# COMBINED SQL FUNCTIONS

A QUICK GUIDE TO EXPLODE AND AGGREGATE AND BEYOND!



# EXPLODE() IN SQL

**PURPOSE:** Transforms each element of an array into a separate row.

INPUT TABLE: user\_activity

USER_ID	ACTIONS
1	['login', 'purchase', 'logout']
2	['login', 'purchase']
3	['login', 'logout']



-- We want to explode the actions into individual rows and count occurrences of each action

```
SELECT action, COUNT(*) AS action_count
FROM (
   SELECT user_id, EXPLODE(actions) AS action
   FROM user_activity
)
GROUP BY action
ORDER BY action_count DESC
```



### **OUTPUT:**

ACTION	ACTION_COUNT
'login'	6
'purchase'	3
'logout'	2

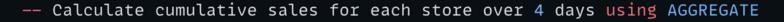
- What It Does: This query first uses EXPLODE() to convert each element of the actions array into a separate row, then counts the occurrences of each action. The outer query groups by action and orders the results by the count.
- Use Case: Ideal for analyzing user activity data to determine the most frequent actions performed by users.

## AGGREGATE() IN SQL

**PURPOSE:** Applies a custom aggregation function to each element in an array.

INPUT TABLE: daily\_sales

STORE_ID	DAILY_REVENUE
1	[500, 700, 800, 400]
2	[300, 900, 1000, 700]
3	[100, 200, 300, 400]



SELECT store\_id, AGGREGATE(daily\_revenue, 0, (acc, x)  $\rightarrow$  acc + x, acc  $\rightarrow$  acc) AS cumulative\_sales FROM daily\_sales



STORE_ID	CUMULATIVE_SALES
1	2400
2	2900
3	1000

#### **EXPLANATION**:

- What It Does: This query uses the AGGREGATE() function to calculate the cumulative sales for each store by summing the daily revenue.
- Use Case: Useful for financial analysis, especially when calculating total sales over a period.

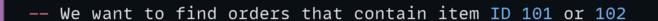


# ARRAY CONTAINS() IN SQL

**PURPOSE:** Checks if a specified array contains a given element.

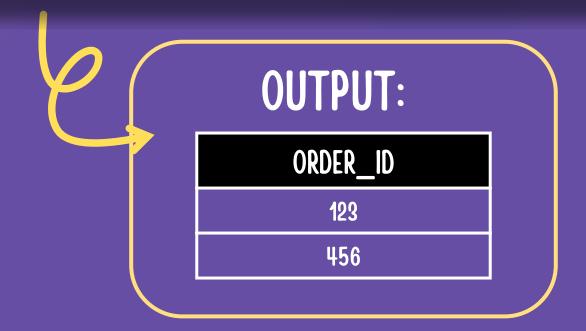
### INPUT TABLE: orders

ORDER_ID	ITEM_IDS
123	[101, 202, 303]
456	[101, 404]
789	[303, 404]
101	[202, 303, 404]



SELECT order\_id FROM orders

WHERE ARRAY\_CONTAINS(item\_ids, 101) OR ARRAY\_CONTAINS(item\_ids, 102)



- What It Does: This query searches the orders table for rows where the item\_ids array contains either 101 or 102. It uses ARRAY\_CONTAINS() to check for the presence of these IDs.
- Use Case: Useful for filtering orders based on specific items, such as finding all orders that contain certain high-priority items.

## MAP() IN SQL

**PURPOSE:** Creates a map (a collection of key-value pairs) from arrays of keys and values.

**INPUT TABLE:** product\_inventory

PRODUCT_ID	WAREHOUSES
1	['NY', 'LA', 'SF']
2	['LA', 'SF']
3	['NY', 'SF', 'TX']
4	['TX', 'NY', 'LA', 'SF']

#### INPUT TABLE: warehouse\_stock

WAREHOUSE	STOCK_LEVELS
'NY'	[100, 200, 150, 300]
'LA'	[150, 250, 100, 350]
'SF'	[200, 300, 200, 400]
'TX'	[250, 400, 300, 500]



#### OUTPUT

shekhar888

PRODUCT_ID	WAREHOUSE_STOCK_MAP
1	{'NY': 100, 'LA': 150, 'SF': 200}
2	{'LA': 150, 'SF': 200}
3	{'NY': 150, 'SF': 200, 'TX': 250}
4	{'TX': 250, 'NY': 100, 'LA': 150, 'SF': 200}

#### **EXPLANATION:**

- What It Does: This query creates a map of warehouses and their corresponding stock levels for each product, which is useful for inventory management.
- Use Case: It helps to easily look up stock levels of products across different warehouses.

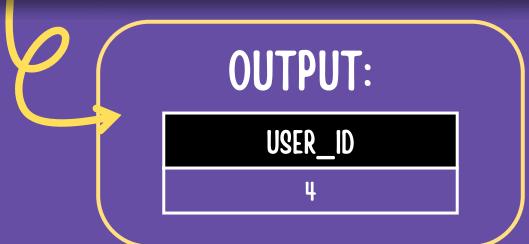
# SIZE() IN SQL

**PURPOSE:** Returns the number of elements in an array.

### INPUT TABLE: subscriptions

USER_ID	SERVICES
1	['Netflix', 'Hulu', 'Disney+']
2	['Netflix', 'Hulu']
3	['Disney+', 'HBO']
4	['Netflix', 'Hulu', 'Disney+', 'HBO', 'Amazon Prime']





- What It Does: This query filters the subscriptions table to find users who have subscribed to more than 3 services by using SIZE() to check the length of the services array.
- Use Case: Useful for identifying highly engaged users who subscribe to multiple services, which can be valuable for targeted marketing or customer segmentation.

# TRANSFORM() IN SQL

**PURPOSE:** Transforms each element in an array using a specified function.

### **INPUT TABLE: discounts**

PRODUCT_ID	ORIGINAL_PRICES
1	[100, 200, 150, 300]
2	[150, 250, 100, 350]
3	[200, 300, 200, 400]



-- Apply a 10% discount to each price in the original\_prices array

SELECT product\_id, TRANSFORM(original\_prices,  $x \rightarrow x * 0.9$ ) AS discounted\_prices FROM discounts

P

### OUTPUT

PRODUCT_ID	DISCOUNTED_PRICES
1	[90, 180, 135, 270]
2	[135, 225, 90, 315]
3	[180, 270, 180, 360]

- What It Does: This query applies a 10% discount to each price in the original\_prices array using the TRANSFORM() function.
- Use Case: Useful for pricing strategies, enabling dynamic price adjustments.

