The Ultimate Guide to Mastering LeetCode SQL Problems

Zoe Dinh Follow

12 min read · Mar 25, 2025

SQL queries operate similarly to relational algebraic expressions, utilizing filters, conditions, and set-based operations to efficiently retrieve and manipulate data. To succeed in SQL challenges on LeetCode, it's crucial to master essential functions and concepts. This article also outlines a step-by-step approach to tackling SQL problems effectively.

SQL Functions Explanation

1. Basic SQL Commands

SELECT, FROM, WHERE — Retrieve specific data

To retrieve **female** student **names** from table: STUDENT(SID, SNAME, SEX, GPA)

```sql
SELECT SNAME
FROM STUDENT
WHERE SEX = 'Female';
```

DISTINCT — Remove duplicates of returned rows

To retrieve **distinct female** student **names** from table: STUDENT(SID, SNAME, SEX, GPA)

```sql
SELECT DISTINCT SNAME
FROM STUDENT
WHERE SEX = 'Female'
```

ORDER BY — Sort results (ASC / DESC), default ASC

To retrieve **distinct female** student **names** in **descending** order **(Z->A)** from table: STUDENT(SID, SNAME, SEX, GPA)

```sql
SELECT DISTINCT SNAME
FROM STUDENT
WHERE SEX = 'Female'
ORDER BY SNAME DESC;
```

LIMIT — Restrict the number of returned rows

To retrieve **five female** student **names** from table: STUDENT(SID, SNAME, SEX, GPA)

```
SELECT SNAME
FROM STUDENT
WHERE SEX = 'Female'
LIMIT 5;
```

2. Filtering and Conditional Logic

WHERE with AND, OR, NOT

AND: To retrieve **female** student **names** with **4.0 GPA** from table: STUDENT(SID, SNAME, SEX, GPA)

```
SELECT SNAME
FROM STUDENT
WHERE SEX = 'Female' AND GPA = 4.0;
```

OR: To retrieve **female** student **names** with **4.0 GPA or 3.0 GPA** from table: STUDENT(SID, SNAME, SEX, GPA)

```
SELECT SNAME
FROM STUDENT
WHERE SEX = 'Female'
AND (GPA = 4.0 OR GPA = 3.0);
```

NOT: To retrieve **female** student **names** without **4.0 GPA or 3.0 GPA** from table: STUDENT(SID, SNAME, SEX, GPA)

```sql
SELECT SNAME
FROM STUDENT
WHERE SEX = 'Female' AND NOT (GPA = 4.0 OR GPA = 3.0);
```

BETWEEN — Filter within a range, including both the lower and upper bounds.

To retrieve **female** student **names** with **GPA between 3.0 and 4.0** from table: STUDENT(SID, SNAME, SEX, GPA)

```sql
SELECT SNAME
FROM STUDENT
WHERE SEX = 'Female'
AND GPA BETWEEN 3.0 AND 4.0;
```

LIKE — Pattern matching (% stands for *any string*, _ stands for *any single character*)

To retrieve students with **name** start with “Zoe” or “Zoey” from table: STUDENT(SID, SNAME, SEX, GPA)

```sql
SELECT *
FROM STUDENT
```

```
WHERE SNAME LIKE 'Zoe%';
```

To retrieve students with **name** start with any single character than “oe” like “Zoe” or “Joey” from table: STUDENT(SID, SNAME, SEX, GPA)

```
SELECT *
FROM STUDENT
WHERE SNAME LIKE '_oe%';
```

IN — Check against multiple values

To retrieve students with **name** “Zoe” or “Zoey” from table: STUDENT(SID, SNAME, SEX, GPA)

```
SELECT *
FROM STUDENT
WHERE SNAME IN ('Zoe', 'Zoey');
```

CASE — Conditional logic in queries

Retrieve students with the SEX column displayed as 0 for males and 1 for females from the table: STUDENT(SID, SNAME, SEX, GPA)

```
SELECT SID, SNAME, GPA,
WHEN
    CASE SEX = 'Male' THEN 0
    CASE SEX = 'Female' THEN 1
    ELSE NULL
END AS SEX
FROM STUDENT;
```

3. Aggregation Functions

COUNT() — Count rows

To count how many students with **name** start with “Zoe” or “Zoey” from table: STUDENT(SID, SNAME, SEX, GPA)

```
SELECT COUNT(*)
FROM STUDENT
WHERE SNAME LIKE 'Zoe%';
```

SUM(), AVG() — Calculate totals and averages

To calculate the **total** number of products sold from the table: TRANSACTION(TID, PID, QUANTITY)

```
SELECT SUM(QUANTITY)
FROM TRANSACTION;
```

To calculate the **average** number of products sold per order from the table: TRANSACTION(TID, PID, QUANTITY)

```
SELECT AVG(QUANTITY)
FROM TRANSACTION;
```

MIN(), MAX() — Find minimum and maximum values

To calculate the **min** number of products sold per order from the table: TRANSACTION(TID, PID, QUANTITY)

```
SELECT MIN(QUANTITY)
FROM TRANSACTION;
```

To calculate the **max** number of products sold per order from the table: TRANSACTION(TID, PID, QUANTITY)

```
SELECT MAX(QUANTITY)
FROM TRANSACTION;
```

GROUP BY — Group data for aggregation

To calculate the **total** number of products sold for **each product** from the table: TRANSACTION(TID, PID, QUANTITY)

```
SELECT PID, SUM(QUANTITY) AS PRODUCT_SUM
FROM TRANSACTION
```

```
GROUP BY PID;
```

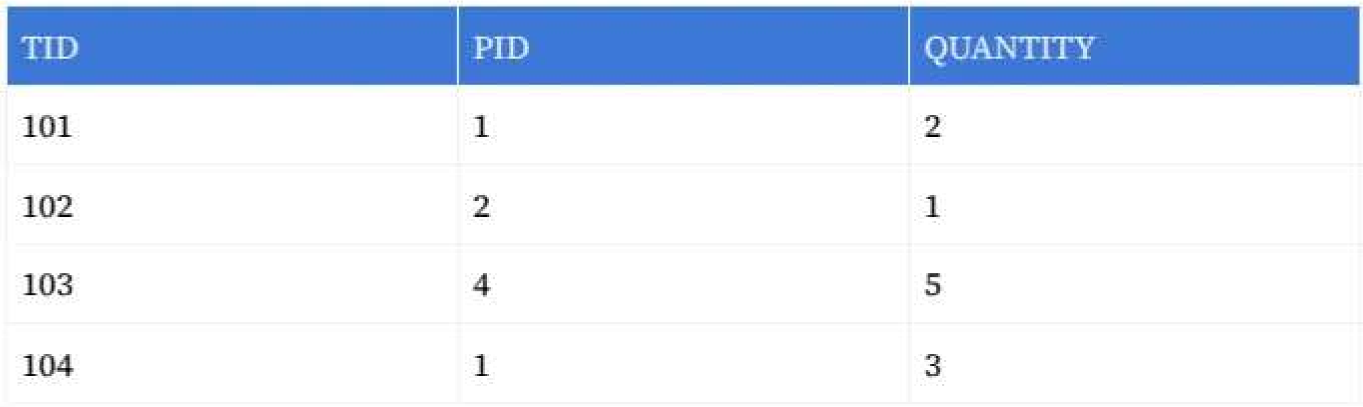
HAVING — Filter aggregated results (like WHERE but for groups)

To calculate the **total quantity** of products sold for **each product**, including only those with **a total quantity of at least 1000** from the table: TRANSACTION(TID, PID, QUANTITY)

```
SELECT PID, SUM(QUANTITY) AS PRODUCT_SUM
FROM TRANSACTION
GROUP BY PID
HAVING SUM(QUANTITY) >= 1000;
```

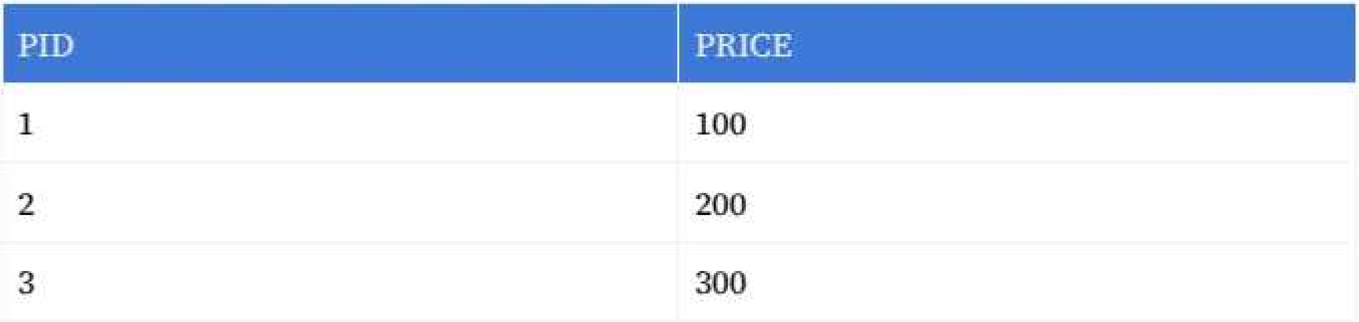
4. Joins and Subqueries

Assume there are two tables:



TID	PID	QUANTITY
101	1	2
102	2	1
103	4	5
104	1	3

TRANSACTION table



PID	PRICE
1	100
2	200
3	300

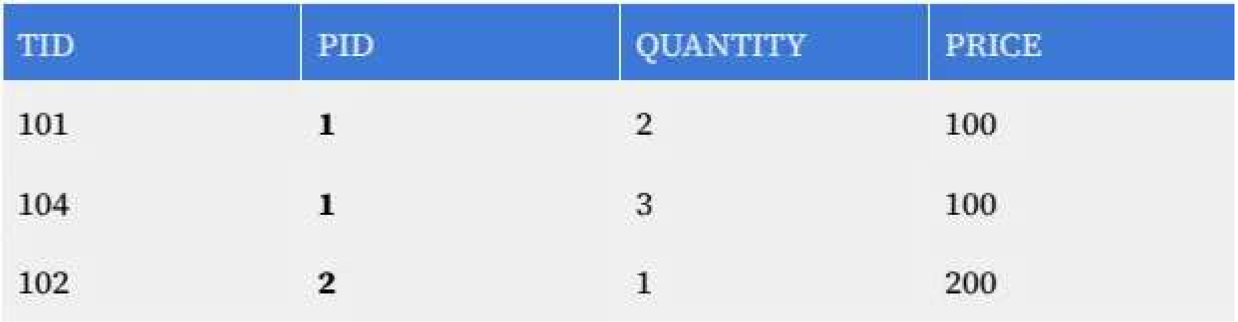
PRODUCT table

INNER JOIN (short as JOIN) — Retrieve matching records from *both tables*

To retrieve transaction details along with the corresponding product price by **matching records from two tables.**

```
SELECT TRANSACTION.TID, TRANSACTION.PID, TRANSACTION.QUANTITY, PRODUCT.PRICE
FROM TRANSACTION T
INNER JOIN PRODUCT
ON PRODUCT.PID = TRANSACTION.PID;
```

Since we use PID (specified in the ON clause) to match the two tables, the common PID values are **1** and **2**, and the results are as follows.



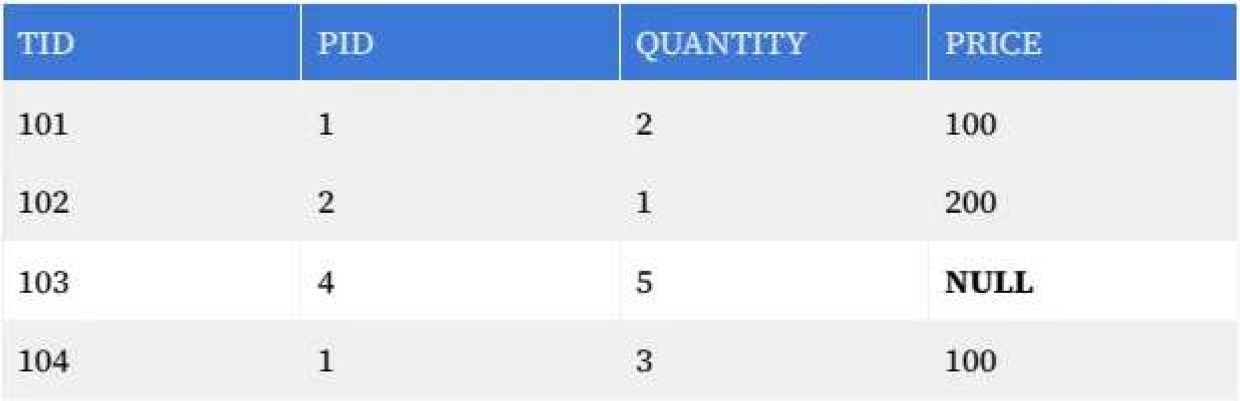
TID	PID	QUANTITY	PRICE
101	**1**	2	100
104	**1**	3	100
102	**2**	1	200

LEFT JOIN — Include unmatched rows from the *left* table

To retrieve transaction details along with the corresponding product price, including **all transaction** records and matching **product prices where available.**

```
SELECT TRANSACTION.TID, TRANSACTION.PID, TRANSACTION.QUANTITY, PRODUCT.PRICE
FROM TRANSACTION
LEFT JOIN PRODUCT
ON PRODUCT.PID = TRANSACTION.PID;
```

Since we use a LEFT JOIN, **all transaction** records are included in the result. The join condition is based on the PID column, meaning **product prices** are shown for matching PID values (1 and 2). However, for transactions with no corresponding product (PID 4), the price appears as NULL. The resulting output is as follows.



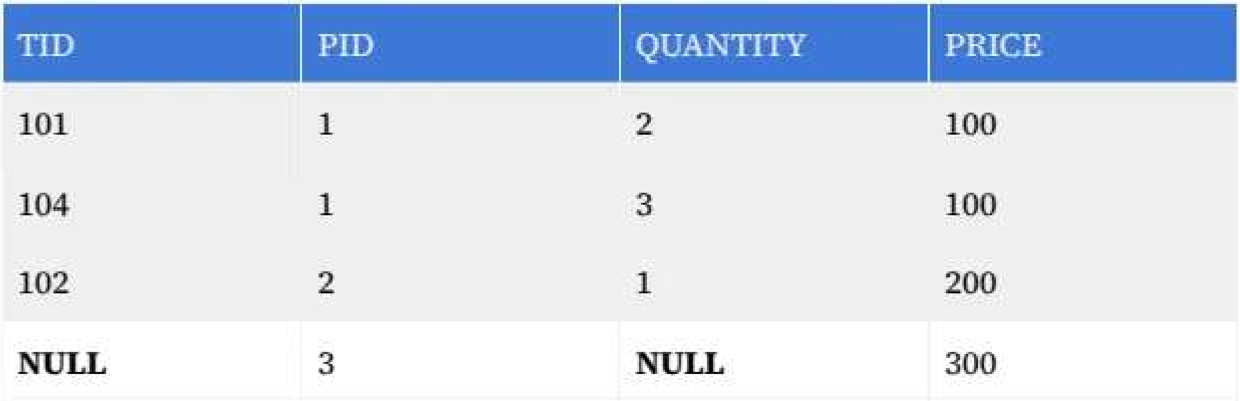
TID	PID	QUANTITY	PRICE
101	1	2	100
102	2	1	200
103	4	5	**NULL**
104	1	3	100

RIGHT JOIN — Include unmatched rows from the *right* table

To retrieve transaction details along with the corresponding product price, including **all product** records and matching **transactions where available.**

```
SELECT TRANSACTION.TID, TRANSACTION.PID, TRANSACTION.QUANTITY, PRODUCT.PRICE
FROM TRANSACTION
RIGHT JOIN PRODUCT
ON PRODUCT.PID = TRANSACTION.PID;
```

Since we use a RIGHT JOIN, **all product** records are included in the result. The join condition is based on the PID column, meaning **transaction details** are shown for matching PID values (1 and 2). However, for products with no corresponding transaction (PID 3), the transaction details appear as NULLs. The resulting output is as follows.



TID	PID	QUANTITY	PRICE
101	1	2	100
104	1	3	100
102	2	1	200
NULL	3	**NULL**	300

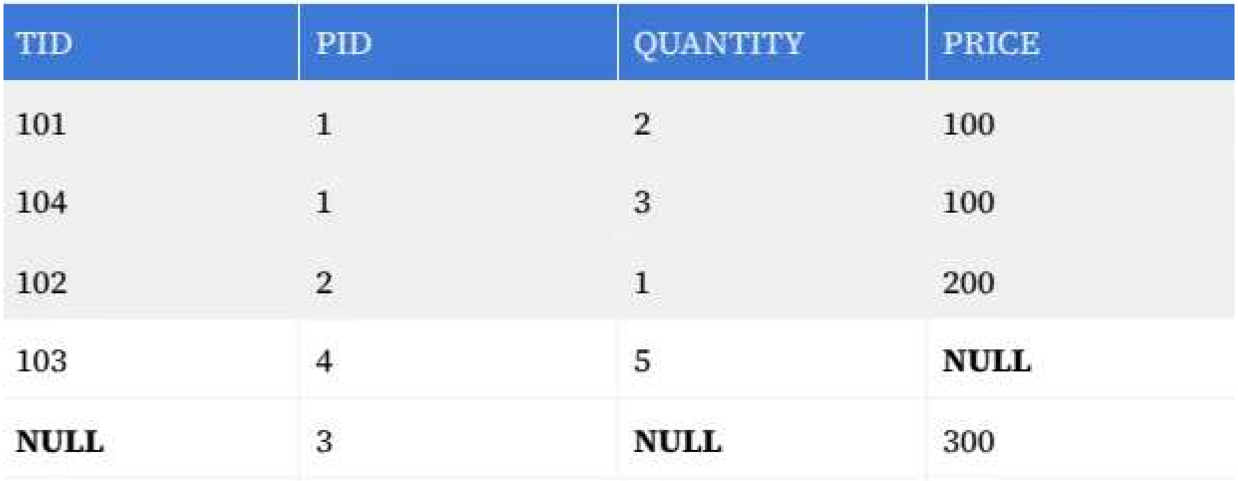
FULL JOIN — Include unmatched rows from *both tables*

To retrieve transaction details along with the corresponding product price, including **all transaction** records and **all product prices.**

```
SELECT TRANSACTION.TID, TRANSACTION.PID, TRANSACTION.QUANTITY, PRODUCT.PRICE
FROM TRANSACTION
```

```
FULL JOIN PRODUCT
ON PRODUCT.PID = TRANSACTION.PID;
```

Since we use a FULL JOIN, all records from **both the transaction and product tables** are included in the result. The join is based on the PID column, so transaction and product prices are displayed for matching PID values (1 and 2). If a transaction has no corresponding product (PID 4), the price appears as NULL. Similarly, if a product has no corresponding transaction (PID 3), the transaction details appear as NULLs. The resulting output is as follows.



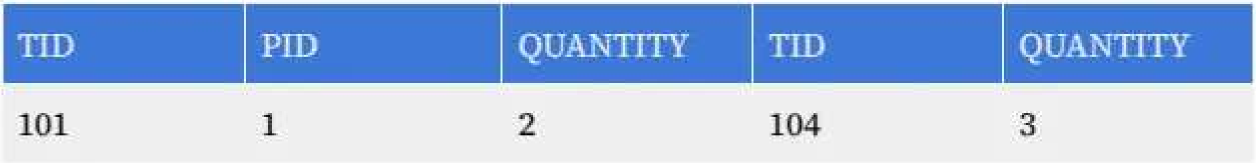
TID	PID	QUANTITY	PRICE
101	1	2	100
104	1	3	100
102	2	1	200
103	4	5	**NULL**
NULL	3	**NULL**	300

SELF JOIN — Join a table with itself

To find transactions where the **same product** (PID) is part of **two different transactions.**

```
SELECT T1.TID, T1.PID, T1.QUANTITY, T2.TID, T2.QUANTITY
FROM TRANSACTION T1
```

```
INNER JOIN TRANSACTION T2 ON T1.PID = T2.PID AND T1.TID <> T2.TID;
```



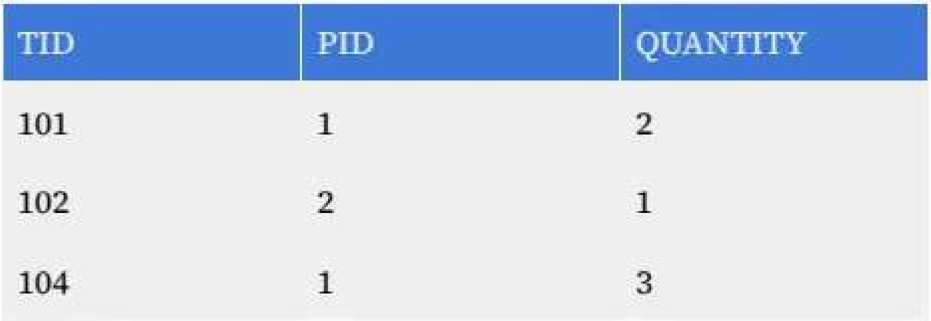
TID	PID	QUANTITY	TID	QUANTITY
101	1	2	104	3

EXISTS / NOT EXISTS — Check if subquery returns results

To find transactions that have a matching product.

```
SELECT T.*
FROM TRANSACTION T
WHERE EXISTS (
SELECT 1
FROM PRODUCT P
WHERE P.PID = T.PID
);
```

The subquery retrieves PID values from the PRODUCT table and checks if a matching product exists. It returns TRUE for transactions where the PID is found in PRODUCT. Since the TRANSACTION table contains PID values {1, 2, 4} and the PRODUCT table contains {1, 2, 3}, transactions with PID 4 are excluded from the final results.

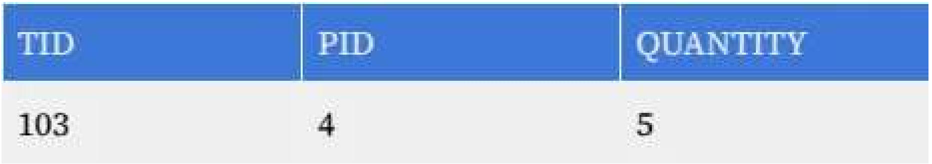


TID	PID	QUANTITY
101	1	2
102	2	1
104	1	3

To find transactions that do not have a matching product.

```
SELECT T.*
FROM TRANSACTION T
WHERE NOT EXISTS (
SELECT 1
FROM PRODUCT P
WHERE P.PID = T.PID
);
```

The subquery retrieves PID values from the PRODUCT table and checks if a matching product does not exist. It returns TRUE for transactions where the PID is not found in PRODUCT. Since the TRANSACTION table contains PID values {1, 2, 4} and the PRODUCT table contains {1, 2, 3}, only transactions with PID 4 remain in the final results, while transactions with PID 1 and PID 2 are excluded.



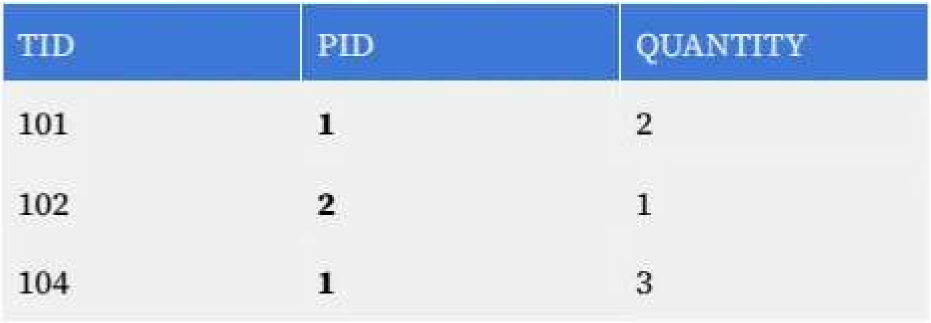
TID	PID	QUANTITY
103	4	5

IN — Check if a value exists in a subquery result

To retrieve transactions where the product exists in the PRODUCT table.

```
SELECT *
FROM TRANSACTION
WHERE PID IN (
SELECT PID
FROM PRODUCT
);
```

The subquery retrieves the PID values {1, 2, 3} from the PRODUCT table. In the TRANSACTION table, the available PID values are {1, 2, 4}. Since PID 4 is not in the PRODUCT table, it gets excluded from the final results. Only transactions with PID 1 and PID 2 remain in the output.



TID	PID	QUANTITY
101	1	2
102	2	1
104	1	3

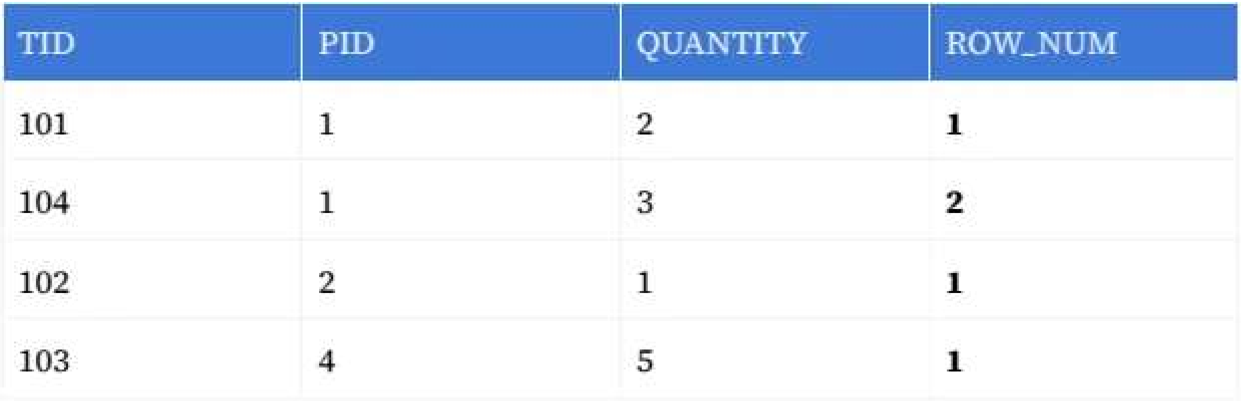
5. Window Functions (Advanced)

ROW_NUMBER() — Assign row numbers

To group transactions by PID and assign a unique number to each transaction within its group, based on the TID.

```
SELECT
  TID,
  PID,
  QUANTITY,
  ROW_NUMBER() OVER (PARTITION BY PID ORDER BY TID) AS ROW_NUM
FROM TRANSACTION;
```

The row numbers are reset for each PID group.



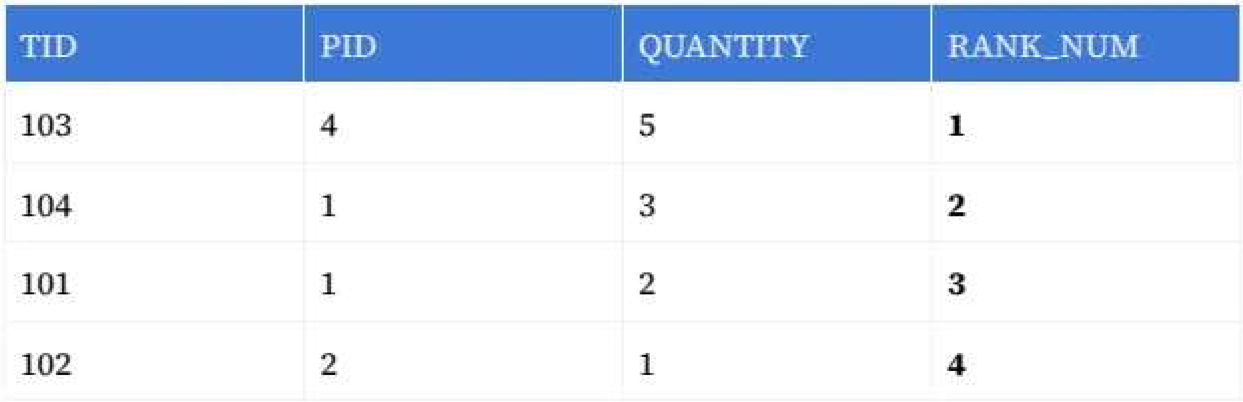
TID	PID	QUANTITY	ROW_NUM
101	1	2	**1**
104	1	3	**2**
102	2	1	**1**
103	4	5	**1**

RANK(), DENSE_RANK() — Assign ranks with handling of duplicates

To rank transactions based on QUANTITY in descending order.

```
SELECT
TID,
PID,
QUANTITY,
RANK() OVER (ORDER BY QUANTITY DESC) AS RANK_NUM
FROM TRANSACTION;
```

Results:

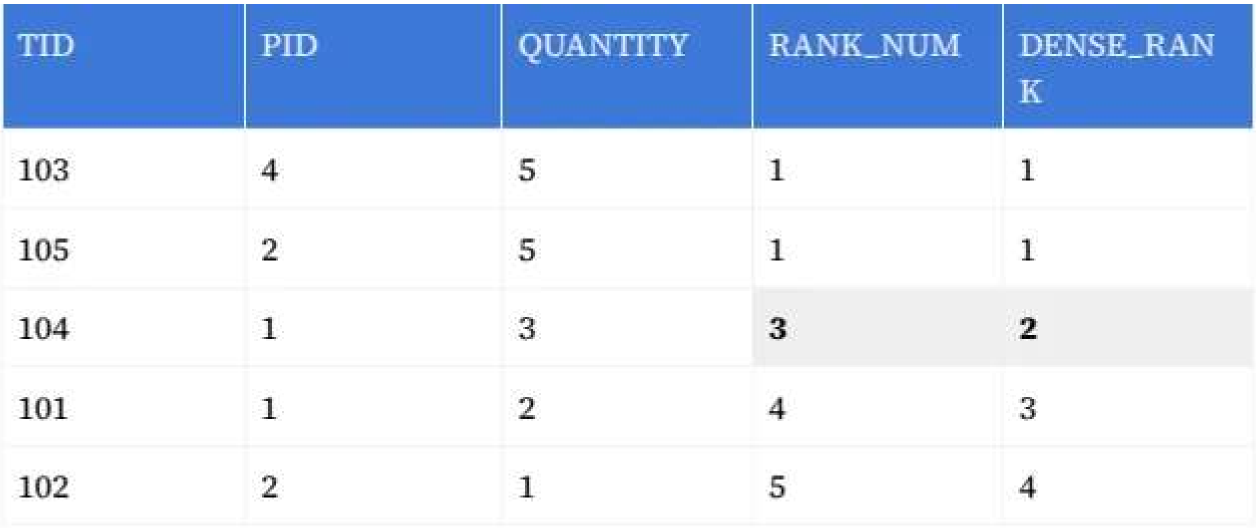


TID	PID	QUANTITY	RANK_NUM
103	4	5	**1**
104	1	3	**2**
101	1	2	**3**
102	2	1	**4**

If we add a new transaction (TID 105) with PID = 2 and QUANTITY = 5, it has the same QUANTITY value as TID 103. Since both transactions share the same quantity, they should be assigned the same rank. To maintain **consecutive ranking without skipping numbers**, we use DENSE_RANK().

```
SELECT
  TID,
  PID,
  QUANTITY,
  RANK() OVER (ORDER BY QUANTITY DESC) AS RANK_NUM
  DENSE_RANK() OVER (ORDER BY QUANTITY DESC) AS DENSE_RANK
FROM TRANSACTION;
```

Results as follows:



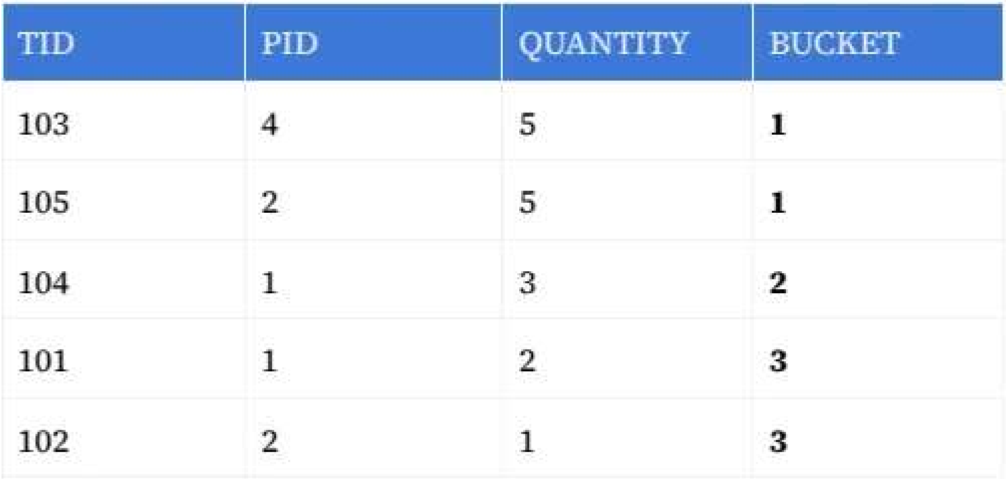
TID	PID	QUANTITY	RANK_NUM	DENSE_RANK
103	4	5	1	1
105	2	5	1	1
104	1	3	**3**	**2**
101	1	2	4	3
102	2	1	5	4

NTILE(n) — Divide results into *n* groups

To divide transactions into **3 groups** by the QUANTITY.

```
SELECT
  TID,
  PID,
  QUANTITY,
  NTILE(3) OVER (ORDER BY QUANTITY DESC) AS BUCKET
FROM TRANSACTION;
```

The results as follows:

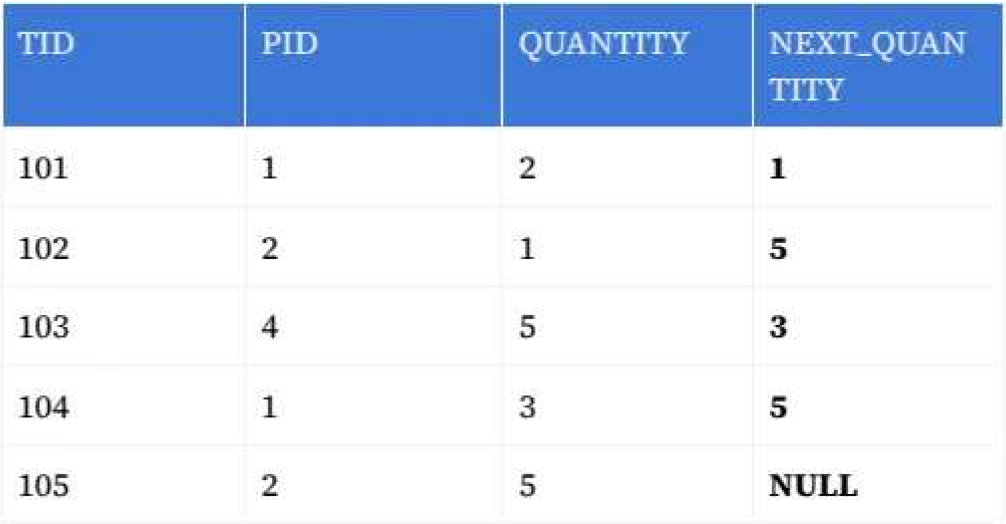


TID	PID	QUANTITY	BUCKET
103	4	5	**1**
105	2	5	**1**
104	1	3	**2**
101	1	2	**3**
102	2	1	**3**

LEAD(), LAG() — Get next/previous row values

To get the **next** transaction’s quantity.

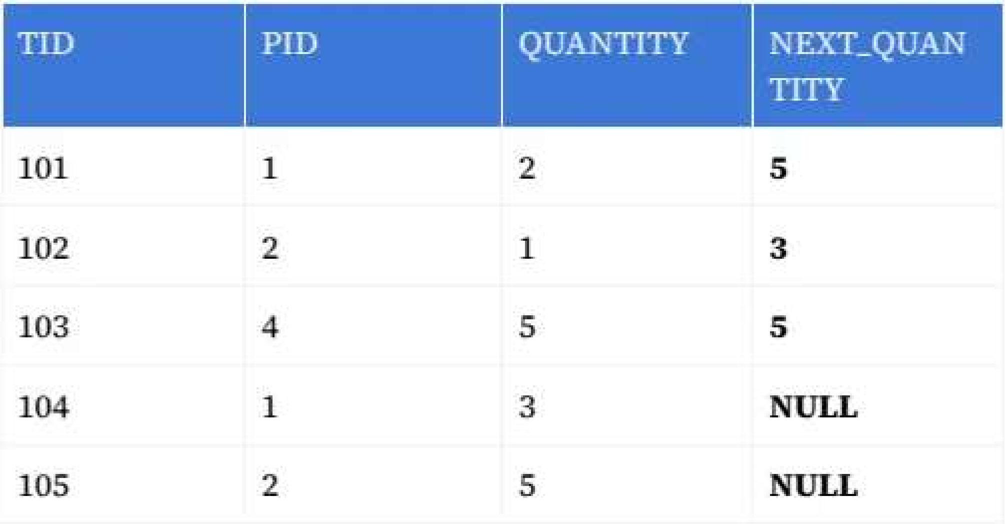
```
SELECT
  TID,
  PID,
  QUANTITY,
  LEAD(QUANTITY) OVER (ORDER BY TID) AS NEXT_QUANTITY
FROM TRANSACTION;
```



TID	PID	QUANTITY	NEXT_QUANTITY
101	1	2	**1**
102	2	1	**5**
103	4	5	**3**
104	1	3	**5**
105	2	5	**NULL**

To get the quantity of the **next two** transactions.

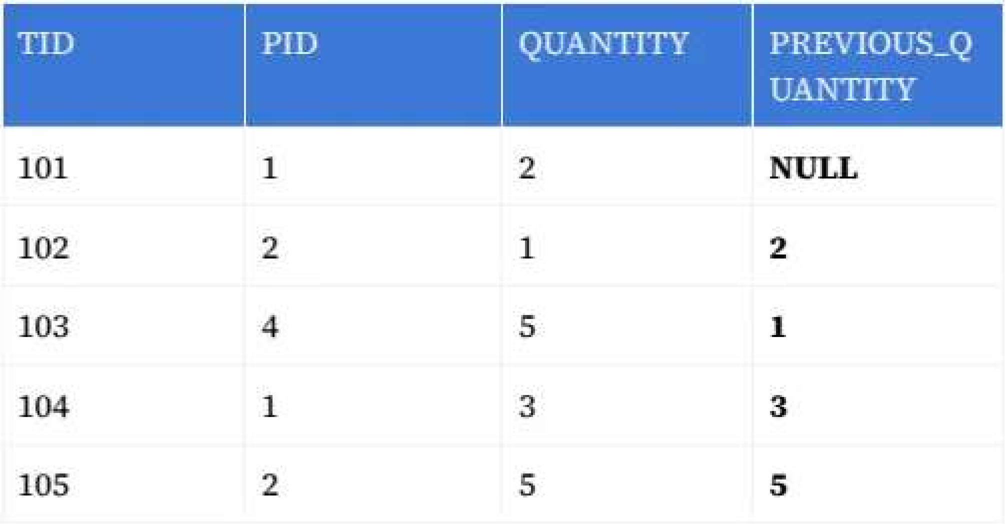
```
SELECT
  TID,
  PID,
  QUANTITY,
  LEAD(QUANTITY, 2) OVER (ORDER BY TID) AS NEXT2_QUANTITY
FROM TRANSACTION;
```



TID	PID	QUANTITY	NEXT_QUANTITY
101	1	2	**5**
102	2	1	**3**
103	4	5	**5**
104	1	3	**NULL**
105	2	5	**NULL**

To get the **previous** transaction’s quantity.

```
SELECT
  TID,
  PID,
  QUANTITY,
  LAG(QUANTITY) OVER (ORDER BY TID) AS PREVIOUS_QUANTITY
FROM TRANSACTION;
```

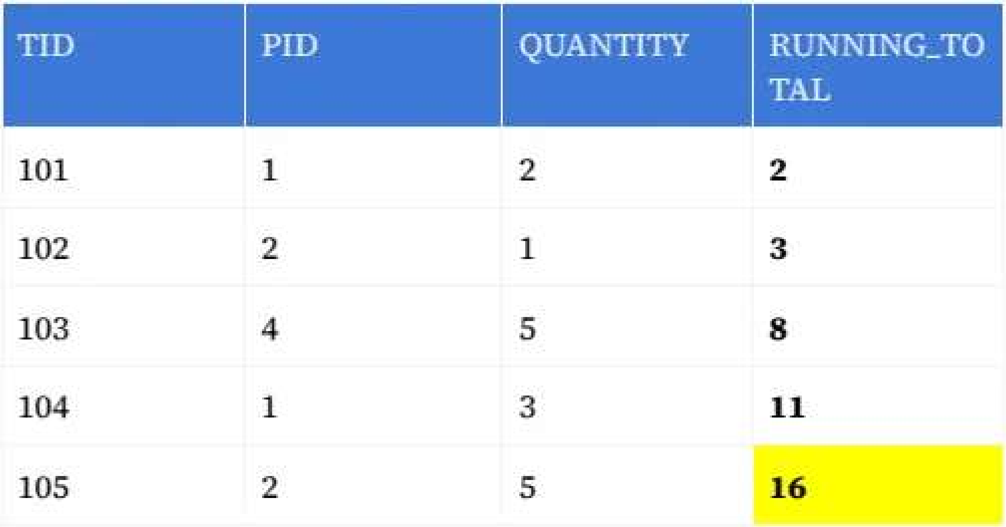


TID	PID	QUANTITY	PREVIOUS_QUANTITY
101	1	2	**NULL**
102	2	1	**2**
103	4	5	**1**
104	1	3	**3**
105	2	5	**5**

SUM() OVER(), AVG() OVER() — Aggregate over *a window*

To calculate the **running total** of sold products.

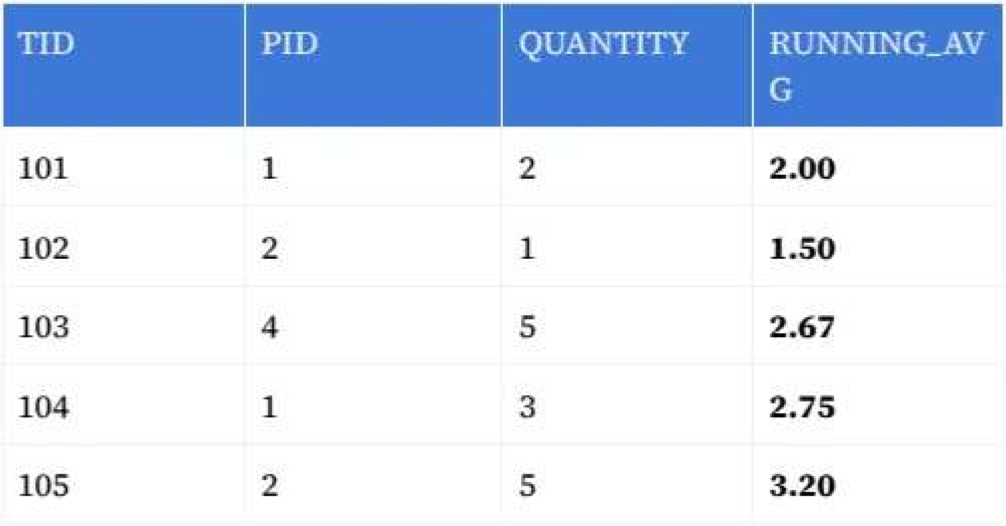
```
SELECT
  TID,
  PID,
  QUANTITY,
  SUM(QUANTITY) OVER (ORDER BY TID) AS RUNNING_TOTAL
FROM TRANSACTION;
```



TID	PID	QUANTITY	RUNNING_TOTAL
101	1	2	**2**
102	2	1	**3**
103	4	5	**8**
104	1	3	**11**
105	2	5	**16**

To calculate the **running average total** of sold products.

```
SELECT
TID,
PID,
QUANTITY,
AVG(QUANTITY) OVER (ORDER BY TID) AS RUNNING_AVG
FROM TRANSACTION;
```



TID	PID	QUANTITY	RUNNING_AVG
101	1	2	**2.00**
102	2	1	**1.50**
103	4	5	**2.67**
104	1	3	**2.75**
105	2	5	**3.20**

6. Date and Time Functions

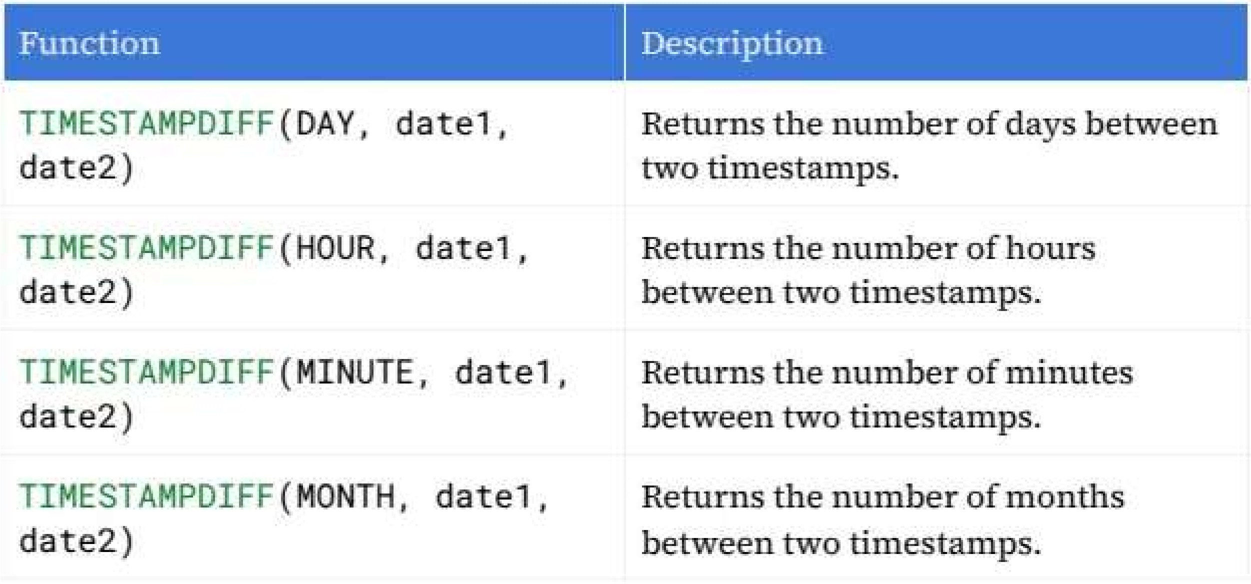
DATE(), YEAR(), MONTH(), DAY(): used to extract only date, year, month or day from a datetime field.

DATEDIFF() — Find the difference between dates

To calculate days between 2025–01–01 and 2024–12–25 (7 days)

```
SELECT DATEDIFF('2025-01-01', '2024-12-25') AS DATES_DIFFERENCE;
```

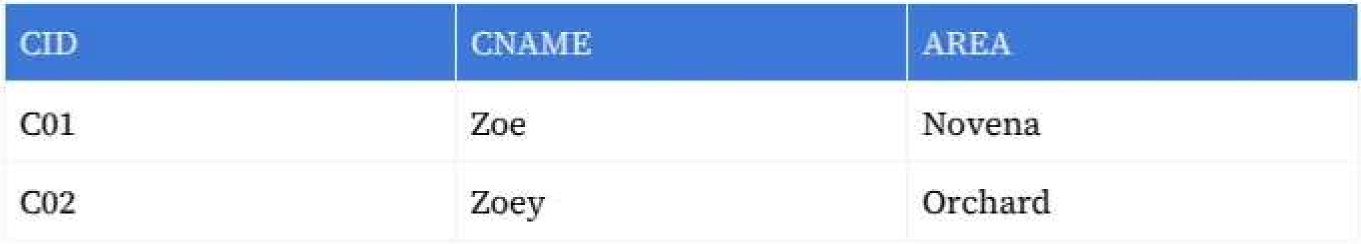
TIMESTAMPDIFF() — Compute differences between timestamps



Function	Description
`TIMESTAMPDIFF(DAY, date1, date2)`	Returns the number of days between two timestamps.
`TIMESTAMPDIFF(HOUR, date1, date2)`	Returns the number of hours between two timestamps.
`TIMESTAMPDIFF(MINUTE, date1, date2)`	Returns the number of minutes between two timestamps.
`TIMESTAMPDIFF(MONTH, date1, date2)`	Returns the number of months between two timestamps.

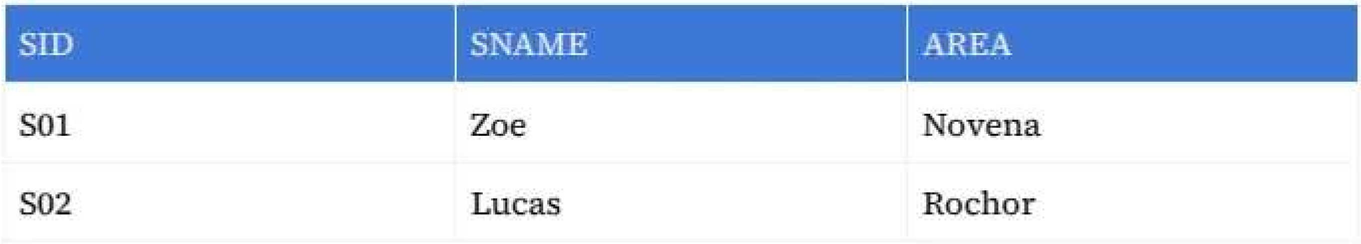
7. Set Operations

Assume we have two tables:



CID	CNAME	AREA
C01	Zoe	Novena
C02	Zoey	Orchard

CUSTOMER table



SID	SNAME	AREA
S01	Zoe	Novena
S02	Lucas	Rochor

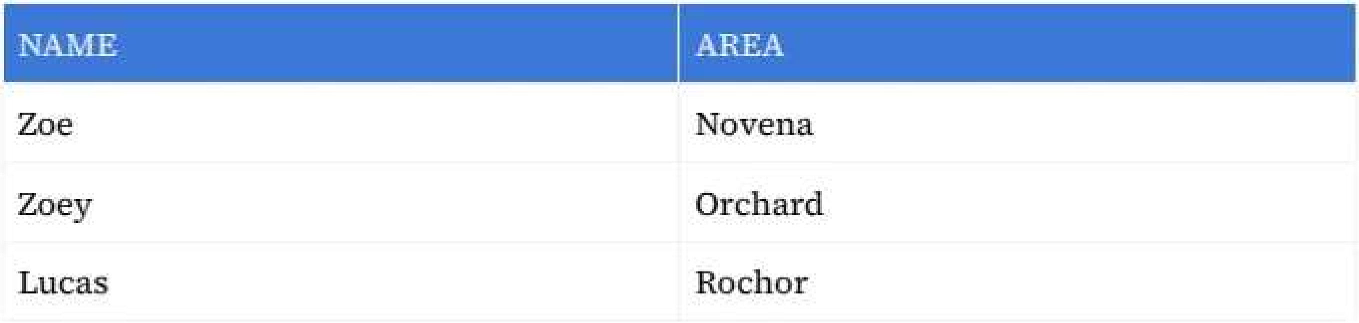
SUPPLIER table

UNION, UNION ALL — Combine multiple result sets

To get the names and corresponding areas from both tables.

```
(SELECT CNAME AS NAME, AREA
FROM CUSTOMER)
UNION
(SELECT SNAME AS NAME, AREA
FROM SUPPLIER)
```

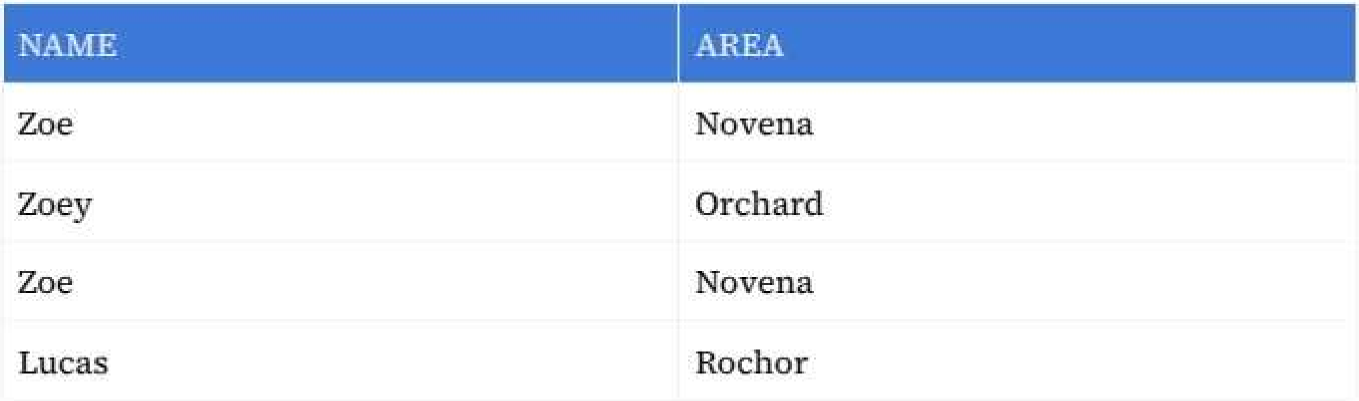
The UNION only returns **distinct names and corresponding areas**, they automatically remove duplicates. Results as follows.



NAME	AREA
Zoe	Novena
Zoey	Orchard
Lucas	Rochor

To get **all names and corresponding areas** from both tables.

```
(SELECT CNAME AS NAME, AREA
FROM CUSTOMER)
UNION ALL
(SELECT SNAME AS NAME, AREA
FROM SUPPLIER)
```



NAME	AREA
Zoe	Novena
Zoey	Orchard
Zoe	Novena
Lucas	Rochor

8. String Functions

CONCAT(), CONCAT_WS() — Merge strings

To combine the first name and last name: CONCAT(FIRST_NAME, LAST_NAME)

To combine the first name and last name with a space (**with separators**) in between: CONCAT_WS(' ', FIRST_NAME, LAST_NAME)

SUBSTRING(), LEFT(), RIGHT() — Extract parts of a string

To get the **first three characters** of the first name:

SUBSTRING(FIRST_NAME, 1, 3)

To get the **last three characters** of the first name: RIGHT(FIRST_NAME, 3)

TRIM(), LTRIM(), RTRIM() — Remove spaces

TRIM() removes spaces from both ends of the string.

LTRIM() removes spaces from the left (beginning) of the string only.

RTRIM() removes spaces from the right (end) of the string only.

REPLACE(), CHAR_LENGTH()

To replace all occurrences of a substring within a string with another substring.

REPLACE('I love apple pie', 'apple', 'orange')

Result: 'I love **orange** pie'

To count how many characters of a string.

CHAR_LENGTH('Hello World')

Result: 11

LOCATE(), POSITION(): returns the position of the first occurrence of a substring within a string.

Step-by-Step Approach to Solving a SQL Problem

Let's use **LeetCode Problem 262 — Trips and Users** as an example: https://leetcode.com/problems/trips-and-users/description/

Step 1. Understanding the Problem

- We need to calculate **the cancellation rate** for **each day** between "2013–10–01" and "2013–10–03".

- Cancellations occur if the trip status is either *cancelled_by_driver* or *cancelled_by_client*.
- Only include trips where both the client and driver **are not banned.**
- The cancellation rate is calculated by dividing the number of canceled trips by the total number of trips for that day, and the result should be rounded to two decimal points.

Step 2. Tables Involved

- Trips: stores information about trips
- Users: stores user details, including their roles and ban status.

Step 3. Identifying Key Requirements

- Filter for trips that occurred between “2013–10–01” and “2013–10–03”.
- Ensure that both the client (client_id) and the driver (driver_id) are not banned.
- Calculate the cancellation rate by dividing the canceled trips by the total trips.

Step 4. Writing the SQL Query

- We need to join the Trips table with the Users table **twice:** once for the client and once for the driver.

```
SELECT
FROM
  Trips t
JOIN
  Users u1 ON t.client_id = u1.users_id
JOIN
  Users u2 ON t.driver_id = u2.users_id
WHERE
```

```
  t.request_at BETWEEN '2013-10-01' AND '2013-10-03'
GROUP BY
  t.request_at
HAVING
  COUNT(*) > 0;
```

- Filter out the trips where either the client or the driver is banned.

```
SELECT
  (CASE
    WHEN t.status IN ('cancelled_by_driver', 'cancelled_by_client')
         AND u1.banned = 'No'
         AND u2.banned = 'No' THEN 1
    ELSE 0
  END)
FROM
  Trips t
JOIN
  Users u1 ON t.client_id = u1.users_id
JOIN
  Users u2 ON t.driver_id = u2.users_id
WHERE
  t.request_at BETWEEN '2013-10-01' AND '2013-10-03'
  AND u1.banned = 'No' AND u2.banned = 'No'
GROUP BY
  t.request_at
HAVING
  COUNT(*) > 0;
```

- Compute the cancellation rate for each day and round it to two decimal points.

```
SELECT
  t.request_at AS 'Day',
  ROUND(
    SUM(CASE
          WHEN t.status IN ('cancelled_by_driver', 'cancelled_by_client')
               AND u1.banned = 'No'
               AND u2.banned = 'No' THEN 1
          ELSE 0
        END)
  / COUNT(*) , 2) AS 'Cancellation Rate'
FROM
  Trips t
JOIN
  Users u1 ON t.client_id = u1.users_id
JOIN
  Users u2 ON t.driver_id = u2.users_id
WHERE
  t.request_at BETWEEN '2013-10-01' AND '2013-10-03'
  AND u1.banned = 'No' AND u2.banned = 'No'
GROUP BY
  t.request_at
HAVING
  COUNT(*) > 0;
```

Step 5: Testing and Validating the Query

- Run the query on sample data to check for correctness.
- Debug errors by analyzing joins and conditions.
- Verify the output against expected results.

I hope you find this article helpful! If you enjoyed it, I'd truly appreciate your support with a clap.

Sql

Leetcode

Database

Written by Zoe Dinh

3 followers · 3 following

Follow

Data Science Graduate - Nanyang Technological University Singapore

No responses yet

Write a response

What are your thoughts?

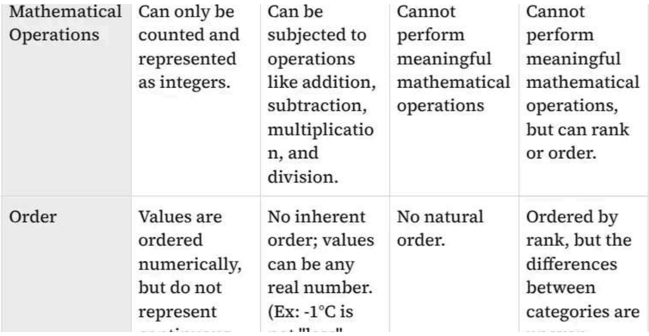
More from Zoe Dinh

Zoe Dinh

Data Science Interview Prep: The Go-To Statistics Cheat Sheet

Statistics helps us understand data, identify trends, and summarize information. It enabl…

Apr 30, 2025



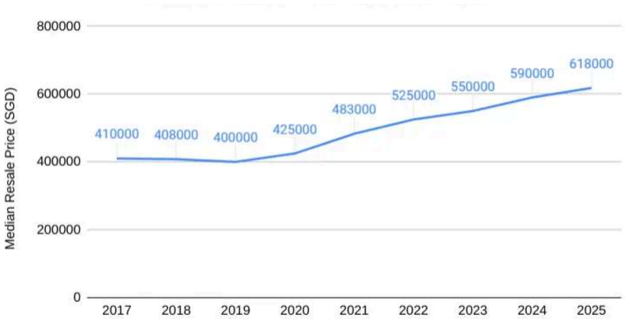
Mathematical Operations	Can only be counted and represented as integers.	Can be subjected to operations like addition, subtraction, multiplicatio n, and division.	Cannot perform meaningful mathematical operations	Cannot perform meaningful mathematical operations, but can rank or order.
Order	Values are ordered numerically, but do not represent	No inherent order; values can be any real number. (Ex: -1°C is	No natural order.	Ordered by rank, but the differences between categories are

Zoe Dinh

Unlock the Power of Data Visualization with the Right Data…

Understanding data types is key to creating clear, accurate, and impactful visualizations.…

Mar 18, 2025

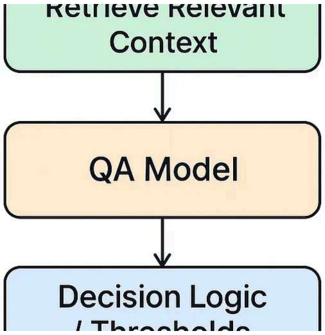


Zoe Dinh

Transforming Data into Insightful Stories with Visualization

Instead of getting lost in a sea of numbers, a well-crafted data visualization transforms…

Mar 18, 2025 3



Zoe Dinh

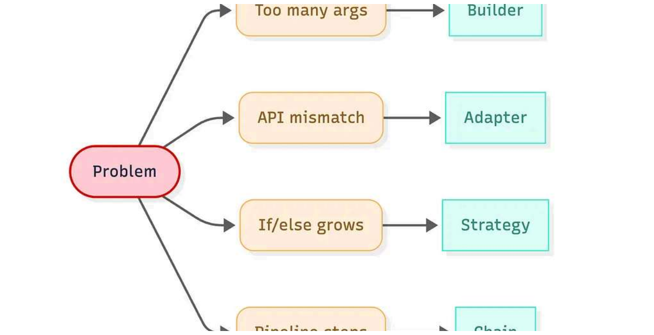
How I Built My Very First Question-Answering Bot

My goal in this project is to develop an application that helps people explore…

Sep 23, 2025

See all from Zoe Dinh

Recommended from Medium

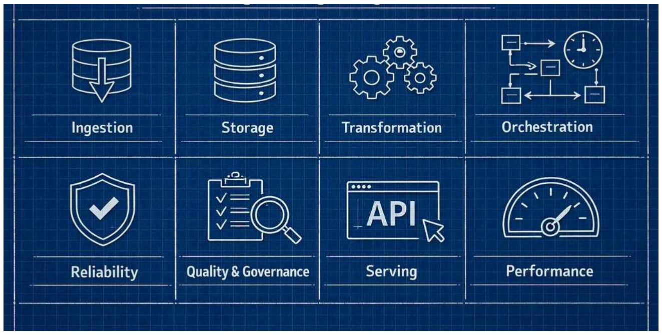


In Women in Technology by Alina Kovtun ✨

Stop Memorizing Design Patterns: Use This Decision Tree Instead

Choose design patterns based on pain points: apply the right pattern with minimal over-...

✦ Jan 29 👏 3.1K 💬 27

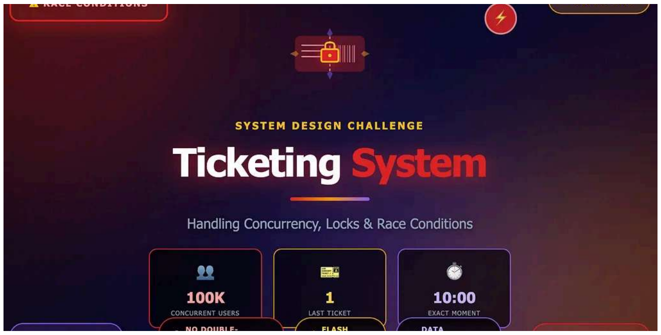


In AWS in Plain English by Khushbu Shah

Data Engineering Design Patterns You Must Learn in 2026

These are the 8 data engineering design patterns every modern data stack is built on....

✦ Jan 5 👏 909 💬 19

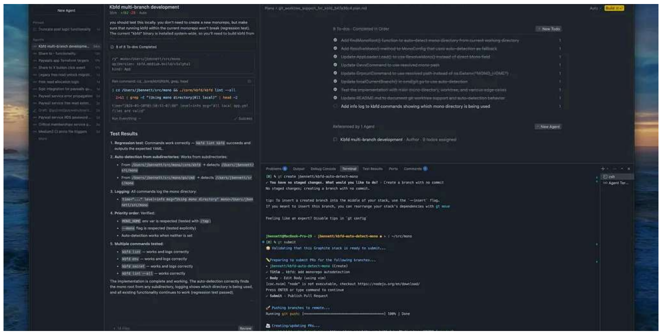


Arvind Kumar

Building a Ticketing System: Concurrency, Locks, and Race...

What happens when 100,000 fans try to book the same concert ticket at exactly 10:00 AM...

✦ Oct 30, 2025 👏 1.7K 💬 18



Jacob Bennett

The 5 paid subscriptions I actually use in 2026 as a Staff Software...

Tools I use that are (usually) cheaper than Netflix

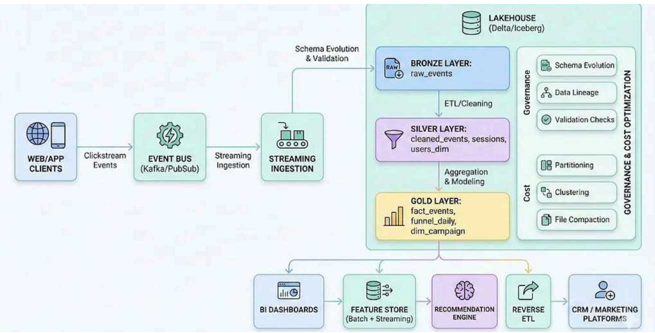
✦ Jan 19 👏 2.8K 💬 68

In Stackademic by HabibWahid

Junior Devs Use try-catch Everywhere. Senior Devs Use…

Try-catch on every method? That’s not safe code—that’s a ticking time bomb. Here’s wh…

Feb 2 275 9



Sarath Sagi

Data Engineering System Design: Clickstream Data Into a Modern…

A complete end-to-end architecture for building scalable clickstream analytics and…

Dec 12, 2025 9

See more recommendations