Data Analytics

Querying Data with SQL

GENERAL ASSEMBLY

What You'll Learn Today

In today's lesson, we'll:

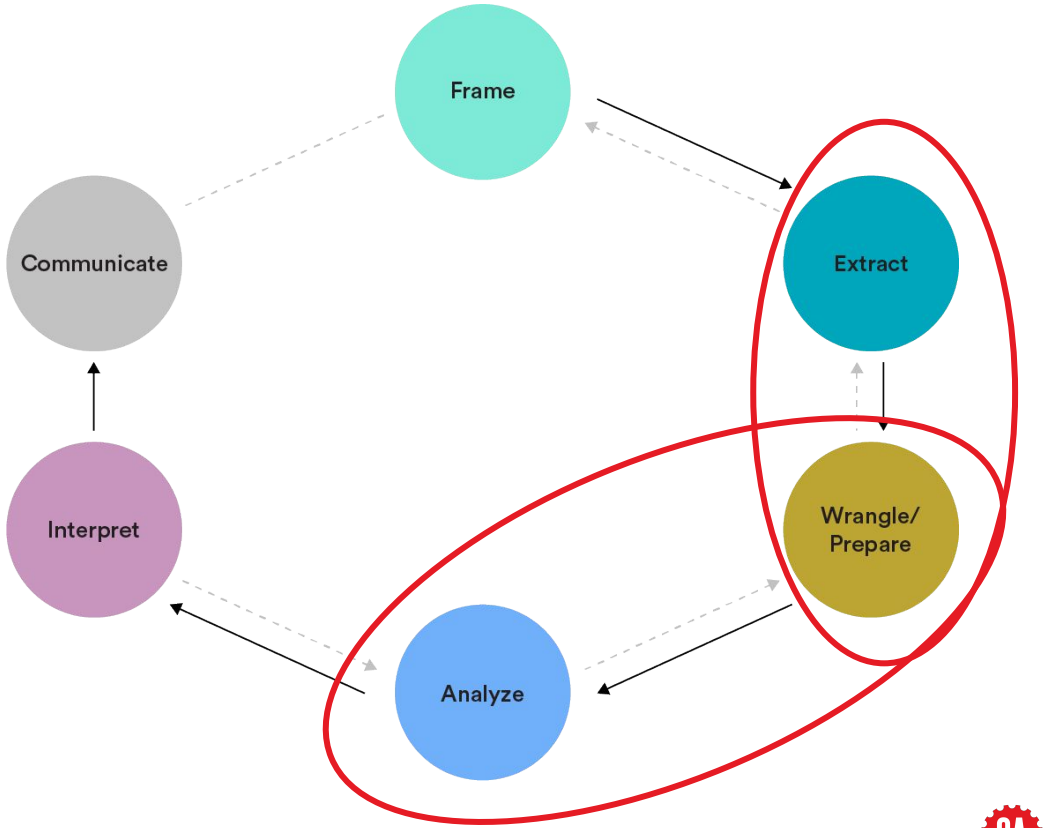
- Practice the concepts and syntax of advanced JOINs such as EXCEPT, FULL, and OUTER.
- Handle NULLs in SQL.
- Apply string, math, and date functions in SQL to prepare and analyze data.
- Practice writing SQL queries with advanced functions to solve business problems.

Where We Are in the DA Workflow

Extract: Select, import, and clean relevant data.

Wrangle/prepare: Clean and prepare relevant data.

Analyze: Structure, comprehend, and visualize data.



Querying Data With SQL

Advanced JOINs and NULLs

Types of JOINs

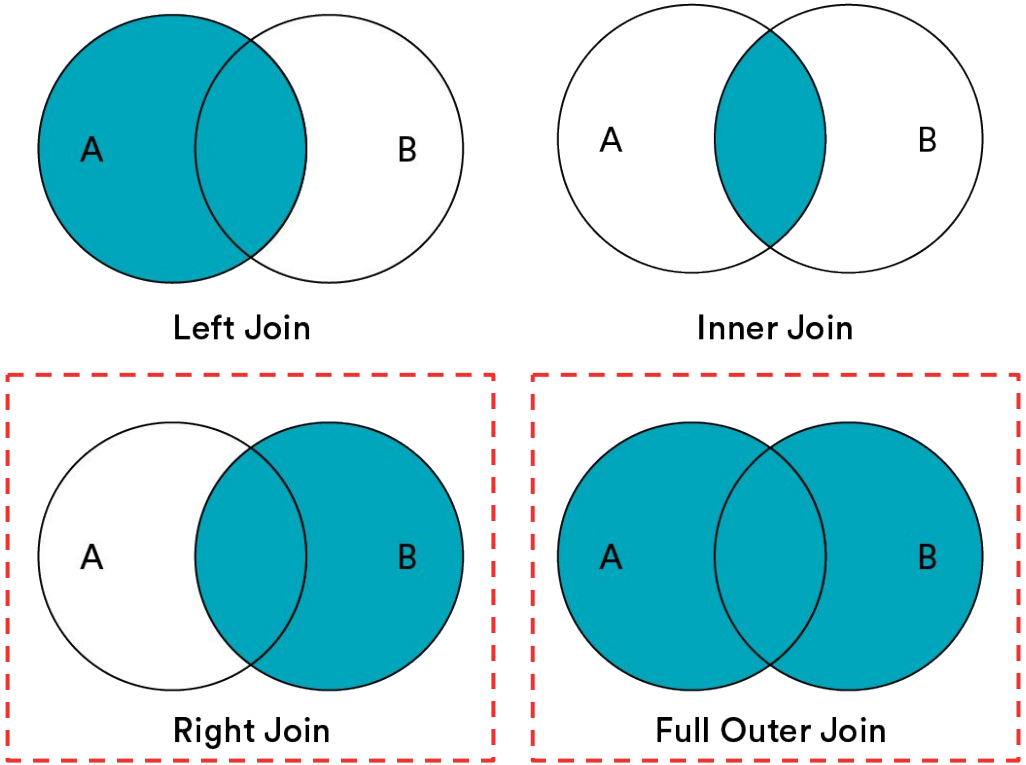
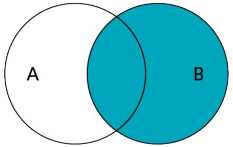


Image source: GA Instruction

RIGHT JOINs



A **RIGHT JOIN** yields data that two tables have in common and data from the **secondary** table that doesn't have matching data to join to in the primary table.

people

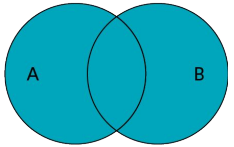
id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	5

vehicles

id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

id	name	vehicle_id	vehicle_name
	NULL	1	Explorer
	NULL	2	Civic
1	Janet	3	Corolla
2	Emily	4	Impala

FULL OUTER JOINs



A **FULL OUTER JOIN** returns **all** data from each table, **regardless** of whether or not they have matching data in the other table.

people

id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	5

vehicles

id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

id	name	vehicle_id	vehicle_name
	NULL	1	Explorer
	NULL	2	Civic
1	Janet	3	Corolla
2	Emily	4	Impala
3	Yoko	5	NULL

Partner Exercise:

How Will You JOIN This?

Superstore's VP of Operations wants to identify the products returned without a given reason for orders that were made in 2020. Work with your partner to figure out the most efficient way to JOIN these tables.

Orders Table — Primary table

order_id [PK] text	order_date timestamp without tim	ship_date timestamp without time	ship_mode text	customer_id text	product_id [PK] text	sales numeric	quantity integer	discount numeric	profit numeric	postal_code text	region_id integer
CA-2020-1011131	2020-01-13 00:00:00	2020-01-15 00:00:00	Second Class	SK-19990	TEC-PH-10003072	629.96	7	0.27	0.19	[null]	6093
CA-2020-1065839	2020-01-13 00:00:00	2020-01-13 00:00:00	Standard Class	MM-17920	OFF-AP-10001947	29.31	8	0.57	0.47	[null]	15598

Returns Table — Secondary table

order_id [PK] text	return_date timestamp without time	return_quantity integer	reason_returned text	product_id text
CA-2020-1011131	2020-02-05 00:00:00	1	Not Given	TEC-PH-10003072
CA-2020-1065839	2020-02-06 00:00:00	1	Not Given	OFF-AP-10001947

Partner Exercise:

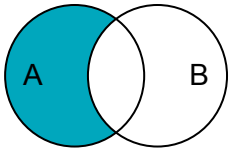
How Will You JOIN This? | **Solution**

Your VP of Operations wants to identify the products returned without a given reason for orders that were made in 2020.

Solution query:

```
SELECT DISTINCT o.product_id, r.reason_returned
FROM orders o
JOIN returns r
    ON o.order_id = r.order_id
    AND o.product_id = r.product_id
WHERE r.reason_returned = 'Not Given'
AND o.order_date BETWEEN '2020-01-01' and '2020-12-31';
```

EXCEPTION JOINs



An **EXCEPTION JOIN** returns **only** the data from the **primary** table selected that **doesn't have matching data** to JOIN to in the secondary table.

people

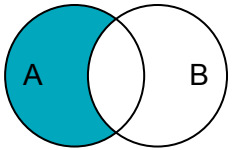
id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	5

vehicles

id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

id	name	vehicle_id	vehicle_name
3	Yoko	NULL	NULL

EXCEPTION JOIN | Basic Syntax



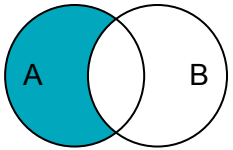

```
SELECT id FROM people
EXCEPT
SELECT id FROM vehicles
```

id	name	vehicle_id	vehicle_name
3	Yoko	NULL	NULL

```
SELECT id FROM vehicles
EXCEPT
SELECT id FROM people
```

id	name	vehicle_id	vehicle_name
NULL	NULL	1	Explorer
NULL	NULL	2	Civic

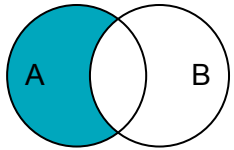
Exception JOINs in PostgreSQL



The syntax for **EXCEPT** in PostgreSQL is:

```
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions]
EXCEPT
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions];
```

EXCEPTION JOIN Syntax | PostgreSQL



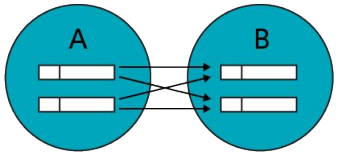
Next, let's look at an example of an **EXCEPT** query in PostgreSQL that returns furniture products that were **not** sold in 2019:

```
SELECT product_id
FROM products
WHERE category = 'Furniture'

EXCEPT

SELECT product_id
FROM orders
WHERE order_date BETWEEN '2019-01-01' and '2019-12-31';
```


Cross JOIN



A **CROSS JOIN** matches every row of the primary table with every row of the secondary table, resulting in a Cartesian product. This is generally done to produce a new table of all possible combinations.

id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	5

id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

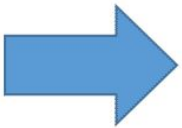
Cross JOIN Example

In this example, a cross JOIN returns every combination of color and size: Since there are two rows in Color and four in Size, the final result is eight rows (2 x 4 = 8).

Tables

Color
Red
Blue

Size
Small
Medium
Large
Extra Large



Query Result

Color	Size
Red	Small
Blue	Small
Red	Medium
Blue	Medium
Red	Large
Blue	Large
Red	Extra Large
Blue	Extra Large

```
SELECT c.Color,
       s.Size
FROM Color c
      CROSS JOIN Size s
```


Source: EssentialSQL

Nulls in SQL

NULL Values

Any field can have the value **NULL**, which **represents a missing value**, but a NULL is different than a zero or a blank because:

- Zero is a value.
- A blank cell could have been left blank *on purpose.*
- In some cases, a blank represents data.

NULL values can be produced with all JOINs except INNER.

Knowing What's Missing | An Example

It's important to know *exactly* what **NULL** indicates for each field. For example, in the same PIN number field of a user login, **NULL** could mean:

- A user signed up but has not yet entered a PIN number.
- A user will never enter a PIN number because they are using another authentication method.
- A user signed up before PIN numbers were supported and is not required to enter one.

Other Values for Missing Data

Be careful! Depending on the datatype of a field, other values could *also* indicate missing values or data collection errors. Some common values include:

- Strings: “N/A”, “Unknown”, “Missing”
- Integers: 0
- Floats: NaN, Inf (Not a Number, Infinity)

Because of this, testing for **NULL** is not always enough.

Partner Exercise:

NULL Values

Referring to the list below, discuss with your partner: **What are some ways that the presence of NULL values can complicate the analysis for your project?**

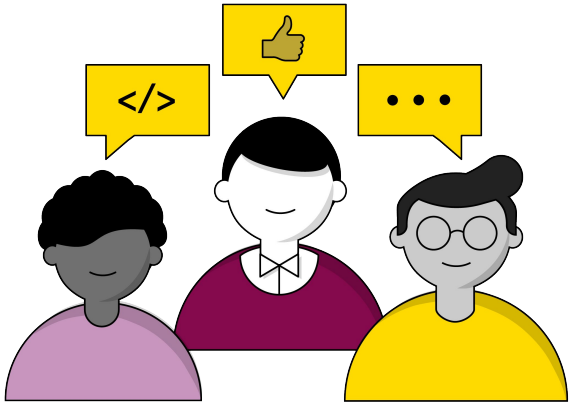
- Nulls cannot be added or subtracted.
- Nulls can undermine data counts.
- Nulls can complicate JOINs.
- Dividing by a NULL yields another NULL.

Be prepared to share your answers.

Discussion:
Blanks and Zeros

So far, we've talked about NULLs and how missing data can be represented. Let's pause for a second and discuss the following:

When are blanks and zero values appropriate?



Working With NULLs

“WHERE” to Find NULLs

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

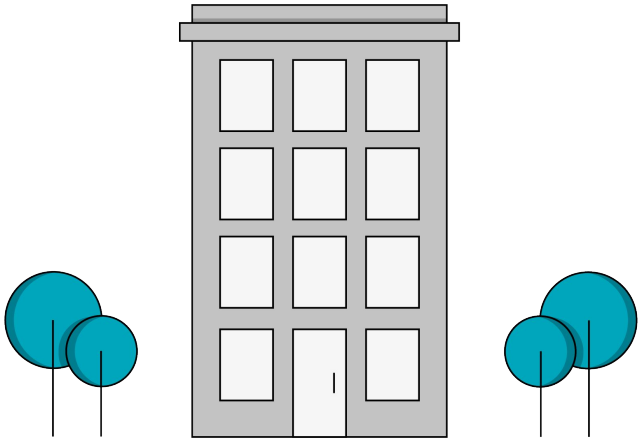
GROUP BY aggregates across values of a variable.

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

LIMIT limits results to the first **n** rows.

The Query Building



Guided Walk-Through:
Finding NULLs in WHERE

You can find **NULL** data using a **WHERE** clause. Let's run the following query:

```
SELECT order_id, region_id

FROM orders

WHERE postal_code = 'NULL';
```

Do you expect problems with this query? Why or why not?

Guided Walk-Through:
Finding NULLs With **IS NULL**

You can find **NULL** data using **IS NULL**:

```
SELECT order_id, region_id

FROM orders

WHERE postal_code IS NULL;
```

This will return all of the rows where the **postal_code** field is blank.

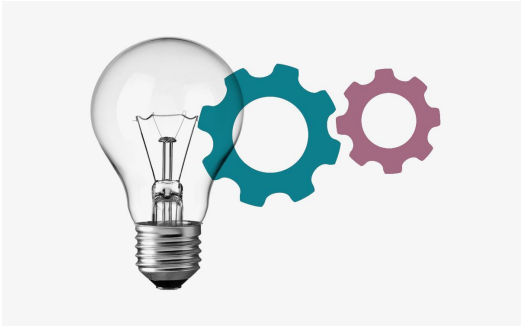
Guided Walk-Through:

Finding Non-NULLs With **IS NOT NULL**

We can also find non-**NULL** values by using **IS NOT NULL**, which is the opposite of **IS NULL**.

Run this query:

```
SELECT order_id, region_id
FROM orders
WHERE postal_code IS NOT NULL;
```

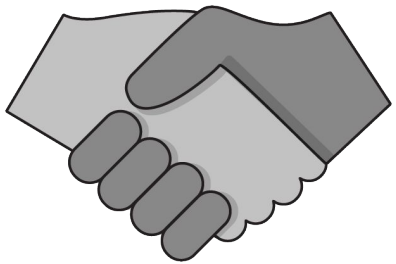


Partner Exercise:

Which Salesperson Is Selling the Most?

Last class, we looked at the number of sales that each salesperson made, including returns. Now, let's look at **the number of sales each salesperson made** based on orders that were *not* returned.

Work with your group to write a query with more than one JOIN.



Partner Exercise:

How Did it Go? | **Solutions**

Your query should look like this:

```sql
SELECT salesperson, COUNT(DISTINCT o.order_id) AS num_sales
FROM orders o
    JOIN regions reg ON o.region_id=reg.region_id
    LEFT JOIN returns r ON o.order_id=r.order_id
                        AND o.product_id = r.product_id
WHERE r.order_id IS NULL
GROUP BY 1
ORDER BY 2 DESC;
```


It Can Be Tricky to Work With NULLs and Zeros

...especially if your analysis requires addition/subtraction or division, because:

- No dividing by zero.
- Can't add or subtract a **NULL**.

Fortunately, various SQL tools can solve either situation:

- **NULLIF**
- **CASE**
- **COALESCE**

NULLIF

NULLIF (field, testing_value)

- Returns NULL if expressions are equal.
- Otherwise returns the first expression.
- Can test for zero values and prevent division-by-zero errors.

Field 1	Field 2
15	1
20	0
25	5
30	3

Output
15
NULL
5
10

Example: Using the sample table, divide Field 1 by Field 2. You must ensure there are no zeros in Field 2:

```
SELECT Field1 / NULLIF ( Field2, 0 )
```

Guided Walk-Through:
Using NULLIF to Prevent Division-by-Zero Errors

Use **NULLIF( )** to prevent division-by-zero errors:

```
SELECT
  product_id,
  quantity/discount AS discount_per_item
FROM orders
WHERE ship_mode = 'Standard Class';
```

RAISES AN ERROR

```
SELECT
  product_id,
  quantity/NULLIF(discount,0) AS discount_per_item
FROM orders
WHERE ship_mode = 'Standard Class';
```

WORKS!

CASE

CASE (WHEN-THEN) can also be used to change zeros to NULLs.

Example:

```
SELECT Field1, Field2,
CASE
     WHEN Field2 = 0 THEN NULL
     WHEN Field2 > 0 THEN Field1/Field2
END AS Field3
FROM yellow_table;
```

Field1	Field2
15	1
20	0
25	5
30	3

Field1	Field3
15	15
20	NULL
25	5
30	10

Guided Walk-Through:
Changing Zeros to NULLs With CASE

Your VP of Sales wants to explore bulk discounts for standard shipments. Let's walk through the most efficient way to avoid NULLs in our calculations:

```
SELECT product_id,
CASE
     WHEN discount = 0 THEN NULL
     WHEN discount > 0 THEN quantity/discount
END AS discount_per_item
FROM orders
WHERE ship_mode = 'Standard Class';
```


COALESCE

COALESCE (field2, alternate_value, 0) is the opposite of NULLIF:

- Turns a NULL into a zero or another value.
- Returns the first **non-null** column argument.
- Could return an average value.

Field 1	Field 2	coalesce
15	1	1
20	NULL	20
25	5	5
NULL	NULL	0

Examples:

```
SELECT COALESCE ( description,
    short_description, '(none)' )
SELECT COALESCE ( field2, field1, 0 )
```


Guided Walk-Through:

Managing NULL Values With COALESCE

COALESCE returns the first **non-NULL** argument in a list.

```
SELECT order_id, product_id, sales,
  COALESCE(postal_code, region_id::text) AS coalesced
FROM orders;
```

***Note**: Arguments in COALESCE( ) must be the same data type.*

JOINing Scenarios

JOINing Scenarios

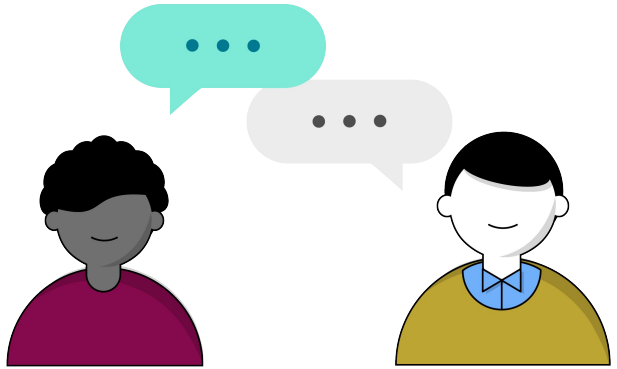
In the next slides, you'll review three sets of scenarios with a partner, identify which type of **JOIN** you'll use, and share your answers with the class. Keep in mind:

Table 1 represents the primary table:

Table 1 => Primary => Left

Table 2 represents the secondary table:

Table 2 => Secondary => Right



Partner Exercise:

Scenarios

1. Table 1 has pending deleted items. Table 2 has item **sales**. You need to find all of the items that are pending deletion and have no sales data.
    - Your JOIN solution choice is...

2. Table 1 has a list of **vendors**. Table 2 has a list of **addresses** for vendors. Table 2 is missing some data. For your purposes, you want to see all of the vendor information. You also want to see all of the address information you can find, even if some of it is missing.
    - Your JOIN solution choice is...

Partner Exercise:

More Scenarios

3. Table 1 has a list of **members**. Table 2 has a list of **items purchased**. You are running a ground beef recall and need to get an exact match of members who have purchased this item.
    - Your JOIN solution choice is...

4. Table 1 has **retail location** information. Table 2 has **product** information. You want to match all products with all locations to create a list of all possible combinations.
    - Your JOIN solution choice is...

Partner Exercise:

Even More Scenarios

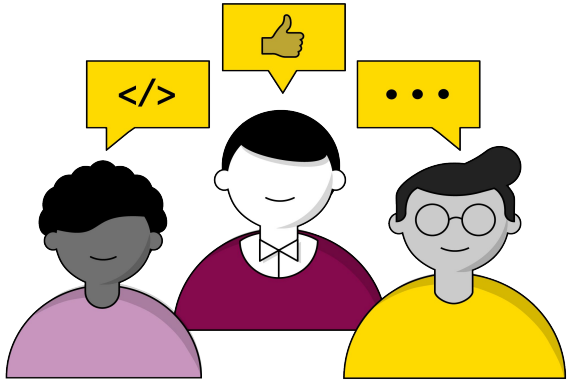
5. Table 1 has a list of **item descriptions**. Table 2 has a list of **item sales**. Table 1 is missing some data. For your purposes, you want to see all of the item sales information. You also want to see all of the item description information, even if some descriptions are missing.
    - Your JOIN solution choice is...

6. Table 1 has pending **deleted** items. Table 2 has **item sales**. You need to find all items that are pending deletion and have **no sales data**.
    - Your JOIN solution choice is...

Discussion:
How Did it Go?

Let’s take a few minutes to debrief.

1. **What were some of the roadblocks when determining which JOIN to use?**
2. **Did you and your partner disagree about some of the solution choices?**
3. **If so, how did you resolve it?**



Solo Exercise:

OUTER and FULL OUTER JOINs

Let's field this request for information:

We want to see all of the information we can get on customers who made orders in 2020.

Construct a query to provide the necessary information.

Solo Exercise:

OUTER vs. RIGHT JOINs — Which Is Better?

Sample Solution 1 - 18,773 rows

```
SELECT *
FROM customers
RIGHT OUTER JOIN orders
ON orders.customer_id = customers.customer_id
WHERE DATE_PART('year', orders.order_date) = '2020'
```

Sample Solution 2 - 18,773 rows

```
SELECT *
FROM customers a
RIGHT JOIN orders b ON a.customer_id=b.customer_id
WHERE DATE_PART('year', b.order_date) = '2020'
```


Optimizing Queries With JOINs

Discussion:
Query Runtime and JOINs

No one likes a slow query or a system that crashes regularly.

In order for a query to run efficiently, we need to reduce the number of calculations that a database must perform when running our query.

With that in mind, do you know why it's important to consider performance when creating JOINs?

Guided Walk-Through:
Optimizing JOINs

The VP of Sales is launching a “Go Green” campaign and wants to provide discounts on recycled products. Let’s run each of these queries:

```sql
SELECT DISTINCT a.product_id
FROM orders a, products b
WHERE b.product_name LIKE '%Recycled%';
```

Creates a CROSS JOIN (Cartesian Product) by excluding the ON or a link in the WHERE clause.

```sql
SELECT DISTINCT a.product_id
FROM orders a
INNER JOIN products b USING (product_id)
WHERE b.product_name LIKE '%Recycled%';
```

Uses INNER JOIN and applies WHERE before — more efficient query

Recommended Practice for Faster Queries

A couple more things you can do:

- Follow the SQL order of operations.
- `SELECT` specific fields instead of using `SELECT *`.
- Filter one or both of the tables with a `WHERE` clause in the same query as the `JOIN`. Depending on the server environment, it may save time.
- `HAVING` statements are calculated after `WHERE` statements.
- When testing `JOINs`, use `LIMIT` to control query sizes.
- Use `IS NULL` or `IS NOT NULL` to test for NULLs in a column.

Source: SQL Query Order of Operations

Independent Practice

Solo Exercise:

JOINs and NULLs

Write queries to answer the following questions for Superstore:

1. Show the orders made by customers in the Consumer segment.
    a. Limit to 1,000 rows.
    b. Experiment with grouping and order.
2. How many orders included Photo Frame products?
3. Which Photo Frame products were not sold?
4. Which distinct products were sold in France?
5. Which Recycled products were sold in the United States?
6. Which unique products, other than Photo Frame products, were sold in Canada?
7. Were there any products that were not sold?
8. Were there any orders from countries outside of our sales regions?

Solo Exercise:

JOINs & NULLs | **Solutions 1 & 2**

1. Show the orders made by customers in the Consumer segment. Try connecting the keys with USING. Limit to 1,000 rows. Experiment with grouping and order:

```
SELECT * FROM orders a
INNER JOIN customers b ON a.customer_id = b.customer_id
WHERE b.segment = 'Consumer' LIMIT 1000;
```

2. How many orders included Photo Frame products?
    - 7,015 orders

```
SELECT COUNT(DISTINCT order_id) FROM orders a
INNER JOIN products b ON a.product_id = b.product_id
WHERE b.product_name like '%Photo Frame%';
```

Solo Exercise:
JOINs & NULLs | **Solutions 3 & 4**

3. Which Photo Frame products were not sold?
    - 0 items were not sold (distinct and distinct on don't change the row count).

```
SELECT product_name FROM products a
LEFT JOIN orders b ON a.product_id = b.product_id
WHERE a.product_name LIKE '%Photo Frame%' AND b.order_id IS NULL;
```

4. Which distinct products were sold in France?
    - 9,401 unique products

```
SELECT DISTINCT a.product_id
FROM orders a INNER JOIN regions b ON a.region_id = b.region_id
WHERE b.country like '%France%';
```

Solo Exercise:

JOINs & NULLs | **Solutions 5 & 6**

5. Which Recycled products were sold in the United States?
    - 403 unique products

```
SELECT DISTINCT a.product_id FROM orders a
INNER JOIN regions b ON a.region_id = b.region_id
INNER JOIN products c ON a.product_id = c.product_id
WHERE b.country LIKE '%United States%'
     AND c.product_name LIKE '%Recycled%';
```

6. Which unique products, other than Photo Frame products, were sold in Canada?
    - 4632 unique products

```
SELECT DISTINCT a.product_id FROM orders a
INNER JOIN regions b ON a.region_id = b.region_id
INNER JOIN products c ON a.product_id = c.product_id
WHERE b.country LIKE '%Canada%'
     AND c.product_name NOT LIKE '%Photo Frame%';
```

Solo Exercise:
JOINs & NULLs | **Solutions 7 & 8**

7. Were there any products that were not sold? This is the same as asking whether there are any products listed in the products table that are not listed in the **`orders`** table?
    - There was one, FUR-TA-10000304

```
SELECT p.product_id FROM products p
EXCEPT
SELECT DISTINCT o.product_id FROM orders o;
```

8. Were there any orders from countries outside of our sales regions?
    - There are no rows in the **`orders`** table with a **`NULL`** region_id

```
SELECT o.region_id FROM orders o
EXCEPT
SELECT r.region_id FROM regions r;
```

Querying Data with SQL

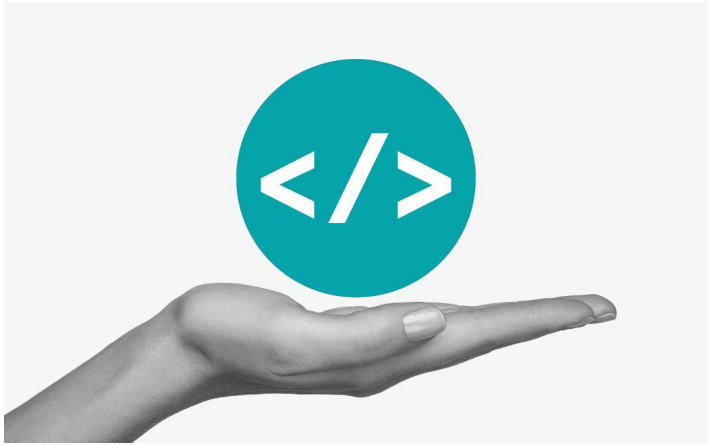
SQL Functions

Discussion:

Functions, Functions, Functions!

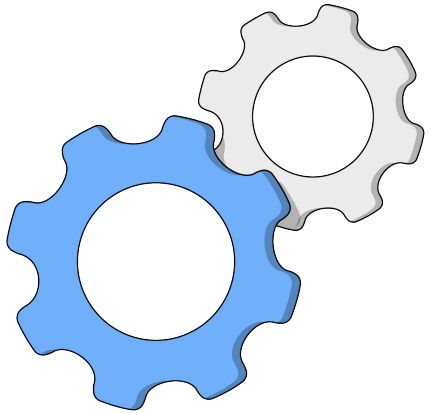
The SQL server has over **200** functions. Some are straightforward and common, while others are highly specialized. Not to mention they can vary across environments...

That's a lot to choose from! How do you know when and how to execute them?



What Do You Do With 200+ Functions?

Answer: Build a foundational understanding of how functions relate to ***operators, delimiters, arguments, and expressions*** — which exist in all SQL environments. Let's start with *operators and delimiters*:



- **Operators:** Symbols or keywords used to perform logical and mathematical operations in SQL.
- **Delimiters:** Used to separate or mark the start and end of SQL language elements.

Source: TutorialsPoint

Discussion:

Operators and Delimiters — What's the Difference?

Knowing that "***Operators*** are the ***math*** of SQL. ***Delimiters*** are the ***grammar*** of SQL," **can you tell which is which in this query?**

```
SELECT order_id , profit , MAX ( sales ) ,
    CASE
        WHEN CAST ( profit AS int ) >= 1000 THEN ' large '
        WHEN CAST ( profit AS int ) >= 50 THEN ' large '
        ELSE ' small '
    END AS profit_size
FROM orders
GROUP BY order_id , profit ;
```


Operators

There are three main categories of operators:

Arithmetic	Relational	Logical
Work the same as the identical operators found in math.	Produce TRUE, FALSE, or UNKNOWN results.	Expand on the basic relational operators.

`+`
`-`
`*`
`/`
`**`
	`=`
	`!= or <>`
	`<`
	`<= or !>`
	`>`
	`>= or !<`
		`ALL/AND`
		`ANY/OR`
		`BETWEEN`
		`EXISTS`
		`IN`
		`LIKE`
		`NOT`
		`IS NULL`
		`UNIQUE`

Delimiters

Similar to regular punctuation, delimiters are used to separate or mark the start and end of SQL language elements.

If delimiters are omitted, the query results in *error* conditions.

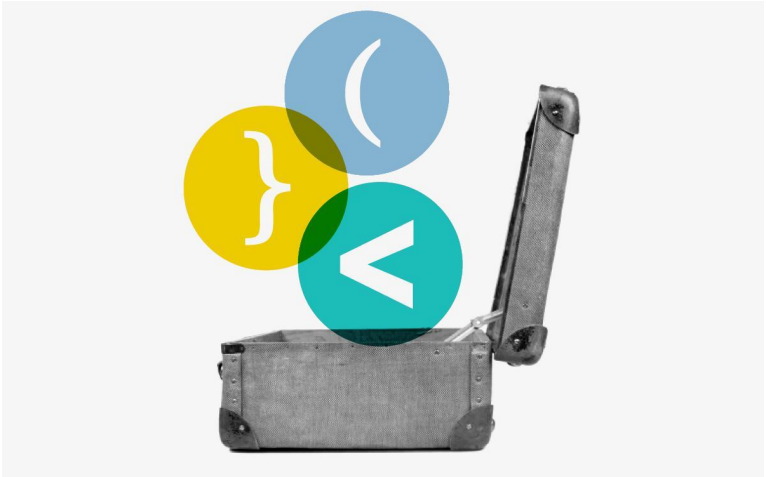
Delimiters

Delimiters	Description
`,`	Separates list elements.
`.`	Connects elements of a qualified name/decimal.
`;`	Terminates a statement.
`=`	Assignment operator/equality in conditional statement.
`:`	Connects prefix to statements, lower bound to upper bound, and is used in RANGE statements.
Blank	Separates elements.
`( )`	Encloses lists, expressions, iteration factors, repetition factors, and information associated with a keyword.
`--` `/* text */`	Basic comment. In-line comment, multi-line comment.
`'text'`	Denotes a string, as opposed to text being used as a variable name.

Arguments and Expressions

Arguments are literal **values** or **variables** in an SQL function.

Expressions are formed by a **combination** of SQL operators, delimiters, table columns, constants, and/or SQL functions.



Arguments and Expressions | An Example

SUM(col1, col2) + 5 is an expression where:

- **SUM(col1, col2)** is an SQL function.
- **col1, col2** are arguments.
- **+** is an arithmetic operator.
- **5** is a constant.

Where Functions Live in a Query

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

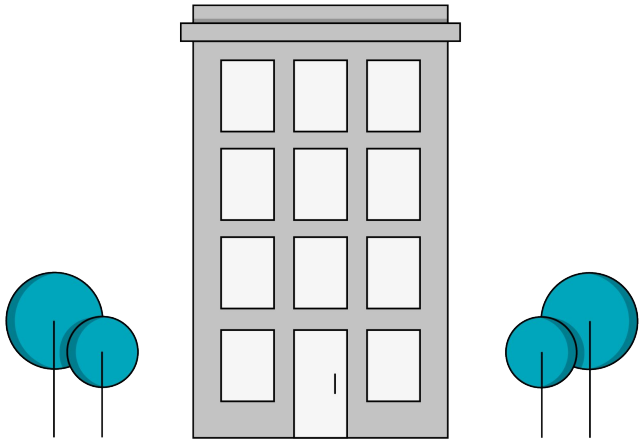
GROUP BY aggregates across values of a variable.

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

LIMIT limits results to the first **n** rows.

The Query Building



Math Functions

Discussion:

When Do You Use Math Functions?

Let's consider how SQL math functions can be used in your work (or course projects). Scenarios may include calculating:

- An employee's time-and-a-half pay rate.
- The retail cost of a product after adding in the markup percentage.
- Profit margins.
- Estimated sales in a given quarter.
- Customers with the highest number of transactions.

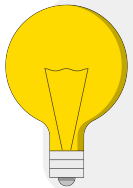
What are some other business problems that can be solved by applying SQL math functions?

Math Functions | An Overview

Math functions return calculated values from your data. Many of these functions will require a **GROUP BY** statement to clearly identify which rows to select.

Best when: You have numeric input(s) and want to return numeric values.

Frequently Used Math Functions
`COUNT`
`AVG`
`MIN / MAX`
`SUM`
`ROUND`



Pro tip: The default is to use all rows, but some functions aren't able to do this.

Math Functions | COUNT

COUNT returns the number of rows that match some specified criteria. If the criteria includes only a column name, COUNT will return the number of non-NULL values in that column.

Syntax: `Count ( field1 )`

Example use case: Count the number of instances when a transaction occurred.

Math Functions | Average

The **AVG** function returns the average value of a numeric column.

Syntax: `AVG ( field1 )`

Example use case: You can use AVG to help you figure out things like the monthly average shipping costs or the average sales in a quarter.

Math Functions | Minimum

The **MIN** function returns the smallest value of the selected column.

Syntax: `MIN ( field1 )`

Example use case: Use this function to identify the customer with the lowest amount of total sales.

Math Functions | Maximum

The **MAX** function returns the largest value of the selected column.

Syntax: `MAX ( field1 )`

Example use case: Similarly, you can use this function to find the customer with the highest amount of total sales.

Math Functions | SUM

The **SUM** function is used to find the sum of a field in various records.

Syntax: `SUM ( field1 )`

Example use case: Use SUM to calculate the total earning of an employee, including regular salary and overtime pay.

Math Functions | ROUND

The **ROUND** function returns a given number rounded **n** places to the right of the decimal point.

If **n** is negative, it will be rounded **n** places to the left of the decimal point. This function operates within queries, not with hard-coded numbers.

Syntax: `ROUND ( numeric_expression, n )`

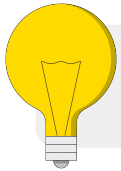
Try it: x = 177.3589, what will each return?

ROUND(x, 2) = ? **ROUND**(x, -2) = ?

Integers and Floats... What's the Difference?

Before we continue, let's make a distinction regarding data types:

- **Integer**: A *whole number*. Doesn't have fractions, decimals, etc.
  - They are expressed as 1, 2, 3, 5000, 38983498, etc.
- **Float**: Lets you express up to approximately 15,000 digits after the decimal point for accuracy.
  - They are expressed as 1.0000, 2.0134, 3.1419, 3944.39, etc.



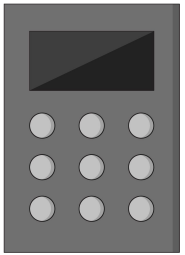
Pro tip: Most of the time, storing 10 digits is enough.

Discussion:

Data Types | Inputs and Outputs

Take a look at the calculations below. What do you notice?

- integer **+** integer = integer ➜ 3 + 5 = 8
- integer * integer = integer ➜ 3 * 9 = 27
- integer **/** integer = integer ➜ 5 / 3 = 1

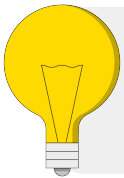


When completing a “math” calculation, **SQL uses the same data type for both inputs and outputs**. So when you input an integer, your output is also an integer.

Converting Data Types

To convert between data types, we can call out which data type we want using the **CAST** command.

- `CAST( SUM ( total ) AS int )` would change the SUM(total) from a decimal to an integer.
- `CAST( COUNT ( total ) AS decimal )` would change the COUNT(total) from an integer to a decimal.



We can also use a shorthand version by typing **::[datatype]** at the end. For example, we can change an integer, “3,” to a decimal, “3.0,” via **3::decimal**.

Partner Exercise:

Optional Practice With Math Functions

Work with a partner to write queries for the following:

1. Which shipping class has the highest quantity of shipped items?
2. How many orders had three or more items returned?
3. What's the least expensive subcategory of products?

Hint: Your first query will look something like this:

```
SELECT
    FUNCTION(_)
FROM
GROUP BY
ORDER BY
```

Partner Exercise:

Optional Practice With Math Functions | Solutions

1. **ANSWER:** Standard Class

```
SELECT ship_mode, SUM(quantity) AS total_quantity
FROM orders
GROUP BY ship_mode
ORDER BY total_quantity DESC;
```

2. **ANSWER:** 113

```
SELECT COUNT(order_id)
FROM returns
WHERE CAST(return_quantity AS int) >= 3;
```

3. **ANSWER:** Binders

```
SELECT sub_category, MIN(product_cost_to_consumer) AS min_cost
FROM products
GROUP BY sub_category
ORDER BY min_cost ASC;
```

String Functions

Discussion:

When Do You Use String Functions?

Let’s consider how string functions are used in your work (or course projects). Scenarios may include:

- Find all customers with the last name that starts with a specific letter.
- Identify all products that contain a specific word.
- Map employees that live in a given city.
- Correct spelling errors from survey data.
- Remove duplicate names from a mailing list.

What are some other business problems that can be solved by applying SQL string functions?

String Functions | Overview

String functions are used to combine fields that are character (text) data types and return either character or number values.

Best when: You want to organize or clean up text in varchar and char columns.

Keep in mind: Different SQL dialects have varied syntax for string, but the principles are consistent.

Frequently Used String Functions
CONCAT
LENGTH
REPLACE
LOWER / UPPER
LEFT / RIGHT
SUBSTRING
TRIM
SRPOS

String Functions | CONCAT

CONCAT combines two or more fields or expressions together.

Syntax: `CONCAT ( field1, field2, field3… )`

Example use case: When you want to generate mailing labels with the customers first and last names, which are kept in separate columns, in the database.

String Functions | LENGTH

LENGTH counts the length of characters in a field.

Syntax: `LENGTH ( field1 )`

Example use case: You can use this to do a string character count on things like product names and descriptions.

String Functions | REPLACE

REPLACE is similar to the Excel function SUBSTITUTE; it replaces a value in a field with another value.

Syntax: `REPLACE ( field_to_change, content_to_replace, new_content )`

Example use case: You can use this to correct spelling errors in your data or find and remove duplicate values.

String Functions | LENGTH + REPLACE

Combining a **LENGTH** function with a **REPLACE** function can also work as a word counter for text fields.

Let's use "the quick brown fox jumped over the lazy dog" as an example:

```
LENGTH ('the quick brown fox jumped over the lazy dog')
LENGTH ('thequickbrownfoxjumpedoverthelazydog')
```

= **44** characters

= **36** characters

What happens if we combine the two statements?

String Functions | LENGTH + REPLACE (Cont.)

If we combine the two statements, we'd see the difference: **44 - 36 = 8**

As we are counting spaces to get a word count, we'd add one to the result:

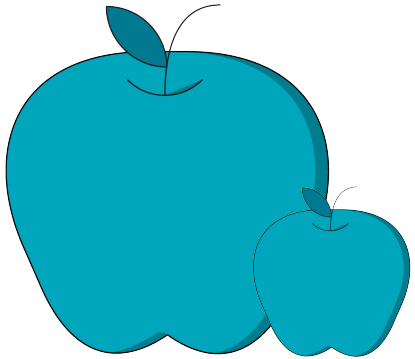
```
LENGTH ('the quick brown fox jumped over the lazy dog')
- LENGTH ( REPLACE ('the quick brown fox jumped over the lazy dog'),' ','')
```

Example use case: You can also use this to edit a product description or check for correct character lengths in a product ID.

Changing the Case

Changing the case is useful if you have **data inconsistencies in a table or across tables** and need to create a common key for a JOIN. For example:

- If a company is called “apple inc,” “APPLE inc,” or “Apple Inc” in the same table, then a WHERE or LIKE clause may match one but not all.
- Or if you need to join on a company name as a common key, there could be a mismatch across different tables.



String Functions | LOWER + UPPER

LOWER converts a field or expression to lowercase. **UPPER** converts a field or expression to uppercase.

Syntax: `LOWER ( field1 )` and `UPPER ( field1 )`

Example use case: You can use this to ensure the labeling of items such as customer IDs or product IDs (e.g. “Rh-19595” or “TEC-ac-10003033”) are consistent throughout a database.

String Functions | LEFT + RIGHT

LEFT/**RIGHT** substring selects a given number of characters from one side of the string.

- **LEFT:** Selects characters from the left.
- **RIGHT:** Selects characters from the right.

Syntax: `LEFT ( field1, length )` or `RIGHT ( field1, length )`

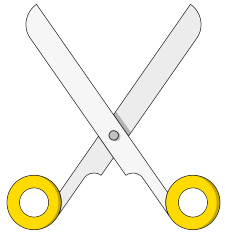
Example use case: Create a location ID for each employee by using the first three letters of a city name with the first five numbers of a zip code.

String Functions | TRIM

TRIM: Removes specific characters from the start of the field (**leading** characters), end of the field (**trailing** characters), or both.

Syntax:

```
TRIM ( leading 'characters', from field1 )
TRIM ( trailing 'characters', from field1 )
TRIM ( both 'characters', from field1 )
```



Example use case: You can use this to ensure the labeling of items such as customer IDs or product IDs (e.g. “RH-19595” or “TEC-AC-10003033”) do not have leading or trailing blank spaces.

String Functions | LEFT and RIGHT TRIM

LEFT and **RIGHT TRIM** remove blanks from the specified side.

- **LTRIM**: Trims all blanks from the left side.
- **RTRIM**: Trims all blanks from the right side.

Syntax: `LTRIM ( field1 )` or `RTRIM ( field1 )`

Example use case: Similar to TRIM, you can use these functions to ensure the labeling of items such as customer IDs or product IDs (e.g. “RH-19595” or “TEC-AC-10003033”) do not have leading or trailing blank spaces.

Guided Walk-Through:
Using STRPOS

STRPOS is used to find the position from where the substring is being matched within the “main” string.

Syntax: `STRPOS ( string, substring )`

Example: Say your product manager wants you to identify the brand name of each product in your product catalog.

First, you need to identify where to split the product_name, which is in the format “brand_name , product_details”: `STRPOS(product_name, ',')`.

Guided Walk-Through:

Using SUBSTRING

SUBSTRING allows you to retrieve specific characters within a field.

Syntax:

```
SUBSTRING ( field1, start position, number of characters to retrieve from start )
```

Now that we know the position of the delimiter, we can use SUBSTRING to retrieve all of the characters before the delimiter. Use REPLACE, to remove the end comma.

```
SELECT
REPLACE(SUBSTRING(product_name, 1,(STRPOS(product_name, ','))),',','')
FROM products;
```

Partner Exercise:

Optional Practice With String Functions

In the Superstore data set, **which product category has the longest new name that is less than 100 characters** when you combine its “sub_category field” with its “product_name” field as a “new_name” field?

Work with a partner to print:

- The category name in all caps.
- The length of the new name.
- The first five characters of the “product_id” without any hyphens or blank spaces.

Partner Exercise:

Optional Practice With String Functions | Solution

Answer: “Office Supplies”

```sql
SELECT UPPER(category) as upper_category,
LENGTH(CONCAT(sub_category, product_name)) AS new_name,
SUBSTRING(REPLACE(product_id, '-', ''), 1, 5) as sub_product_id
FROM products
WHERE LENGTH(CONCAT(sub_category, product_name)) < 100
ORDER BY 2 DESC
LIMIT 5;
```


Date Functions

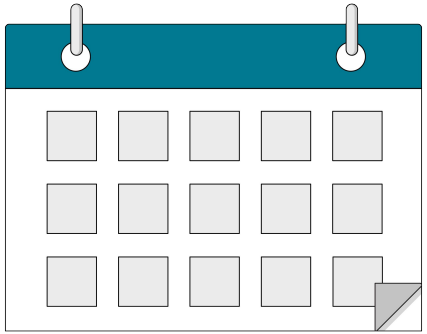
Discussion:

DATE Functions

Let's consider how dates are used in your work (or course projects).

Scenarios may include:

- Billing date by day of week.
- Changes in day of the week by year.
- Comparison of days of the week by two dates.
- Estimation based on day of week or on the previous year.
- Count of customers on a given day.
- Durations of shipments.
- Order date to ship date.



What are some other business problems where the ability to retrieve and format dates is highly useful?

Date Functions | Overview

Date functions are used to format dates.

Best approach: Have an overall understanding of what DATE functions can do and consult your vendor's DATE documentation.

Syntax varies depending on the SQL dialect and vendor. For example:

- IBM's current query tool uses a TIMESTAMPDIFF function, while pgAdmin uses an AGE function to the same effect.

Frequently Used Date Functions
`CURRENT_DATE`
`AGE`
`DATE_PART`

Date Functions | CURRENT DATE

CURRENT_DATE: Brings back the current date from your computer system.

Syntax: `CURRENT_DATE`

Example use case: You can use it to easily and quickly find out the current date (without having to look at a calendar!).

Date Functions | AGE

AGE returns the difference between two dates.

Syntax: `AGE ( date1, date2 )`

Example use case: When you need to calculate the duration of an event; for example, the number of days it takes to ship or return an order.

Date Functions | DATE_PART

DATE_PART returns a specific subfield of a date field.

Syntax: `DATE_PART ( text, timestamp )`

Examples:

- “text” values: ‘minute’, ‘hour’, ‘day’, ‘month’, ‘year’
- “timestamp” format: ‘2001-02-16 20:38:40’

Example:

`DATE_PART ('month', TIMESTAMP '2020-03-29' )` = **3** (March)

Date Functions | DATE_TRUNC

DATE_TRUNC returns a date truncated to the first day of the specified date part.

Syntax: `DATE_TRUNC ( text, timestamp )`

Examples:

- “text” values: 'minute', 'hour', 'day', 'month', 'year'
- “timestamp” format: '2001-02-16 20:38:40'

Example:

`DATE_TRUNC ( 'month', '2020-03-29' )` **= 2020-03-01 00:00:00**

Partner Exercise:

Selecting Dates

Most transaction-level databases will timestamp an entry by the millisecond, but generally, questions are asked at the day, month, or year level. We'll need to extract and aggregate this data. Work with your partner to return results for the following questions.

1. What **range of order dates** are we using in the Superstore data set?
    a. **Hint**: MIN/MAX functions will give us some useful information.
2. What **range of shipping dates**?
3. What **order date has the largest quantity of items sold?**

Partner Exercise:

Selecting Dates | Solutions

Answer 1: Range of order dates

```
SELECT
    MIN(order_date),
    MAX(order_date)
FROM orders;
```

Answer 2: Range of shipping dates

```
SELECT
    MIN(ship_date),
    MAX(ship_date)
FROM orders;
```

Answer 3: Order date with the largest quantity of items sold

```
SELECT order_date,
    SUM(quantity)
FROM orders
GROUP BY order_date
ORDER BY SUM(quantity) DESC;
```

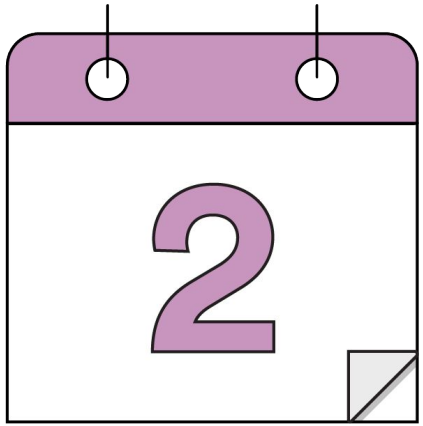
Re-Aggregating Dates

There are two ways to get the month from the date:

1. **DATE_TRUNC** function
2. **DATE_PART** function

The main difference between the two is that:

- DATE_TRUNC aggregates to the level of “date detail” you specify.
- DATE_PART aggregates data at a **combined** level of date detail.



Partner Exercise:

Re-Aggregating Dates | Let's Try It!

With your partner, run both of these queries and explain the difference between the results to each other. Be prepared to share your answer with the class.

```sql
SELECT
    SUM(quantity),
    DATE_TRUNC('month', order_date)
FROM orders
GROUP BY 2
ORDER BY 2;
```

```sql
SELECT
    SUM(quantity),
    DATE_PART('month', order_date)
FROM orders
GROUP BY 2
ORDER BY 2;
```

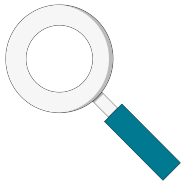

Discussion:

Date and Time Functions

Let’s take a look at the documentation linked below:

https://www.postgresql.org/docs/9.1/static/functions-datetime.html

What level of detail might you want to use for date timestamps in Superstore or your unit projects?



Guided Walk-Through:
Re-Aggregating Dates | TO_CHAR

How do you get rid of time when you don't need hour/minute?

We can ***wrap*** the **TO_CHAR** function around our **DATE_TRUNC** to clean up the time information.

Syntax:

```
SELECT
    TO_CHAR(DATE_TRUNC('month',order_date),'YYYY-MM-DD'),
    SUM(quantity)
FROM orders
GROUP BY 1;
```


Source: PostgresqlTutorial

Guided Walk-Through:
DATES and WHERE Clauses

Unlike in Excel, dates are non-numeric and are considered strings in SQL.

Because of this, dates need to be *wrapped* in tick marks/single quotes. Most SQL formats treat dates using the **YYYY-MM-DD** format.

Let's say we want to find all of the Superstore sales *after* a specific date. We would use:

```
SELECT *
FROM orders
WHERE order_date >= '2020-01-01'
```


Partner Exercise:

DATES and WHERE Clauses

Pair up again and determine the proper code to use for the following criteria.

1. The number of orders by month.
2. The number of orders that occurred on each Fourth of July.
3. The number of orders between Memorial Day and Labor Day 2019*.
4. The number of orders for Second Class shipments in January and February 2019, and Standard Class shipments in July and August 2019.

* ***Hint:*** *the dates in question 3 are holidays in the USA. You can google these dates.*

Partner Exercise:

DATES and WHERE Clauses | Solutions 1 & 2

1. The number of orders by month.

```
SELECT
     TO_CHAR(DATE_TRUNC('month', order_date), 'YYYY-MM') AS month, COUNT(DISTINCT
     order_id) AS number_of_orders
FROM orders
GROUP BY 1
ORDER BY 1;
```

2. The number of orders that occurred on each Fourth of July.

```
SELECT
     TO_CHAR(DATE_TRUNC('day', order_date), 'YYYY-MM-DD') AS day, COUNT(DISTINCT order_id)
     AS number_of_orders
FROM orders
WHERE TO_CHAR(DATE_TRUNC('day', order_date), 'MM-DD') = '07-04'
GROUP BY 1
ORDER BY 1;
```

Partner Exercise:

DATES and WHERE Clauses | Solution 3

3. The number of orders between Memorial Day and Labor Day 2019.

```sql
SELECT count(distinct order_id) AS num_orders
FROM orders
WHERE order_date BETWEEN '2019-05-27' AND '2019-09-02';
```


Partner Exercise:

DATES and WHERE Clauses | Solution 4

4. The number of orders for Second Class shipments in January and February 2019, and Standard Class shipments in July and August 2019.

```
SELECT ship_mode,
     TO_CHAR(DATE_TRUNC('month', order_date), 'YYYY-MM-DD'),
     COUNT(DISTINCT order_id) AS number_of_orders
FROM orders
WHERE
     (TO_CHAR(DATE_TRUNC('day', order_date), 'YYYY-MM-DD')
     BETWEEN '2019-01-01' and '2019-02-28' and ship_mode = 'Second Class')
     OR (TO_CHAR(DATE_TRUNC('day', order_date), 'YYYY-MM-DD')
     BETWEEN '2019-07-01' and '2019-08-31' and ship_mode = 'Standard Class')
GROUP BY 1,2;
```

Optional Practice

Solo Exercise:

Optional Practice | String Functions

Pretend our Superstore is having a furniture sale this weekend and you need to create place card stands for every furniture product. To make this easier, you use SQL!

1. For every unique product name, create a sentence that says “In the **X** department, the **Y** is on sale for 50% off!” where X is the sub-category and Y is the name of the product.
2. Your boss says printing place cards for each product is free if the length is less than 100 characters. Based on your question #1 answer, how many products have place cards with more than 100 characters?
3. You notice the Executive Impressions wall clock names are really long. For these products, omit the words ‘Executive Impressions from the place card. Write a query that returns the same verbiage as in #1 but without the vendor name for only Executive Impressions wall clocks . How many characters is it now and do they meet the 100 character limit?

Solo Exercise:

Optional Practice | String Functions

You are doing an analysis on certain customer types. Answer the following using **customers**:

4. Looking at the customer roster, create a column that only shows the customer's last name. Ensure there are no extra white spaces. *Hint: Use SUBSTRING, STRPOS, and TRIM.

5. Your team wants to issue a market research survey to a sample pool of customers. Using the LEFT function, find the customers and their segments who fall into the AD, AF, or AJ group.

6. Split the order_id into 3 separate columns. Limit results to 100 rows.

7. Superstore is updating its shipping categories and need to refresh this in our database. In the orders table, change 'Standard Class' to 'Economy Class' and show how many orders were placed in 2018 for each of the ship mode classes.

Solo Exercise:

Optional Practice | Date Functions

Use your tables to answer the following questions:

8. By day, how many orders were there between Thanksgiving 2019 (Nov 28, 2019) and New Years Day 2020 (Jan 1, 2020)?

9. For each ship mode, what is the highest # of days it takes to ship a product after it has been ordered for products priced higher than $1000?

10. Using CASE WHEN logic, sum up sales, quantity, and profit by each calendar quarter of 2018. *Hint: Dates are in YYYY-MM-DD format and you can use BETWEEN.

11. Complete #10 again using the DATE_TRUNC or DATE_PART functions instead of CASE WHEN.

Solo Exercise:

Optional Practice | **Solutions 1, 2, & 3**

1. `SELECT DISTINCT CONCAT('In the ', sub_category, ' department, the ', product_name, ' is on sale for 50% off') as sale_announce FROM products ORDER BY 1;` **Returns 3688 Rows**

2. `SELECT COUNT(DISTINCT LENGTH(CONCAT('In the ', sub_category, ' department, the ', product_name, ' is on sale for 50% off'))) as sale_announce_over100 FROM products WHERE LENGTH(CONCAT('In the ', sub_category, ' department, the ', product_name, ' is on sale for 50% off')) > 100 ORDER BY 1;` **Count is 53 have a length > 100**

3.
```
SELECT DISTINCT CONCAT('In the ', sub_category, ' department, the ',
REPLACE(product_name,'Executive Impressions ','') ,' is on sale for 50% off'),
LENGTH(CONCAT('In the ', sub_category, ' department, the ', REPLACE(product_name,'Executive
Impressions ','') ,' is on sale for 50% off')) as sale_announce FROM products WHERE
product_name ILIKE '%Executive Impressions%' AND product_name ILIKE '%wall clock%' ORDER BY 1;
```
All but 1 Executive Impressions Wall Clock are now under 100 characters. The exception is the Contract Wall Clock with Quartz Movement

Solo Exercise:

Optional Practice | **Solutions 4, 5, 6 & 7**

4. `SELECT TRIM(SUBSTRING(customer_name, STRPOS(customer_name, ' '))) as last_name FROM customers;` **Returns 1590 last name**

5. `SELECT DISTINCT LEFT(customer_id, 2) as customer_group, customer_name, segment FROM customers WHERE LEFT(customer_id, 2) IN ('AD','AF','AJ');` **Returns new column customer_group and 7 rows**

6. `SELECT order_id, LEFT(order_id,2) as col1, SPLIT_PART(order_id,'-',2) as col2,SPLIT_PART(order_id,'-',3) as col3 FROM orders LIMIT 100;` **Returns 3 new columns and 100 rows**

7. `SELECT CASE WHEN ship_mode = 'Standard Class' THEN 'Economy Class' ELSE ship_mode END as ship_mode_class, COUNT(*) FROM orders WHERE DATE_PART('year',order_date) = 2018 GROUP BY 1;` **Second Class 92794, Economy Class 111545, First Class 15175, Same Day 5787**

Solo Exercise:

Optional Practice | **Solutions 8 & 9**

8.
```
SELECT COUNT(DISTINCT order_id) as unique_orders FROM orders WHERE order_date BETWEEN
'2019-11-28' AND '2020-01-01';
```

Returns 68393

9.
```
SELECT ship_mode, MAX(AGE(ship_date,order_date)) as max_days_to_ship FROM orders
WHERE sales > 1000 GROUP BY 1 ORDER BY 2 DESC ;
```

Standard Class 7 days, Same Day 5 days, Second Class 5 days, First Class 3 days

Solo Exercise:

Optional Practice | **Solutions 10 & 11**

10.
```
SELECT CASE WHEN order_date BETWEEN '2018-01-01' AND
'2018-03-31' THEN 'Q1_2018' WHEN order_date BETWEEN
'2018-04-01' AND '2018-06-30' THEN 'Q2_2018' WHEN
order_date BETWEEN '2018-07-01' AND '2018-09-30' THEN
'Q3_2018' WHEN order_date BETWEEN '2018-10-01' AND
'2018-12-31' THEN 'Q4_2018' END as quarter, SUM(sales)
as sales, SUM(quantity) as qty, SUM(profit) as profit
FROM orders WHERE DATE_PART('year', order_date) = 2018
GROUP BY 1 ORDER BY 1;
```

quarter text	sales numeric	qty bigint	profit numeric
Q1_2018	9658301.36	125569	76403.56
Q2_2018	14484830.01	203262	100383.91
Q3_2018	19602341.04	286845	111806.38
Q4_2018	24521548.60	352405	153783.73

11.
```
SELECT DATE_PART('quarter', order_date) as quarter,
SUM(sales) as sales, SUM(quantity) as qty, SUM(profit)
as profit FROM orders WHERE DATE_PART('year',
order_date) = 2018 GROUP BY 1 ORDER BY 1;
```

quarter double precision	sales numeric	qty bigint	profit numeric
1	9658301.36	125569	76403.56
2	14484830.01	203262	100383.91
3	19602341.04	286845	111806.38
4	24521548.60	352405	153783.73

Querying Data With SQL

Wrapping Up

Recap

Today in class, we...

- Practiced the concepts and syntax of advanced JOINs such as EXCEPT, FULL, and OUTER.
- Handled NULLs in SQL.
- Applied string, math, and date functions in SQL to prepare and analyze data.
- Practiced writing SQL queries with advanced functions to solve business problems.

Looking Ahead

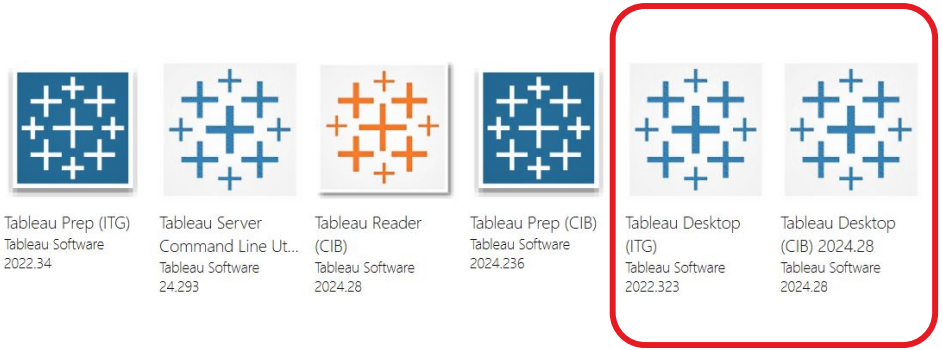
Optional Homework:

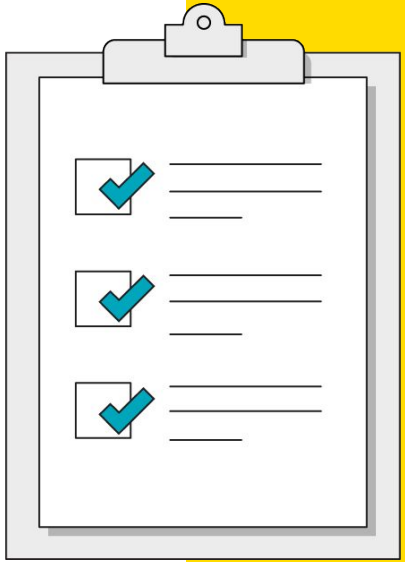
- Optional myGA Lesson: **Intermediate SQL** (unit)

Up Next: Project prep

Before You Go...

- Check to make sure you have **Tableau** installed for tomorrow.
- If you have any problems with your installation, let us know!





Additional Resources

- What are the SQL Database Functions?
- PostgreSQL functions — from Neon
- PostgreSQL functions — from Tech On The Net
- PostgreSQL functions — from w3resource
- SQL Server: Functions — Listed by Category — from TechOnTheNet
- Useful SQL Functions — from TutorialPoints


GA