting them together we can solve just about any complex query. One of the key requirements is that these CTEs should not try to do too much. They should have a single purpose or responsibility so you can write, test and debug them independently.

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 so you chain 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>
```

We'll talk about chaining in a little bit.

When you use CTEs you can read a query top to bottom and easily understand what's going on. When you use sub-queries it's a lot harder to trace the logic and figure out which column is defined where and what scope it has. You have to read the innermost subquery first and then remember each of the definitions.

Query Decomposition

Getting our user data from the current form to the final form of one row per user is not something that can be done in a single step. Well you probably could hack something together that works but that will not be very easy to maintain. It's a complex query.

In order to solve it, we need to decompose (break down) our complex query into smaller, easier to write pieces. Here's how to think about it:

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 have separate tables for these activities, so our first step is to aggregate the data from each of the tables to the `user_id` and `aciticity_date` granularity and put each one on its own CTE.

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

Sub-problem 1

Calculate user metrics for post types and post activity types.

To get there we first have to manipulate the granularity of the `post_history` table so we have one row per `user_id` per `post_id` per `activity_type` per `activity_date`.

That would look like this:

```
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 >= '2021-06-01'
        AND ph.creation_date <= '2021-09-30'
    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       |

Table 3.1
```

We then join this with the `posts_questions` and `post_answers` on `post_id`. That would look like this:

```sql
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 >= '2021-06-01'
        AND ph.creation_date <= '2021-09-30'
    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 >= '2021-06-01'
        AND creation_date <= '2021-09-30'
    UNION ALL
    SELECT
        id AS post_id,
        'answer' AS post_type,
    FROM
        bigquery-public-data.stackoverflow.posts_answers
    WHERE
        TRUE
        AND creation_date >= '2021-06-01'
        AND creation_date <= '2021-09-30'
 )
SELECT
    pa.user_id,
    CAST(pa.activity_date AS DATE) AS activity_date,
```

```
    pa.activity_type,
    pt.post_type
FROM
    post_activity pa
    JOIN post_types pt ON pa.post_id = pt.post_id
WHERE user_id = 16366214
ORDER BY activity_date;

user_id |date      |activity_type|post_type|
--------+----------+-------------+---------+
16366214|2021-07-18|created      |question |
16366214|2021-07-20|created      |question |
16366214|2021-07-25|edited       |question |
16366214|2021-07-25|created      |answer   |
16366214|2021-07-01|created      |question |
16366214|2021-07-25|edited       |question |

Table 3.2
```

The final result should look like this:

```
user_id |date      |question_created|answer_created|question_edited|answer_edited|
--------+----------+----------------+--------------+---------------+-------------+
16366214|2021-07-25|               0|             1|              2|            0|
16366214|2021-07-18|               1|             0|              0|            0|
16366214|2021-07-01|               1|             0|              0|            0|
16366214|2021-07-20|               1|             0|              0|            0|

Table 3.1
```

How do we go from *Table 3.2* to *Table 3.1*? If you recall from **Chapter 2**, we can use aggregation and pivoting:

```
--code snippet will not actually run
SELECT
    user_id,
    CAST(pa.activity_date AS DATE) AS activity_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_activity pa
     JOIN post_types pt ON pt.post_id = pa.post_id
WHERE user_id = 16366214
GROUP BY 1,2
```

This query will not run and is only shown for demonstration purposes.

Sub-problem 2

Calculate comments metrics. There are two types of comments: 1. Comments by a user (on one or many posts) 2. Comments on a user's post (by other users)

The query and final result should look like this:

```
--code snippet will not actually run
, 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 >= '2021-06-01'
        AND c.creation_date <= '2021-09-30'
    GROUP BY
        1,2
)
, 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 >= '2021-06-01'
        AND creation_date <= '2021-09-30'
    GROUP BY
        1,2
)
SELECT
    c1.user_id,
    c1.activity_date,
    c1.total_comments AS comments_by_user,
    c2.total_comments AS comments_on_user_post
FROM comments_by_user c1
     LEFT OUTER JOIN comments_on_user_post c2
        ON c1.user_id = c2.user_id
        AND c1.activity_date = c2.activity_date
WHERE
    c1.user_id = 16366214

user_id |activity_date|comments_by_user|comments_on_user_post|
--------+-------------+----------------+---------------------+
16366214|   2021-07-19|               1|                    2|
16366214|   2021-07-21|               1|                 NULL|
16366214|   2021-07-26|               3|                    4|

Table 3.3
```

Sub-problem 3

Calculate votes metrics. There are two types of votes: 1. Upvotes on a user's post 2. Downvotes on a user's post

The query and final result should look like this:

```
--code snippet will not actually run
, 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 >= '2021-06-01'
        AND v.creation_date <= '2021-09-30'
    GROUP BY
        1,2
)
SELECT
    v.user_id,
    v.activity_date,
    v.total_upvotes,
    v.total_downvotes
FROM
    votes_on_user_post v
WHERE
    v.user_id = 16366214

user_id |activity_date|total_upvotes|total_downvotes|
--------+-------------+-------------+---------------+
16366214|   2021-07-26|            2|              0|
16366214|   2021-07-06|            0|              1|
16366214|   2021-07-07|            0|              0|

Table 3.4
```

By now you should start to see very clearly how the final result is constructed. All we have to do is take the 3 results from the sub-problems and join them together on `user_id` and `activity_date` This will allow us to have a single table with a granularity of one row per user and all the metrics aggregated on the day level like this:

```
--code snippet will not actually run
SELECT
    pm.user_id,
    pm.user_name,
    CAST(SUM(pm.posts_created) AS NUMERIC) AS total_posts_created,
    CAST(SUM(pm.posts_edited) AS NUMERIC)  AS total_posts_edited,
    CAST(SUM(pm.answers_created) AS NUMERIC) AS total_answers_created,
    CAST(SUM(pm.answers_edited) AS NUMERIC)  AS total_answers_edited,
    CAST(SUM(pm.questions_created) AS NUMERIC) AS total_questions_created,
    CAST(SUM(pm.questions_edited) AS NUMERIC)  AS total_questions_edited,
    CAST(SUM(vu.total_upvotes) AS NUMERIC)   AS total_upvotes,
    CAST(SUM(vu.total_downvotes) AS NUMERIC) AS total_downvotes,
    CAST(SUM(cu.total_comments) AS NUMERIC)  AS total_comments_by_user,
    CAST(SUM(cp.total_comments) AS NUMERIC)  AS total_comments_on_post,
    CAST(COUNT(DISTINCT pm.activity_date) AS NUMERIC) 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
```

In the next chapter we'll extend these patterns and see how they help us with query maintainability.

Chapter 5: Query Maintainability

In this chapter we're going to extend the pattern of decomposition into the realm of query maintainability. Breaking down large queries into small pieces doesn't only make them easier to read, write and understand, it also makes them easier to maintain.

The Reusability Principle

We start off with a very important principle that rarely gets talked about in SQL. When you're designing a query and breaking it up into CTEs, there is one principle to keep in mind. The CTEs should be constructed in such a way that they can be reused if needed later. This principle makes code easier to maintain and compact.

Let's take a look at the example from the previous chapter:

```
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 >= '2021-06-01'
        AND ph.creation_date <= '2021-09-30'
    GROUP BY
        1,2,3,4,5
)
```

This CTE performs several operations like aggregation, to decrease granularity of the underlying data, and filtering. Its main purpose is to get a mapping between `user_id` and `post_id` at the right level of granularity so it can be used later.

What's great about this CTE is that we can use it both for generating user metrics as shown here:

```
--code snippet will not actually run
SELECT
    user_id,
    CAST(pa.activity_date AS DATE) AS activity_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_activity pa
     JOIN post_types pt ON pt.post_id = pa.post_id
WHERE user_id = 16366214
GROUP BY 1,2
```

and to join with comments and votes to user level data via the `post_id`

```
--code snippet will not actually run
, 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 >= '2021-06-01'
        AND c.creation_date <= '2021-09-30'
    GROUP BY
        1,2
)
, 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 >= '2021-06-01'
        AND v.creation_date <= '2021-09-30'
    GROUP BY
        1,2
)
```

This is at the heart of well-designed CTE. Notice here that we're being very careful about granularity multiplication! If we simply joined with `post_activity` on post_id without specifying the `activity_type` we'd get at least two times the number of rows. Since a post can only be created once, we're pretty safe in getting a single row by filtering.

The DRY Principle

In the previous section we saw how we can decompose a large complex query into multiple smaller components. The main benefit for doing this is that it makes the queries more readable. In that same vein, the DRY (Don't Repeat Yourself) 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, you should put that code in a CTE and reference that CTE where it's needed.

To illustrate let's 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 >= '2021-06-01'
        AND ph.creation_date <= '2021-09-30'
    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 >= '2021-06-01'
        AND creation_date <= '2021-09-30'
)
, 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 >= '2021-06-01'
        AND creation_date <= '2021-09-30'
)
SELECT
    user_id,
    CAST(activity_date AS DATE) AS activity_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
    (SELECT * FROM questions
     UNION ALL
     SELECT * FROM answers) AS p
WHERE
    user_id = 16366214
GROUP BY 1,2;
```

This query will get you the same results as table 3.1 in the previous chapter but as you can see the `eustions` and `answers` CTEs both have almost identical code. Imagine if you had to do this for all question types. You'd be copying and pasting a lot of code. Also, the subquery that handles the UNION is not ideal. I'm not a fan of subqueries

Since both questions and answers tables have the exact same schema, a great way to deal with the above problem is by appending their rows using the `UNION` operator like this:

```
SELECT
    id AS post_id,
    'question' AS post_type,
FROM
    bigquery-public-data.stackoverflow.posts_questions
WHERE
    TRUE
    AND creation_date >= '2021-06-01'
    AND creation_date <= '2021-09-30'

UNION ALL

SELECT
    id AS post_id,
    'answer' AS post_type,
FROM
    bigquery-public-data.stackoverflow.posts_answers
WHERE
    TRUE
    AND creation_date >= '2021-06-01'
    AND creation_date <= '2021-09-30'
```

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. This is slower because of the extra operations to find and 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

You can achieve the first requirement by using `SELECT` to choose only the columns that match across multiple tables or if you know the tables have the same exact schema. Note that when you union tables with different schemas, you have to line up all the columns in the right order. This is useful when two tables have the same column named differently.

For example:

```
SELECT
    col1 as column_name
FROM
    table1

UNION ALL

SELECT
    col2 as column_name
FROM
    table2
```

As a rule of thumb, when you append 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

One of the benefits of building reusable CTEs is that if you find yourself copying and pasting the same CTE in multiple places, you can turn it into a view and store it in the database.

Views are great for encapsulating business logic that applies to many queries. They're also used in security applications

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.

What could be made into a view in our specific query?

I think the `post_types` CTE would be a good candidate. That way whenever you have to combine all the post types you don't have to use that CTE everywhere.

```sql
CREATE OR REPLACE VIEW v_post_types AS
    SELECT
        id AS post_id,
        'question' AS post_type,
    FROM
        bigquery-public-data.stackoverflow.posts_questions
    WHERE
        TRUE
        AND creation_date >= '2021-06-01'
        AND creation_date <= '2021-09-30'
    UNION ALL
    SELECT
        id AS post_id,
        'answer' AS post_type,
    FROM
        bigquery-public-data.stackoverflow.posts_answers
    WHERE
        TRUE
        AND creation_date >= '2021-06-01'
        AND creation_date <= '2021-09-30';
```

*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.

So far we've talked about how to optimize queries so they're easy to read, write, understand and maintain. In the next chapter we tackle patterns regarding query performance.

Chapter 6: Query Performance

In this chapter we're going to talk about query performance, aka how to make your queries run faster. Why do we care about making queries run faster? Faster queries get you results faster, while consuming fewer resources, making them cheaper on modern data warehouses.

This chapter isn't only about speed. You can make your queries run really fast with a few clever hacks, but that might make your code unreadable and unmaintainable. We need to strike a balance between the performance, accuracy and maintainability.

Reducing Rows

The most important pattern that improves query performance is reducing data as much as possible before you join it.

What does that mean?

So far we've learned that decomposing (aka breaking down) a query via CTEs is the best way to tackle complex queries. But what kinds of operations should your perform in the CTE? We've already seen aggregation and calculation of metrics that can be used later. One of the best uses for CTEs is filtering.

You might have noticed this little snipped in every CTE:

```
WHERE
    TRUE
    AND creation_date >= '2021-06-01'
    AND creation_date <= '2021-09-30'
```

What we're doing here is filtering each table to only 90 days so we can both to keep costs down and make the query faster. This is what I mean by reducing the dataset before joining.

In this case we actually only want to work with 90 days worth of data. if we needed all historical data, we couldn't reduce it beforehand and we'd have to work with the full table.

Let's look at some implications of this pattern.

Don't use functions in WHERE

In case you didn't know, you can put anything in the where clause. You already know about filtering on dates, numbers and strings of course but you can also filter calculations, functions, CASE statements, etc.

Here's a rule of thumb when it comes to making queries faster. Always try to make the wHERE clause simple. Compare a column to another column or to a fixed value and avoid using functions.

When you use compare a column to a fixed value or to another column, the query optimizer can filter down to the relevant rows much faster. When you use a function or a complicated formula, the optimizer needs to scan the entire table to do the filtering. This is negligible for small tables but when dealing with millions of rows query performance will suffer.

Let's see some examples:

The `tags` column in both questions and answers is a collection of strings separated by | character as you see here:

```
SELECT
    q.id AS post_id,
    q.creation_date,
    q.tags
FROM
    bigquery-public-data.stackoverflow.posts_questions q
WHERE
    TRUE
    AND creation_date >= '2021-06-01'
    AND creation_date <= '2021-09-30'
LIMIT 10

post_id |creation_date          |tags                                                |
--------+-----------------------+----------------------------------------------------+
67781287|2021-05-31 20:00:59.663|python|selenium|screen-scraping|thesaurus       |
67781291|2021-05-31 20:01:48.593|python                                              |
67781295|2021-05-31 20:02:38.043|html|css|bootstrap-4                               |
67781298|2021-05-31 20:03:01.413|xpages|lotus-domino                                 |
67781300|2021-05-31 20:03:12.987|bash|awk|sed                                       |
67781306|2021-05-31 20:03:54.117|c                                                   |
67781310|2021-05-31 20:04:33.980|php|html|navbar                                    |
67781313|2021-05-31 20:04:57.957|java|spring|dependencies                          |
67781314|2021-05-31 20:05:12.723|python|qml|kde                                     |
67781315|2021-05-31 20:05:15.703|javascript|reactjs|redux|react-router|components|
```

The tags pertain to the list of topics or subjects that a post is about. One of the tricky things about storing tags like this is that you don't have to worry about the order in which they appear. There's no categorization system here. A tag can appear anywhere in the string.

How would you go about filtering all the posts that are about SQL? Since the tag `|sql|` can appear anywhere in the string, you'll need a way to search the entire string. One way to do that is to use the `INSTR()` function like this:

```
SELECT
    q.id AS post_id,
    q.creation_date,
    q.tags
FROM
    bigquery-public-data.stackoverflow.posts_questions q
```

```
WHERE
    TRUE
    AND creation_date >= '2021-06-01'
    AND creation_date <= '2021-09-30'
    AND INSTR(tags, "|sql|") > 0
LIMIT 10

post_id |creation_date          |tags                           |
--------+-----------------------+-------------------------------+
67941534|2021-06-11 13:55:08.693|mysql|sql|database|datatable   |
67810767|2021-06-02 14:40:44.110|mysql|sql|sqlite              |
67814136|2021-06-02 20:55:41.193|mysql|sql|where-clause        |
67849335|2021-06-05 07:58:09.493|php|mysql|sql|double|var-dump  |
68074104|2021-06-21 16:08:25.487|php|sql|postgresql|mdb2       |
67920305|2021-06-10 07:32:21.393|python|sql|pandas|pyodbc      |
68015950|2021-06-17 04:47:27.713|c#|sql|.net|forms|easy-modbus  |
68058413|2021-06-20 13:28:00.980|java|sql|spring|kotlin|jpa    |
68060567|2021-06-20 18:39:04.150|mysql|sql|ruby-on-rails|graphql|
68103046|2021-06-23 11:40:56.087|php|mysql|sql|stored-procedures|
```

This should be pretty simple to understand. We're searching for the sub-string `|sql|` anywhere in the `tags` column. The `INSTR()` searches for a sub-string within a string and returns the position of the character where it's found. Since we don't care about that, we only care that it's found our condition is > 0.

This is a very typical example of using functions in the `WHERE` clause. This particular query might be fast but in general this pattern is not advised. So what can you do instead?

Use the `LIKE` keyword to look for patterns. Many query optimizers perform much better with `LIKE` then with using a function:

```
SELECT
    q.id AS post_id,
    q.creation_date,
    q.tags
FROM
    bigquery-public-data.stackoverflow.posts_questions q
```

```
WHERE
    TRUE
    AND creation_date >= '2021-06-01'
    AND creation_date <= '2021-09-30'
    AND tags LIKE "%|sql|%"
LIMIT 10

post_id |creation_date          |tags                           |
--------+-----------------------+-------------------------------+
67941534|2021-06-11 13:55:08.693|mysql|sql|database|datatable   |
67810767|2021-06-02 14:40:44.110|mysql|sql|sqlite              |
67814136|2021-06-02 20:55:41.193|mysql|sql|where-clause        |
67849335|2021-06-05 07:58:09.493|php|mysql|sql|double|var-dump  |
68074104|2021-06-21 16:08:25.487|php|sql|postgresql|mdb2       |
67920305|2021-06-10 07:32:21.393|python|sql|pandas|pyodbc      |
68015950|2021-06-17 04:47:27.713|c#|sql|.net|forms|easy-modbus  |
68058413|2021-06-20 13:28:00.980|java|sql|spring|kotlin|jpa     |
68060567|2021-06-20 18:39:04.150|mysql|sql|ruby-on-rails|graphql|
68103046|2021-06-23 11:40:56.087|php|mysql|sql|stored-procedures|
```

Reducing Columns

Almost every book or course will tell you to start exploring a table by doing:

```
SELECT *
FROM bigquery-public-data.stackoverflow.posts_questions
LIMIT 10
```

This may be ok in a traditional RDBMS, but with modern data warehouses things are different. Because they store data in columns vs rows `SELECT *` will scan the entire table and your query will be slower.

In addition to that, in BigQuery you get charged by how many bytes of a table you scan. Doing a `SELECT *` on a very large table will be just as expensive if you return 10 rows or 10 million rows.

By selecting only the columns you need you ensure that your query is as efficient as it needs to be.

Here's an example you've seen before. In the `post_activity` CTE we select only the `id` column which is the only one we need to join with `post_activity` on. The `post_type` is a static value which is negligible when it comes to performance.

```
-- code snippet will not run
,post_types AS (
    SELECT
        id AS post_id,
        'question' AS post_type,
    FROM
        bigquery-public-data.stackoverflow.posts_questions
    WHERE
        TRUE
        AND creation_date >= '2021-06-01'
        AND creation_date <= '2021-09-30'
    UNION ALL
    SELECT
        id AS post_id,
        'answer' AS post_type,
    FROM
        bigquery-public-data.stackoverflow.posts_answers
    WHERE
        TRUE
        AND creation_date >= '2021-06-01'
        AND creation_date <= '2021-09-30'
 )
```

Premature Ordering

As a rule of thumb you should leave ordering until the very end, if it is at all necessary. Sorting data is generally an expensive operation in databases so it should be reserved for when you really need it. Window functions for example

sometimes necessitate ordering. We'll cover them in chapter 8.

If you know that your data will be used by a business intelligence tool like Looker or Tableau then you should leave the ordering up to the tool itself so the user can sort data any way they see fit.

For example, the following is unnecessary and slows down performance because the query is inside a CTE. You don't need to sort your data yet.

```
-- code snippet will not run
, 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 >= '2021-06-01'
        AND v.creation_date <= '2021-09-30'
    GROUP BY
        1,2
    ORDER BY
        v.creation_date
)
```

Bounded Time Windows

Many analytical queries need to go back a certain number of days/weeks/months in order to calculate trend-based metrics. These are known as "lookback windows." You specify a period of time to look back (e.g. 30 days ago, 90 days ago, a week ago, etc) and you aggregate data to today's date.

If you don't specify a bounded or sliding time window, your query performance will get worse over time as more data is considered.

What makes this problem hard to detect is that initially your query could be very fast at first. Since there isn't a lot of data in the table performance doesn't suffer. As data gets added to the table however your query will start to get slower.

Let's take the above example to illustrate. In this query I'm specifying a fixed time window, from Jun 1st to Sep 30th. No matter how big the table gets, my query performance will remain the same.

```
-- code snippet will not run
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 >= '2021-06-01'
    AND v.creation_date <= '2021-09-30'
GROUP BY
    1,2
ORDER BY
    v.creation_date
```

A more common pattern is the sliding time window where the period under consideration is always fixed but it's dynamically based on when it's being run.

```
-- code snippet will not run
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 >= DATE_ADD(CURRENT_DATE(), INTERVAL -90 DAY)
GROUP BY
    1,2
ORDER BY
    v.creation_date
```

As you can see, the query is always looking at the last 90 days worth of data but the specific days it's looking into are not fixed. If you run it today, the results will be different from yesterday.

Let's now change this slightly and see what happens:

```
-- code snippet will not run
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('2021-12-15' as TIMESTAMP)
GROUP BY
    1,2
ORDER BY
    v.creation_date
```

This query is also looking at the last 90 days worth of data but unlike the query above, the lower boundary is fixed. This query's performance will get worse over time.

That wraps up query performance. There's a lot more to learn about improving query performance but that's not the purpose of this book. In the next chapter we'll cover how to make your queries robust against unexpected changes in the underlying data.

Chapter 7: Query Robustness

In this chapter we're going to talk about how to make your queries robust to most data problems you'll encounter. Spend enough time working with real world data and you'll eventually get burned by one of these unexpected data issues.

Robustness means that your query will not break if the underlying data changes in unpredictable ways.

Here are some of the ways that data can change:

1. New columns are added that have NULL values for past data
2. Existing columns that didn't have NULLs before now contain NULLs
3. Columns that contained numbers or dates stored as strings now contain other values
4. The formatting of dates or numbers gets messed up and the type conversion fails.
5. The denominator in a ratio calculation becomes zero

Ideally these things should not happen but in reality they happen more often than we'd like. The purpose of this chapter is to teach you how to anticipate these problems before they happen and

This is why I like to call this chapter **Defense Against Dirty Data**

We'll break these patterns down into two three groups: 1. Dealing with formatting issues 3. Dealing with NULLs 2. Dealing with division by zero

Dealing with formatting issues

SQL supports 3 primitive data types, strings, numbers and dates. They allow for mathematical operations with numbers and calendar operations with dates. Oftentimes you might see numbers and dates stored as strings.

This makes it super easy to load data from text files into tables without worrying about formatting. However in order to operate on actual dates and numbers, you need to convert the strings to the native SQL type for number or date.

The standard function for converting data in SQL is `CAST()` Some other database implementations, like SQL Server, also use their own custom function called `CONVERT()`. We will 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('2021-13-01' 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 thing can 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 Bad Data

One of the easiest ways to deal with formatting issues when converting data is to simply ignore bad formatting. What this means is we simply skip the malformed rows and don't deal with them at all. This works great in cases when the error is unfixable or occurs very rarely. So if a few rows out of 10 million are malformed and can't be fixed we can skip them

However the `CAST()` function will fail if it encounters an issue, as we just saw, and we want our query to be robust. To deal with this problem some databases introduce "safe" casting functions like `SAFE_CAST()` in BigQuery or `TRY_CAST()` in SQL Server. Not all servers provide this function though.

These functions will not fail when the formatting is unexpected but return `NULL` instead which then can be handled by using `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 COALESCE(SAFE_CAST('2021-' as INTEGER), 0) AS num;

num|
---+
  0|
```

Force Formatting

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 when you need to look for repeating patterns in the incorrect data and force the formatting.

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 different types of irregularities 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 formatting for all the dates here so we'll have to split this up and apply the pattern separately using the CASE statement:

```
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;
```

You can repeat this pattern as many times as you want to handle each case.

Here's an example using numbers

```
WITH weights AS (
    SELECT '32.5lb' AS wt
    UNION ALL
    SELECT '45.2lb' AS wt
```

```
    UNION ALL
    SELECT '53.1lb' AS wt
    UNION ALL
    SELECT '77kg' AS wt
    UNION ALL
    SELECT '68kg' AS wt
)
SELECT
    CAST(CASE WHEN wt LIKE '%lb' THEN SUBSTRING(wt, 1, INSTR(wt, 'lb')-1)
              WHEN wt LIKE '%kg' THEN SUBSTRING(wt, 1, INSTR(wt, 'kg')-1)
         END AS DECIMAL) AS weight,
    CASE WHEN wt LIKE '%lb' THEN 'LB'
         WHEN wt LIKE '%kg' THEN 'KG'
    END AS unit
FROM weights
```

I'm using the SUBSTRING() function again to extract parts of a string, but this time I add the function INSTR() which searches for a string within another string and returns the first occurrence of it or 0 if not found.

Dealing with NULLs

As a rule, you should always assume any column can be NULL at any point in time so it's a good idea to provide a default value for that column as part of your SELECT. This way you make sure that even if your data becomes NULL your query will not fail.

NULLs in SQL represent unknown values. While the data may appear to be blank or empty in the results, 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 whenever you perform any type of calculation with NULL like adding or subtracting, multiplying or dividing. Doing any operation with an unknown value is still unknown.

Since you cannot compare to NULL using the equals sign (=) SQL deals with NULLs using the `IS` keyword. `IS NULL` literally means is unknown. To replace NULLs with a default value when you're doing conversions, you use `COALESCE()` which takes a comma-separated list of values and returns the first non-null value.

So in order to protect against unexpected NULLs it's often a good idea for your production queries to wrap `COALESCE()` 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 COALESCE(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 we saw earlier but implemented using "defensive coding" where we replace malformed data with a fixed value of `1900-01-01`. This

protects our query from failing and later we can investigate why the data was junk.

Dealing with division by zero

Whenever you need to calculate ratios you always have to worry about division by zero. Your query might work when you first write it, but if the denominator ever becomes zero your query will fail.

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 some cases but it also will filter the entire dataset causing counts to be wrong. 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 works but is 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 simply deal with NULL values using `COALESCE()`

```
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 `DIV0()` which automatically returns `0` if there's a division by zero error.

Comparing 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|
```

If you ever have to join on an email column these functions are absolutely essential. It's best to combine them just to be sure:

```
SELECT TRIM(LOWER('String')) = TRIM(LOWER(' string')) AS test;

test|
----+
true|
```

That wraps up our chapter on query robustness. In the next chapter we get to see the entire query for user engagement. It's also a great opportunity to review what we've learned so far.

Chapter 7: Finishing the Project

In this chapter we wrap up our query and go over it one more time highlighting the various patterns we've learned so far. This is a good opportunity to test yourself and see what you've learned. Analyze the query and see what patterns you recognize.

So here's the whole query

```sql
 -- 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 >= '2021-06-01'
        AND ph.creation_date <= '2021-09-30'
    GROUP BY
        1,2,3,4,5
)
-- Get the post types we care about questions and answers only and combine them in one CTE
,post_types AS (
    SELECT
        id AS post_id,
        'question' AS post_type,
    FROM
        bigquery-public-data.stackoverflow.posts_questions
    WHERE
        TRUE
        AND creation_date >= '2021-06-01'
        AND creation_date <= '2021-09-30'
    UNION ALL
    SELECT
        id AS post_id,
        'answer' AS post_type,
    FROM
        bigquery-public-data.stackoverflow.posts_answers
    WHERE
        TRUE
        AND creation_date >= '2021-06-01'
        AND creation_date <= '2021-09-30'
 )
 -- Finally calculate the post metrics
```

```
, 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
)
, 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 >= '2021-06-01'
        AND creation_date <= '2021-09-30'
    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 >= '2021-06-01'
        AND c.creation_date <= '2021-09-30'
    GROUP BY
```

```
        1,2
)
, 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 >= '2021-06-01'
        AND v.creation_date <= '2021-09-30'
    GROUP BY
        1,2
)
, total_metrics_per_user AS (
    SELECT
        pm.user_id,
        pm.user_name,
        CAST(SUM(pm.posts_created) AS NUMERIC)     AS total_posts_created,
        CAST(SUM(pm.posts_edited) AS NUMERIC)      AS total_posts_edited,
        CAST(SUM(pm.answers_created) AS NUMERIC)   AS total_answers_created,
        CAST(SUM(pm.answers_edited) AS NUMERIC)    AS total_answers_edited,
        CAST(SUM(pm.questions_created) AS NUMERIC) AS total_questions_created,
        CAST(SUM(pm.questions_edited) AS NUMERIC)  AS total_questions_edited,
        CAST(SUM(vu.total_upvotes) AS NUMERIC)     AS total_upvotes,
        CAST(SUM(vu.total_downvotes) AS NUMERIC)   AS total_downvotes,
        CAST(SUM(cu.total_comments) AS NUMERIC)    AS total_comments_by_user,
        CAST(SUM(cp.total_comments) AS NUMERIC)    AS total_comments_on_post,
        CAST(COUNT(DISTINCT pm.activity_date) AS NUMERIC) 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
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(IFNULL(SAFE_DIVIDE(
        total_posts_edited, streak_in_days), 0), 1) AS edits_per_day,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_answers_created, streak_in_days), 0), 1) AS answers_per_day,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_questions_created, streak_in_days), 0), 1) AS questions_per_day,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_comments_by_user, streak_in_days), 0), 1) AS comments_by_user_per_day,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_answers_created, total_posts_created), 0), 1) AS answers_per_post,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_questions_created, total_posts_created), 0), 1) AS questions_per_post,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_upvotes, total_posts_created), 0), 1) AS upvotes_per_post,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_downvotes, total_posts_created), 0), 1) AS downvotes_per_post,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_comments_by_user, total_posts_created), 0), 1)  AS user_comments_per_post,
    ROUND(IFNULL(SAFE_DIVIDE(
        total_comments_on_post, total_posts_created), 0), 1)  AS comments_on_post_per_post
FROM
    total_metrics_per_user
ORDER BY
    total_posts_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 the results by any of them to see different types of users. 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.