**Subquery Formatting**

When writing **Subqueries**, it is easy for your query to look incredibly complex. In order to assist your reader, which is often just yourself at a future date, formatting SQL will help with understanding your code.

The important thing to remember when using subqueries is to provide some way for the reader to easily determine which parts of the query will be executed together. Most people do this by indenting the subquery in some way - you saw this with the solution blocks in the previous concept.

The examples in this class are indented quite far—all the way to the parentheses. This isn’t practical if you nest many subqueries, but in general, be thinking about how to write your queries in a readable way. Examples of the same query written multiple different ways is provided below. You will see that some are much easier to read than others.

**Badly Formatted Queries**

Though these poorly formatted examples will execute the same way as the well formatted examples, they just aren't very friendly for understanding what is happening!

Here is the first, where it is impossible to decipher what is going on:

**SELECT** \* **FROM** (**SELECT** DATE\_TRUNC('day',occurred\_at) **AS** **day**, channel, **COUNT**(\*) **as** **events** **FROM** web\_events **GROUP** **BY** 1,2 **ORDER** **BY** 3 **DESC**) sub;

This second version, which includes some helpful line breaks, is easier to read than that previous version, but it is still not as easy to read as the queries in the **Well Formatted Query** section.

**SELECT** \*

**FROM** (

**SELECT** DATE\_TRUNC('day',occurred\_at) **AS** **day**,

channel, **COUNT**(\*) **as** **events**

**FROM** web\_events

**GROUP** **BY** 1,2

**ORDER** **BY** 3 **DESC**) sub;

**Well Formatted Query**

Now for a well formatted example, you can see the table we are pulling from much easier than in the previous queries.

**SELECT** \*

**FROM** (**SELECT** DATE\_TRUNC('day',occurred\_at) **AS** **day**,

channel, **COUNT**(\*) **as** **events**

**FROM** web\_events

**GROUP** **BY** 1,2

**ORDER** **BY** 3 **DESC**) sub;

Additionally, if we have a **GROUP BY**, **ORDER BY**, **WHERE**, **HAVING**, or any other statement following our subquery, we would then indent it at the same level as our outer query.

The query below is similar to the above, but it is applying additional statements to the outer query, so you can see there are **GROUP BY** and **ORDER BY** statements used on the output are not tabbed. The inner query **GROUP BY** and **ORDER BY** statements are indented to match the inner table.

**SELECT** \*

**FROM** (**SELECT** DATE\_TRUNC('day',occurred\_at) **AS** **day**,

channel, **COUNT**(\*) **as** **events**

**FROM** web\_events

**GROUP** **BY** 1,2

**ORDER** **BY** 3 **DESC**) sub

**GROUP** **BY** **day**, channel, **events**

**ORDER** **BY** 2 **DESC**;

### Subqueries Part II

In the first subquery you wrote, you created a table that you could then query again in the **FROM** statement. However, if you are only returning a single value, you might use that value in a logical statement like **WHERE**, **HAVING**, or even **SELECT** - the value could be nested within a **CASE** statement.

On the next concept, we will work through this example, and then you will get some practice on answering some questions on your own.

### Expert Tip

Note that you should not include an alias when you write a subquery in a conditional statement. This is because the subquery is treated as an individual value (or set of values in the **IN** case) rather than as a table.

Also, notice the query here compared a single value. If we returned an entire column **IN** would need to be used to perform a logical argument. If we are returning an entire table, then we must use an **ALIAS** for the table, and perform additional logic on the entire table.

The **WITH** statement is often called a **Common Table Expression** or **CTE**. Though these expressions serve the exact same purpose as subqueries, they are more common in practice, as they tend to be cleaner for a future reader to follow the logic.

**Data Cleaning in Sql**

**LEFT** pulls a specified number of characters for each row in a specified column starting at the beginning (or from the left). As you saw here, you can pull the first three digits of a phone number using **LEFT(phone\_number, 3)**.

**RIGHT** pulls a specified number of characters for each row in a specified column starting at the end (or from the right). As you saw here, you can pull the last eight digits of a phone number using **RIGHT(phone\_number, 8)**.

**LENGTH** provides the number of characters for each row of a specified column. Here, you saw that we could use this to get the length of each phone number as **LENGTH(phone\_number)**.

There are 350 company names that start with a letter and 1 that starts with a number. This gives a ratio of 350/351 that are company names that start with a letter or 99.7%.

**SELECT** **SUM**(**num**) nums, **SUM**(letter) letters

**FROM** (**SELECT** **name**, **CASE** **WHEN** **LEFT**(**UPPER**(**name**), 1) **IN** ('0','1','2','3','4','5','6','7','8','9')

**THEN** 1 **ELSE** 0 **END** **AS** **num**,

**CASE** **WHEN** **LEFT**(**UPPER**(**name**), 1) **IN** ('0','1','2','3','4','5','6','7','8','9')

**THEN** 0 **ELSE** 1 **END** **AS** letter

**FROM** accounts) t1;

There are 80 company names that start with a vowel and 271 that start with other characters. Therefore 80/351 are vowels or 22.8%. Therefore, 77.2% of company names do not start with vowels.

**SELECT** **SUM**(vowels) vowels, **SUM**(other) other

**FROM** (**SELECT** **name**, **CASE** **WHEN** **LEFT**(**UPPER**(**name**), 1) **IN** ('A','E','I','O','U')

**THEN** 1 **ELSE** 0 **END** **AS** vowels,

**CASE** **WHEN** **LEFT**(**UPPER**(**name**), 1) **IN** ('A','E','I','O','U')

**THEN** 0 **ELSE** 1 **END** **AS** other

**FROM** accounts) t1;

**POSITION** takes a character and a column, and provides the index where that character is for each row. The index of the first position is 1 in SQL. If you come from another programming language, many begin indexing at 0. Here, you saw that you can pull the index of a comma as **POSITION(',' IN city\_state)**.

**STRPOS** provides the same result as **POSITION**, but the syntax for achieving those results is a bit different as shown here: **STRPOS(city\_state, ',')**.  
note, both **POSITION** and **STRPOS** are case sensitive, so looking for **A** is different than looking for **a**. 

Therefore, if you want to pull an index regardless of the case of a letter, you might want to use **LOWER** or **UPPER** to make all of the characters lower or uppercase.

1. Use the accounts table to create **first** and **last** name columns that hold the first and last names for the primary\_poc.

**SELECT** **LEFT**(primary\_poc, STRPOS(primary\_poc, ' ') -1 ) first\_name,

**RIGHT**(primary\_poc, **LENGTH**(primary\_poc) - STRPOS(primary\_poc, ' ')) last\_name

**FROM** accounts;

1. Now see if you can do the same thing for every rep name in the sales\_reps table. Again provide **first** and **last** name columns.

**CONCAT**

Piping ||

Each of these will allow you to combine columns together across rows. In this video, you saw how first and last names stored in separate columns could be combined together to create a full name: **CONCAT(first\_name, ' ', last\_name)** or with piping as **first\_name || ' ' || last\_name**.

1. Each company in the accounts table wants to create an email address for each primary\_poc. The email address should be the first name of the **primary\_poc** . last name **primary\_poc** @ company name .com.

WITH t1 AS (

**SELECT** **LEFT**(primary\_poc, STRPOS(primary\_poc, ' ') -1 ) first\_name, **RIGHT**(primary\_poc, **LENGTH**(primary\_poc) - STRPOS(primary\_poc, ' ')) last\_name, **name**

**FROM** accounts)

**SELECT** first\_name, last\_name, **CONCAT**(first\_name, '.', last\_name, '@', **name**, '.com')

**FROM** t1;

1. You may have noticed that in the previous solution some of the company names include spaces, which will certainly not work in an email address. See if you can create an email address that will work by removing all of the spaces in the account name, but otherwise your solution should be just as in question 1. Some helpful documentation is [**here**](https://www.postgresql.org/docs/8.1/static/functions-string.html).

WITH t1 AS (

**SELECT** **LEFT**(primary\_poc, STRPOS(primary\_poc, ' ') -1 ) first\_name, **RIGHT**(primary\_poc, **LENGTH**(primary\_poc) - STRPOS(primary\_poc, ' ')) last\_name, **name**

**FROM** accounts)

**SELECT** first\_name, last\_name, **CONCAT**(first\_name, '.', last\_name, '@', **REPLACE**(**name**, ' ', ''), '.com')

**FROM** t1;

**TO\_DATE**

**CAST**

Casting with ::

**DATE\_PART('month', TO\_DATE(month, 'month'))** here changed a month name into the number associated with that particular month.

Then you can change a string to a date using **CAST**. **CAST** is actually useful to change lots of column types. Commonly you might be doing as you saw here, where you change a string to a date using **CAST(date\_column AS DATE)**. However, you might want to make other changes to your columns in terms of their data types. You can see other examples [**here**](http://www.postgresqltutorial.com/postgresql-cast/).

In this example, you also saw that instead of **CAST(date\_column AS DATE)**, you can use **date\_column::DATE**.

### Expert Tip

Most of the functions presented in this lesson are specific to strings. They won’t work with dates, integers or floating-point numbers. However, using any of these functions will automatically change the data to the appropriate type.

**LEFT**, **RIGHT**, and **TRIM** are all used to select only certain elements of strings, but using them to select elements of a number or date will treat them as strings for the purpose of the function. Though we didn't cover **TRIM** in this lesson explicitly, it can be used to remove characters from the beginning and end of a string. This can remove unwanted spaces at the beginning or end of a row that often happen with data being moved from Excel or other storage systems.

There are a number of variations of these functions, as well as several other string functions not covered here. Different databases use subtle variations on these functions, so be sure to look up the appropriate database’s syntax if you’re connected to a private database.The **[Postgres literature](http://www.postgresql.org/docs/9.1/static/functions-string.html" \t "_blank)** contains a lot of the related functions.

Notice, this new date can be operated on using **DATE\_TRUNC** and **DATE\_PART** in the same way as earlier lessons.

**SELECT** date orig\_date, (**SUBSTR**(date, 7, 4) || '-' || **LEFT**(date, 2) || '-' || **SUBSTR**(date, 4, 2))::DATE new\_date

**FROM** sf\_crime\_data;

In general, **COALESCE** returns the first non-NULL value passed for each row.

**SELECT** **COALESCE**(a.**id**, a.**id**) filled\_id, a.**name**, a.website, a.lat, a.**long**, a.primary\_poc, a.sales\_rep\_id, o.\*

**FROM** accounts a

**LEFT** **JOIN** orders o

**ON** a.**id** = o.account\_id

**WHERE** o.total **IS** NULL;

**Window Function**

PostgreSQL’s documentation does an excellent job of [**introducing the concept of Window Functions**](https://www.postgresql.org/docs/9.1/static/tutorial-window.html): a window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

Through introducing window functions, we have also introduced two statements that you may not be familiar with: **OVER** and **PARTITION BY**. These are key to window functions. Not every window function uses **PARTITION BY**; we can also use **ORDER BY** or no statement at all depending on the query we want to run. You will practice using these clauses in the upcoming quizzes. If you want more details right now, [**this resource**](https://blog.sqlauthority.com/2015/11/04/sql-server-what-is-the-over-clause-notes-from-the-field-101/) from Pinal Dave is helpful.

Note: You can’t use window functions and standard aggregations in the same query. More specifically, you can’t include window functions in a ***GROUP BY*** clause.

## Ranking Total Paper Ordered by Account

**SELECT** **id**,

account\_id,

total,

**RANK**() **OVER** (**PARTITION** **BY** account\_id **ORDER** **BY** total **DESC**) **AS** total\_rank

**FROM** orders

The ORDER BY clause is one of two clauses integral to window functions.

The ORDERand PARTITION define what is referred to as the “window”—the ordered subset of data over which calculations are made. Removing ORDER BY just leaves an unordered partition; in our query's case, each column's value is simply an aggregation (e.g., sum, count, average, minimum, or maximum) of all the standard\_qty values in its respective account\_id.

As Stack Overflow user mathguy [**explains**](https://stackoverflow.com/questions/41364665/analytic-count-over-partition-with-and-without-order-by-clause):

The easiest way to think about this - leaving the ORDER BY out is equivalent to "ordering" in a way that all rows in the partition are "equal" to each other. Indeed, you can get the same effect by explicitly adding the ORDER BY clause like this: ORDER BY 0(or "order by" any constant expression), or even, more emphatically, ORDER BY NULL.

You can use window functions to identify what percentile (or quartile, or any other subdivision) a given row falls into. The syntax is NTILE(\*# of buckets\*). In this case, ORDER BY determines which column to use to determine the quartiles (or whatever number of ‘tiles you specify).

Use the NTILE functionality to divide the accounts into 4 levels in terms of the amount of standard\_qty for their orders. Your resulting table should have the account\_id, the occurred\_at time for each order, the total amount of standard\_qtypaper purchased, and one of four levels in a standard\_quartile column.

**SELECT** **id**,

account\_id,

occurred\_at,

standard\_qty,

NTILE(4) **OVER** (**PARTITION** **BY** account\_id **ORDER** **BY** standard\_qty) **AS** standard\_quartile

**FROM** orders

**ORDER** **BY** account\_id **DESC**

Use the NTILE functionality to divide the accounts into two levels in terms of the amount of gloss\_qty for their orders. Your resulting table should have the account\_id, the occurred\_at time for each order, the total amount of gloss\_qtypaper purchased, and one of two levels in a gloss\_half column.

**SELECT** **id**,

account\_id,

occurred\_at,

gloss\_qty,

NTILE(2) **OVER** (**PARTITION** **BY** account\_id **ORDER** **BY** gloss\_qty) **AS** gloss\_half

**FROM** orders

**ORDER** **BY** account\_id **DESC**

Use the NTILE functionality to divide the orders for each account into 100 levels in terms of the amount of total\_amt\_usd for their orders. Your resulting table should have the account\_id, the occurred\_at time for each order, the total amount of total\_amt\_usd paper purchased, and one of 100 levels in a total\_percentilecolumn.

**SELECT** **id**,

account\_id,

occurred\_at,

total\_amt\_usd,

NTILE(100) **OVER** (**PARTITION** **BY** account\_id **ORDER** **BY** total\_amt\_usd) **AS** total\_percentile

**FROM** orders

**ORDER** **BY** account\_id **DESC**