SELECT $\boxed{...}$ FROM table_name1 $\boxed{...}$

Specifying output columns

Filtering output rows

CTE (Common Table Expressions)

CTE allows you to grab a subset of data and store that data with a name, which you can then select from and perform more operations on.

A CTE allows you to define a temporary named result set that available temporarily in the execution scope of a statement such as SELECT, INSERT, UPDATE, DELETE, or MERGE.

Syntax

```
WITH expression_name[(column_name [,...])]
AS
   (CTE_definition)
SQL_statement;
```

- First, specify the expression name (**expression_name**) to which you can refer later in a query.
- Next, specify a list of comma-separated columns after the **expression_name**. The number of columns must be the same as the number of columns defined in the **CTE_definition**.
- Then, use the AS keyword after the expression name or column list if the column list is specified.
- After, define a SELECT statement whose result set populates the common table expression.
- Finally, refer to the common table expression in a query (**SQL_statement**) such as SELECT, INSERT, UPDATE, DELETE, or MERGE

Let's say you've been asked to calculate the average time between transactions by a particular user. You have a table called transactions that contains a username and the time of the transaction.

When you get a difficult question like the one above, take a minute and ask yourself **what the ideal table would have to look like to allow you to answer your question with one SELECT statement**.

In the above example, the ideal table was one that included one record for each transaction, and a column that gave the time of the next transaction.

Example

Id	FirstName	LastName	Education	Occupation	YearlyIncome	Sales
1	John	Yang	Bachelors	Professional	115000	3578.27
2	Rob	Johnson	Bachelors	Management	105000	3399.99
3	Ruben	Torres	Partial College	Skilled Manual	50000	699.0982
4	Christy	Zhu	Bachelors	Professional	105000	3078.27
5	Rob	Huang	High School	Skilled Manual	85000	2319.99
6	John	Ruiz	Bachelors	Professional	70000	539.99
7	Tutorial	Gateway	Masters Degree	Management	105000	2320.49
8	Christy	Mehta	Partial High School	Clerical	50000	24.99
9	Rob	Verhoff	Partial High School	Clerical	45000	24.99
10	Christy	Carlson	Graduate Degree	Management	95000	2234.99
11	Gail	Erickson	Education	Professional	115000	4319.99
12	Barry	Johnson	Education	Management	105000	4968.59
13	Peter	Krebs	Graduate Degree	Clerical	50000	59.53
14	Greg	Alderson	Partial High School	Clerical	45000	23.5

```
WITH Total_Sale_CTE AS (
    SELECT Occupation,
           Education,
           SUM(YearlyIncome) AS Income,
           SUM(Sales) AS Sale
    FROM employee_table
    GROUP BY Education, Occupation)

SELECT * FROM Total_Sale_CTE
```

	Occupation	Education	Income	Sale
1	Clerical	Graduate Degree	50000	59.53
2	Clerical	Partial High School	140000	73.48
3	Management	Bachelors	105000	3399.99
4	Management	Education	105000	4968.59
5	Management	Graduate Degree	95000	2234.99
6	Management	Masters Degree	105000	2320.49
7	Professional	Bachelors	290000	7196.53
8	Professional	Education	115000	4319.99
9	Skilled Manual	High School	85000	2319.99
10	Skilled Manual	Partial College	50000	699.0982

Filter, Aggregate, Join

Generally, you want to follow these steps with your string of CTEs: filter, aggregate, join. Filter using WHERE, aggregate using GROUP BY, and join using JOIN.

By filtering and aggregating your data before joining, you write the most efficient SQL. Joins are expensive to process so you want the fewest possible rows before joining two tables together. Sometimes aggregating first won't be possible, but usually you'll be able to limit the size of the tables you're joining with at least a WHERE clause or two.

It's important to note that if you have a JOIN and a WHERE clause in the same CTE, SQL processes the JOIN first. In other words, the following (to the left) is very inefficient, because the entirety of your tables would be joined together and only then filtered to data after 8/1/2017:

```
--inefficient
SELECT *
FROM table_a a
INNER JOIN table_b b
ON a.username = b.username
WHERE a.day >= '2017-08-01'
```

```
--efficient
WITH a AS (
        SELECT *
        FROM table_a
        WHERE day >= '2017-08-01')

b AS (
        SELECT *
        FROM table_b
        WHERE day >= '2017-08-01')

SELECT *
FROM a
INNER JOIN b
ON a.username=b.username;
```

Window Functions

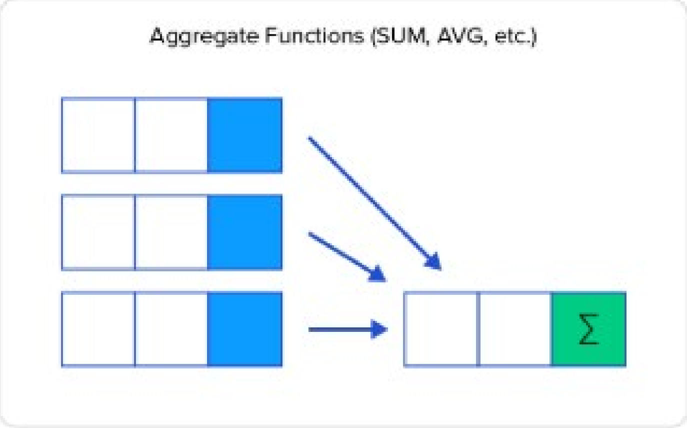
Like a GROUP BY clause, **window functions** separate your table into several chunks and operates on each chunk individually. But unlike GROUP BY, the rows are not combined. **Window functions** operate on a set of rows and return a single value for each row from the underlying query.

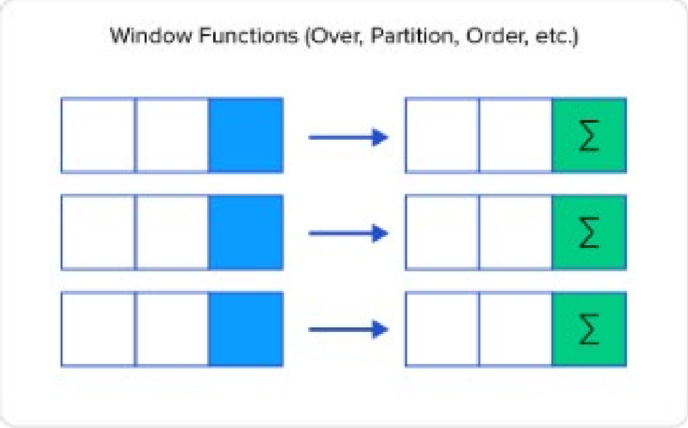
Syntax

```
AGG_FUNC ( scalar_expression )
OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

SUM, COUNT, AVG, etc.

RANK, LAG, FIRST_VALUE, etc.





AGG_FUNC() should be some aggregation function like SUM, COUNT, AVG, RANK , LAG, FIRST_VALUE, etc. (Note, some functions might have additional parameters)

scalar_expression is an expression evaluated against the value of the first row of the ordered partition of a result set. The scalar_expression can be a column, subquery, or expression that evaluates to a single value. It cannot be a window function. For some AGG_FUNC() it is not required.

OVER() specifies the window for which the aggregation is performed. If no argument is provided, the aggregation is calculated on all the rows.

The PARTITION BY clause distributes rows of the result set into partitions to which the AGG_FUNC() function is applied. If you skip the PARTITION BY clause, the AGG_FUNC() function will treat the whole result set as a single partition.

Some of the functions require ORDER BY, and it's not supported by the others. When the order of the rows is important when applying the calculation, the ORDER BY is required.

Window Functions, Examples

some_table

Order_id	Order_date	Customer_name	City	Order_amount
1001	04/01/2017	David Smith	GuildFord	10000
1002	04/02/2017	David Jones	Arlington	20000
1003	04/03/2017	John Smith	Shalford	5000
1004	04/04/2017	Michael Smith	GuildFord	15000
1005	04/05/2017	David Williams	Shalford	7000
1006	04/06/2017	Paum Smith	GuildFord	25000
1007	04/10/2017	Andrew Smith	Arlington	15000
1008	04/11/2017	David Brown	Arlington	2000
1009	04/20/2017	Robert Smith	Shalford	1000
1010	04/25/2017	Peter Smith	GuildFord	500

```
SELECT order_id,
       city,
       order_amount,
       SUM(order_amount) OVER()
       AS total_volume
FROM some_table
```

Order_id	City	Order_amount	Total_volume
1001	GuildFord	10000	100500
1002	Arlington	20000	100500
1003	Shalford	5000	100500
1004	GuildFord	15000	100500
1005	Shalford	7000	100500
1006	GuildFord	25000	100500
1007	Arlington	15000	100500
1008	Arlington	2000	100500
1009	Shalford	1000	100500
1010	GuildFord	500	100500

```
SELECT order_id,
       city,
       order_amount,
       SUM(order_amount) OVER(
       PARTITION BY city
       ) AS total_volume
FROM some_table
```

Order_id	City	Order_amount	Total_volume
1002	Arlington	20000	37000
1007	Arlington	15000	37000
1008	Arlington	2000	37000
1001	GuildFord	10000	50500
1004	GuildFord	15000	50500
1006	GuildFord	25000	50500
1010	GuildFord	500	50500
1003	Shalford	5000	13000
1005	Shalford	7000	13000
1009	Shalford	1000	13000

Window Functions, Examples, p.2

RANK()

```
SELECT order_id,
       city,
       order_amount,
       RANK() OVER(
       PARTITION BY city
       ORDER BY order_amount
       ) AS amount_rank
FROM some_table
```

Order_id	City	Order_amount	Amount_rank
1008	Arlington	2000	1
1007	Arlington	15000	2
1002	Arlington	20000	3
1001	GuildFord	10000	1
1004	GuildFord	15000	2
1010	GuildFord	15000	2
1006	GuildFord	25000	4
1003	Shalford	5000	1
1009	Shalford	5000	1
1005	Shalford	7000	3

FIRST_VALUE()

```
SELECT order_id,
       city,
       order_amount,
       FIRST_VALUE(city) OVER(
       ORDER BY order_amount
       ) AS lowest_amount_city
FROM some_table
```

Order_id	City	Order_amount	Lowest_amount_city
1008	Arlington	2000	Arlington
1003	Shalford	5000	Arlington
1009	Shalford	5000	Arlington
1005	Shalford	7000	Arlington
1001	GuildFord	10000	Arlington
1004	GuildFord	15000	Arlington
1007	Arlington	15000	Arlington
1010	GuildFord	15000	Arlington
1002	Arlington	20000	Arlington
1006	GuildFord	25000	Arlington

DENSE_RANK()

```
SELECT order_id,
       city,
       order_amount,
       DENSE_RANK() OVER(
       PARTITION BY city
       ORDER BY order_amount
       ) AS amount_rank
FROM some_table
```

Order_id	City	Order_amount	Amount_rank
1008	Arlington	2000	1
1007	Arlington	15000	2
1002	Arlington	20000	3
1001	GuildFord	10000	1
1004	GuildFord	15000	2
1010	GuildFord	15000	2
1006	GuildFord	25000	3
1003	Shalford	5000	1
1009	Shalford	5000	1
1005	Shalford	7000	2

LAG()

```
SELECT order_id,
       city,
       order_amount,
       LAG(order_amount, 1, NULL) OVER(
       ORDER BY order_date
       ) AS previous_purchase_amount
FROM some_table
```

Order_id	City	Order_amount	Previous_purchase_amount
1001	GuildFord	10000	Null
1002	Arlington	20000	10000
1003	Shalford	5000	20000
1004	GuildFord	15000	5000
1005	Shalford	7000	15000
1006	GuildFord	25000	7000
1007	Arlington	15000	25000
1008	Arlington	2000	15000
1009	Shalford	5000	2000
1010	GuildFord	15000	5000

Union

UNIONs are just the vertical version of JOINs: whereas JOINs combine tables or CTEs horizontally using a join key, a UNION just **stacks the tables on top of each other** to form one table containing all the rows from the two original tables. The requirement for this is that the two tables that are being UNION-ed have the same exact columns — otherwise there would be no way to logically combine them.

Syntax

```
query_1

UNION

query_2
```

- The number and the order of the columns must be the same in both queries.
- The data types of the corresponding columns must be the same or compatible.

UNION allows you to combine results of two SELECT statements into a single result set which includes all the rows that belongs to the SELECT statements in the union.

An example of when a UNION might be useful is when you have separate tables for two types of transactions, but want a single query to tell you how many of each type of transaction you have.

```
WITH sales AS (
        SELECT 'sale' AS type
        FROM sale_transactions
        WHERE day >= '2017-09-01'),

buys AS (
        SELECT 'buy' AS type
        FROM buy_transactions
        WHERE day >= '2017-09-01'),

unioned AS (
        SELECT type
        FROM buys
        UNION ALL
        SELECT type
        FROM sales)

SELECT type, count(1) AS num_transactions
FROM unioned
GROUP BY type;
```

CASE

CASE statements are exactly the same as the ifelse() functions in environments like R and Excel. They're useful for mapping from one set of predefined values to another.

Syntax

```
CASE input
  WHEN e1 THEN r1
  WHEN e2 THEN r2
  ...
  WHEN en THEN rn
  [ ELSE re ]
END
```

The simple CASE expression compares the input expression (input) to an expression (ei) in each WHEN clause for equality. If the input expression equals an expression (ei) in the WHEN clause, the result (ri) in the corresponding THEN clause is returned.

If the input expression does not equal to any expression and the ELSE clause is available, the CASE expression will return the result in the ELSE clause (re).

In case the ELSE clause is omitted and the input expression does not equal to any expression in the WHEN clause, the CASE expression will return NULL.

```sql
SELECT
    order_status,
    COUNT(order_id) order_count
FROM
    sales.orders
WHERE
    YEAR(order_date) = 2018
GROUP BY
    order_status;
```

order_status	order_count
1	62
2	63
3	13
4	154

```sql
SELECT
    CASE order_status
        WHEN 1 THEN 'Pending'
        WHEN 2 THEN 'Processing'
        WHEN 3 THEN 'Rejected'
        WHEN 4 THEN 'Completed'
    END AS order_status,
    COUNT(order_id) order_count
FROM
    sales.orders
WHERE
    YEAR(order_date) = 2018
GROUP BY
    order_status;
```

order_status	order_count
Pending	62
Processing	63
Rejected	13
Completed	154

```sql
SELECT
    SUM(CASE
            WHEN order_status = 1
            THEN 1
            ELSE 0
        END) AS 'Pending',
    SUM(CASE
            WHEN order_status = 2
            THEN 1
            ELSE 0
        END) AS 'Processing',
    SUM(CASE
            WHEN order_status = 3
            THEN 1
            ELSE 0
        END) AS 'Rejected',
    SUM(CASE
            WHEN order_status = 4
            THEN 1
            ELSE 0
        END) AS 'Completed',
    COUNT(*) AS Total
FROM
    sales.orders
WHERE
    YEAR(order_date) = 2018;
```

Pending	Processing	Rejected	Completed	Total
62	63	13	154	292

HAVING

- HAVING filters records that work on summarized GROUP BY results.
- HAVING applies to summarized group records, whereas WHERE applies to individual records.
- Only the groups that meet the HAVING criteria will be returned.
- HAVING requires that a GROUP BY clause is present.
- WHERE and HAVING can be in the same query.

Problem: List the number of customers in each country. Only include countries with more than 10 customers.

```
SELECT COUNT(Id), Country
  FROM Customer
 GROUP BY Country
HAVING COUNT(Id) > 10
```

Results: 3 records

Count	Country
11	France
11	Germany
13	USA

CROSS JOIN vs INNER JOIN … ON … != …

tabl

col
A
B
C

```
SELECT
  t1.col AS col_1,
  t2.col AS col_2
FROM tabl t1
CROSS JOIN tabl t2
```

col_1	***col_2***
A	A
A	B
A	C
B	A
B	B
B	C
C	A
C	B
C	C

```
SELECT
  t1.col AS col_1,
  t2.col AS col_2
FROM tabl t1
INNER JOIN tabl t2
ON t1.col != t2.col
```

col_1	***col_2***
A	B
A	C
B	A
B	C
C	A
C	B

EXAMPLES

employees table

columns	types
id	int
first_name	varchar
last_name	varchar
salary	int
department_id	int

departments table

columns	types
id	int
name	varchar

Given the tables above, select the top 3 departments by the highest percentage of employees making over 100K in salary and have at least 10 employees.

Example output:

> 100K %	department name	number of employees
90%	engineering	25
50%	marketing	50
12%	sales	12

```
SELECT
   d.name,
   CAST(SUM(CASE
               WHEN salary > 100000 THEN 1
               ELSE 0
          END)
      AS DECIMAL) / COUNT(*) AS percent_employees_over_100K
FROM departments AS d
LEFT JOIN employees AS e
   ON d.id = e.department_id
GROUP BY 1
HAVING COUNT(*) >= 10
ORDER BY 2 DESC
LIMIT 3
```

EXAMPLES

employees table

columns	types
id	int
first_name	varchar
last_name	varchar
salary	int
department_id	int

departments table

columns	types
id	int
name	varchar

Let's say due to an ETL error, the employee table instead of updating the salaries every year when doing compensation adjustments, did an insert instead. The head of HR still needs the current salary of each employee. Write a query to get the current salary for each employee.

Assume no duplicate combination of first and last names. (I.E. No two John Smiths)

```
SELECT e.first_name, e.last_name, e.salary
FROM employees AS e
INNER JOIN (
   SELECT first_name, last_name, MAX(id) AS max_id
   FROM employees
   GROUP BY 1,2
) AS m
   ON e.id = m.max_id
```

The first step would be to remove duplicates. Given we know there aren't any duplicate first and last name combinations, we can remove duplicates from the employees table by just grouping by first and last name and getting the maximum id from the table which would be the last entry and the most up to date salary.

EXAMPLES

`attribution` table

column	type
id	int
created_at	datetime
session_id	int
channel	varchar
conversion	boolean

`user_sessions` table

column	user_id
session_id	int
user_id	int

The attribution table logs each user visit where a user comes onto their site to go shopping. If *conversion* = 1, then on that session visit the user converted and bought an item. The *channel* column represents which advertising platform the user got to the shopping site on that session.
The **`user_sessions`** table maps each session visit back to the user.

First touch attribution is defined as the channel to which the converted user was associated with when they first discovered the website. Calculate the first touch attribution for each user_id that converted.

```
WITH conv_users AS (
   SELECT b.user_id,
             a.channel,
             RANK() OVER (
                PARTITION BY b.user_id
                ORDER BY created_at
             ) AS rank
   FROM attribution a
             INNER JOIN user_session b
             ON a.session_id = b.session_id
   WHERE a.conversion = true
)

SELECT user_id, channel
FROM conv_users
WHERE rank = 1
```

EXAMPLES

transactions table

column	type
user_id	int
created_at	datetime
product_id	int
quantity	int
price	float

Given a transaction table of product purchases, write a query to get the number of customers that were upsold by purchasing additional products.

Note that if the customer purchased two things on the same day that does not count as an upsell. Each row in the transactions table also represents an individual user product purchase.

```
WITH unique_user_date_pairs AS (
   SELECT user_id,
               DATE(created_at)
    FROM transactions
    GROUP BY 1,2
)

SELECT COUNT(*)
FROM (
   SELECT user_id
   FROM unique_user_date_pairs
   GROUP BY 1
   HAVING COUNT(*) > 1
)
```

Opt. 1

```
SELECT COUNT(t.user_id)
FROM (
  SELECT user_id ,
             RANK() OVER (
                PARTITION BY user_id
                ORDER BY DATE(created_at::TIMESTAMP)
             ) AS rnk_purchases
  FROM transactions
)
WHERE rnk_purchases > 1
```

Opt. 2

```
SELECT COUNT(user_id)
FROM (
  SELECT user_id,
             COUNT(DISTINCT(DATE(created_at))) AS ct
  FROM transactions
  GROUP BY user_id
)
WHERE ct > 1
```

Opt. 3

EXAMPLES

columns	type
id	int
user_id	int
item	varchar
created_at	datetime
revenue	float

Given the revenue transaction table above that contains a user_id, created_at timestamp, and transaction revenue, write a query that finds the third purchase of every user.

```
SELECT user_id, item
FROM (
  SELECT
      user_id,
      item,
      ROW_NUMBER() OVER (
        PARTITION BY user_id,
        ORDER BY created_at ASC
      ) AS row_num
  FROM transactions
)
WHERE row_num = 3
```

EXAMPLES

\`users\` table

columns	type
id	int
name	varchar
neighborhood_id	int
joined_date	datetime

\`neighborhoods\` table

columns	type
id	int
name	varchar
city_id	int

Given a users table with information about a user on which neighborhood they live in and a corresponding neighborhoods of all the neighborhoods in the U.S., write a query that returns all of the neighborhoods that have 0 users.

```
SELECT n.name
FROM
   neighborhoods AS n
   LEFT JOIN users AS u
   ON n.id = u.neighborhood_id
WHERE u.id IS NULL
```

Whenever the question asks about finding values with zero users, employees, posts, etc. immediately think LEFT JOIN

EXAMPLES

`scores` table

column	type
id	integer
student	varchar
score	integer

Example:

input

id	student	score
1	Jack	1700
2	Alice	2010
3	Miles	2200
4	Scott	2100

output

one_student	other_student	score_diff
Alice	Scott	90

Given a table of students and their SAT test scores, write a query to return the two students with the closest test scores with the score difference.

Assume a random pick if there are multiple students with the same score difference.

```
SELECT
   s1.student AS one_student,
   s2.student AS other_student,
   ABS(s1.score - s2.score) AS score_diff
FROM
   scores AS s1
   INNER JOIN scores AS s2
      ON s1.id != s2.id
ORDER BY 3 ASC
LIMIT 1
```

EXAMPLES

```
employees                                projects
+---------------+---------+              +---------------+---------+
| id            | int     |<----+     +->| id            | int     |
| first_name    | varchar |     |     |  | title         | varchar |
| last_name     | varchar |     |     |  | start_date    | date    |
| salary        | int     |     |     |  | end_date      | date    |
| department_id | int     |--+  |     |  | budget        | int     |
+---------------+---------+  |  |     |  +---------------+---------+
                             |  |     |
departments                  |  |     |  employees_projects
+---------------+---------+  |  |     |  +---------------+---------+
| id            | int     |<-+  |     +--| project_id    | int     |
| name          | varchar |     +--------| employee_id   | int     |
+---------------+---------+              +---------------+---------+
```

Over budget on a project is defined when the salaries, prorated to the day, exceed the budget of the project.

For example, if Alice and Bob both combined income make 200K and work on a project of a budget of 50K that takes half a year, then the project is over budget given 0.5 * 200K = 100K > 50K.

Write a query to select all projects that are over budget. Assume that employees only work on one project at a time.

```sql
SELECT
    title,
    CASE WHEN
        CAST(project_days AS DECIMAL)/365 * total_salary > budget
        THEN 'overbudget'
        ELSE 'within budget'
    END AS project_forecast
FROM (
    SELECT
        title,
        DATEDIFF(end_date, start_date) AS project_days,
        budget,
        SUM(COALESCE(salary,0)) AS total_salary,
    FROM projects AS p
    LEFT JOIN employees_projects AS ep
        ON p.id = ep.project_id
    LEFT JOIN employees AS e
        ON e.id = ep.employee_id
    GROUP BY 1,2,3
)
```

Notice how we're left joining `projects` to both `employee_projects` and `employees`. This is due to the effect that if there exists no employees on a project, we still need to define it as overbudget and setting the salaries as 0.

We're also grouping by title, project_days, and budget, so that we can get the total sum. Given that each of title, project_days, and budget are distinct for each project, we can do the group by without a fear of duplication in our SUM.