Level Up 2026 – Advanced SQL

Assignment 3: CTEs, Partitioning, and Window Functions (Analytics Focus)

Objective

Introduce **advanced SQL constructs** while building **correct usage boundaries** and **performance intuition**, using the **same evolving `orders` table** from previous sessions.

Participants will learn:

- When advanced SQL improves clarity and analytics
- When it hurts performance or simplicity
- How PostgreSQL actually executes these constructs

All tasks must be performed using **psql CLI or use DBeaver.**

Base Schema (From Session 1)

You already have this table from Assignment 1 & 2:

```
CREATE TYPE order_status_enum AS ENUM ('CREATED', 'PAID', 'CANCELLED');

CREATE TABLE orders (
    order_id UUID NOT NULL DEFAULT gen_random_uuid() PRIMARY KEY,
    customer_id UUID NOT NULL DEFAULT gen_random_uuid(),
    city TEXT NOT NULL,
    order_amount NUMERIC NOT NULL,
    order_status order_status_enum NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT current_timestamp
);
```

Schema Extension (From Session 2)

Extend the existing table:

ALTER TABLE orders
ADD COLUMN metadata JSONB;

Populate `metadata` with realistic values such as:

```
{
  "payment_mode": "UPI",
  "coupon": "NEWUSER",
  "device": "android",
  "app_version": "1.2.3"
}
```

Ensure:

- Different payment modes (`UPI`, `CARD`, `NETBANKING`)
- Some rows without coupons
- Mixed devices (`android`, `ios`, `web`)
- Varying app versions

Problem 1: Common Table Expressions (CTEs)

Context

You are asked to prepare an **analytics query** for product insights.
Another engineer argues that “CTEs make everything slower and should be avoided”.

You need to investigate.

Tasks

1. Write a query to find customers who:
    - Have placed **more than 5 orders**
    - Have at least **3 PAID orders**

Return:

- `customer_id`
- `total_orders`
- `paid_orders`
- `total_order_amount`

2. Write this query **without using a CTE**.

```sql
select c.customer_id,count(*) as total_orders ,o.order_status,
Sum(order_amount) as total_order_amount
from customers c left join orders o on c.customer_id =o.customer_id
group by c.customer_id ,o.order_status
having count(o.order_id ) >5 and o.order_status ='PAID';
```

ults 1 | Results 6 | customers(+) 3 ×

ct c.customer_id,count(*) as total_orde | Enter a SQL expression to filter results (use Ctrl+Space)

customer_id	123 total_orders	A-Z order_status	123 total_order_amount
66666666-6666-6666-6666-666666666666	6	PAID	16,764.63

3. Rewrite the same query **using a CTE**.

```sql
with paid_orders as(
    select * from orders
    where order_status ='PAID'
),
customer_spend as(
    select customer_id,
    count(*) as total_orders,
    sum(order_amount) as total_amount
    from paid_orders
    group by customer_id
)
select * from customer_spend where total_orders >=5 ;
```

Results 1 | orders 2 | customers(+) 3 | orders 4 ×

with paid_orders as(select * from orders w | Enter a SQL expression to filter results (use Ctrl+Space)

	customer_id	123 total_orders	123 total_amount
1	66666666-6666-6666-6666-666666666666	6	16,764.63

4. Run `EXPLAIN ANALYZE` on both versions.

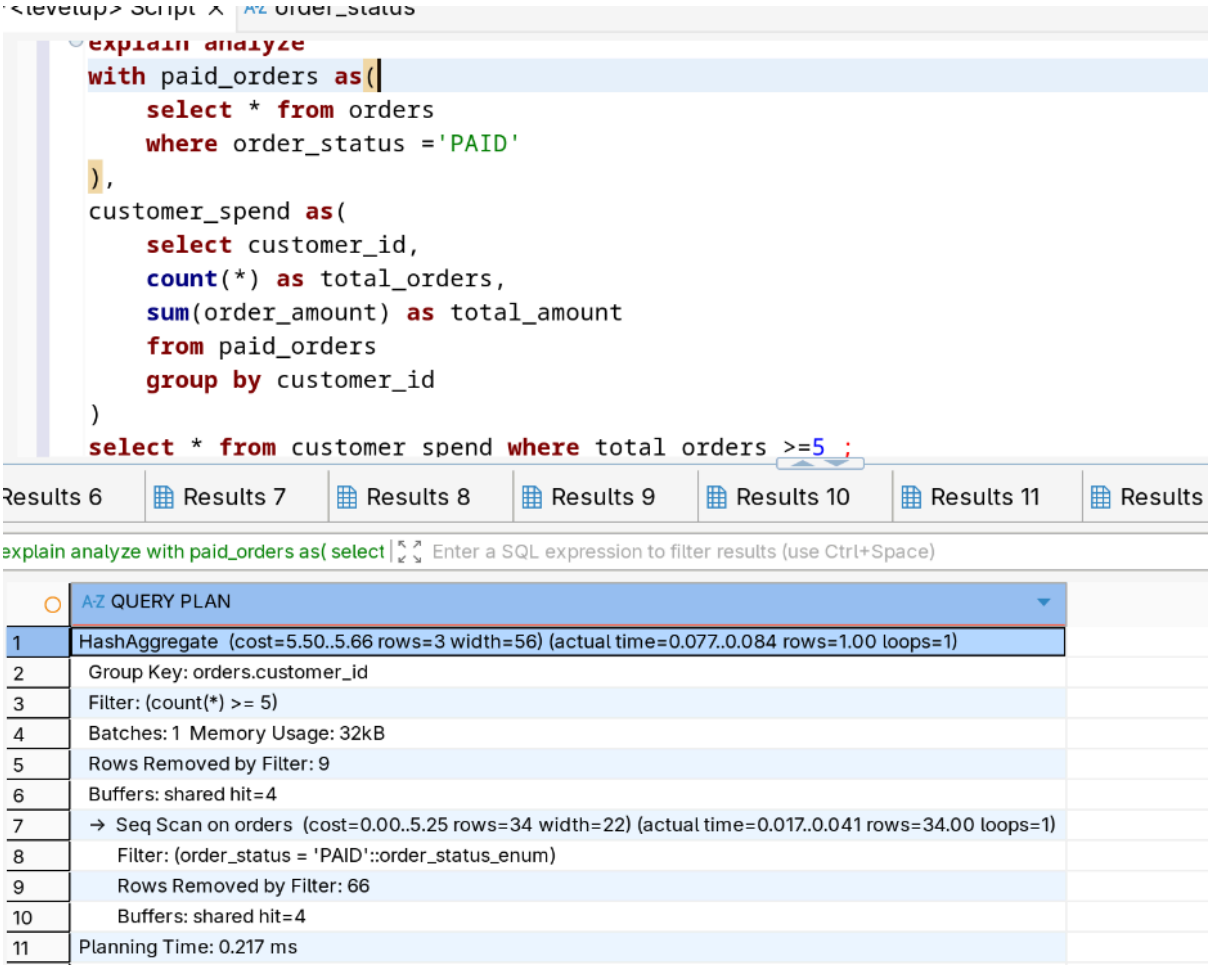
```
explain analyze select c.customer_id,count(*) as total_orders ,o.order_status,
Sum(order_amount) as total_order_amount
from customers c left join orders o on c.customer_id =o.customer_id
group by c.customer_id ,o.order_status
having count(o.order_id ) >5 and o.order_status ='PAID';
```

Results 1 | orders 2 | customers(+) 3 | orders 4 | Results 5 | Results 6

explain analyze select c.customer_id,coun | Enter a SQL expression to filter results (use Ctrl+Space)

	QUERY PLAN
1	HashAggregate (cost=23.34..23.85 rows=11 width=60) (actual time=0.181..0.184 rows=1.00 loops=1)
2	Group Key: c.customer_id
3	Filter: (count(o.order_id) > 5)
4	Batches: 1 Memory Usage: 32kB
5	Rows Removed by Filter: 7
6	Buffers: shared hit=22
7	→ Nested Loop (cost=0.16..23.00 rows=34 width=42) (actual time=0.047..0.141 rows=28.00 loops=
8	Buffers: shared hit=22
9	→ Seq Scan on orders o (cost=0.00..5.25 rows=34 width=42) (actual time=0.020..0.049 rows=3
10	Filter: (order_status = 'PAID'::order_status_enum)
11	Rows Removed by Filter: 66
12	Buffers: shared hit=4
13	→ Memoize (cost=0.16..1.59 rows=1 width=16) (actual time=0.002..0.002 rows=0.82 loops=34)
14	Cache Key: o.customer_id
15	Cache Mode: logical
16	Hits: 24 Misses: 10 Evictions: 0 Overflows: 0 Memory Usage: 2kB
17	Buffers: shared hit=18

Column: QUERY PLAN text(10485760)
Read-only: No corresponding table column



Investigate and Answer

- Are the execution plans different?

  =>No its same first it read table then group it and then filter it.

- What does PostgreSQL materializing CTE mean? What is a Materialized view? What is a view? Differentiate between the two.

  =>materialized means storing data temporarily , view is basically save query and difference is materialized storing data not query and view store query not data.

- Does PostgreSQL materialize the CTE?

  => postgreSQL does not materialized the CTE

- Which version is more readable?

  => CTE version is more readable

- Why might this still be unsafe in a live API path?

  => it can be slow on large data,no index support,no pagination.

Output Expected

- Both queries
- `EXPLAIN ANALYZE` output
- 4–5 line explanation on:
  - readability vs performance trade-off
  - when CTEs are acceptable

Problem 2: CTEs for Analytics Pipelines (Correct Usage)

Context

A reporting job runs **once per day** and computes insights for dashboards.

Tasks

Using **CTEs**, write a query that:

1. Extracts PAID orders
2. Groups them by:
   - date
   - city
3. Computes:
   - daily revenue per city
4. Computes:
   - cumulative revenue per city (ordered by date)

Break the query into **at least two CTEs**.

```sql
WITH paid_orders AS ( SELECT
        DATE(created_at) AS order_date,
        city,order_amount
    FROM orders WHERE order_status = 'PAID'
),
daily_revenue AS ( SELECT
        order_date,city,
        SUM(order_amount) AS daily_revenue
    FROM paid_orders GROUP BY order_date, city
)
SELECT order_date,city,
    daily_revenue,SUM(daily_revenue) OVER (
     PARTITION BY city ORDER BY order_date
    )
    AS cumulative_revenue FROM daily_revenue
ORDER BY city, order_date;
```

esults 8 | Results 9 | Results 10 | Results 11 | Results 12 | Res

/ITH paid_orders AS (SELECT DATE(creat

	order_date	A-Z city	123 daily_revenue	123 cumulative_revenue
1	2025-10-31	Ahmedabad	1,907.45	1,907.45
2	2025-11-03	Ahmedabad	3,090.96	4,998.41
3	2025-12-02	Ahmedabad	133.02	5,131.43
4	2025-12-27	Ahmedabad	1,295.39	6,426.82
5	2026-01-22	Ahmedabad	2,952.59	9,379.41
6	2025-11-10	Bangalore	3,843.89	3,843.89
7	2025-11-19	Bangalore	1,219.43	5,063.32
8	2025-11-22	Bangalore	2,884.31	7,947.63
9	2025-11-30	Bangalore	4,928.11	12,875.74

Investigate and Answer

- Why is this a good use case for CTEs?
  => Because it break query into smaller and clear so it improve readability
- Why would this be a bad idea inside a REST API handler?

  =>it slow down the database due to fetching and filtering data multiple times

- What does “optimization fence” mean in PostgreSQL?

  =>it mean CTE first calculate and store data and then run the query.

Problem 3: JSONB in Analytics Queries

Context

Product wants analytics sliced by:

- payment mode
- device type
- coupon usage

Tasks

1. Write a query to compute:
    - total PAID order amount per `payment_mode`

```
select metadata->>'payment_mode' as payment_mode
     ,SUM(order_amount) as total_amount_paid
     from orders where order_status='PAID'
     group by metadata->>'payment_mode'
     order by total_amount_paid;
```

esults 1 ×

elect metadata->>'payment_mode' as pa | Enter a SQL expression to filter results (use Ctrl+Space)

	A-Z payment_mode	123 total_amount_paid
1	[NULL]	3,333.52
2	wallet	19,105.49
3	UPI	25,214.65
4	card	36,401.09

2. Write a query to find:
    - percentage of PAID orders using coupons

```
select ROUND(100.0 * COUNT(*) FILTER (WHERE metadata ? 'coupon')
        / COUNT(*),2) AS coupon_usage_percentage
        from orders where order_status='PAID';
```

esults 1 ×

elect ROUND(100.0 * COUNT(*) FILTER (| Enter a SQL expression to filter results (use Ctrl+Space)

	123 coupon_usage_percentage
1	67.65

3. Write a query to show:

- daily order count split by `device`

```sql
SELECT DATE(created_at) AS order_date,
    metadata->>'device' AS device,
    COUNT(*) AS order_count
FROM orders
GROUP BY DATE(created_at), metadata->>'device'
ORDER BY order_date, device;
```

Results 1

SELECT DATE(created_at) AS order_date, r | Enter a SQL expression to filter results (use Ctrl+Space)

	order_date	device	order_count
1	2025-10-30	[NULL]	2
2	2025-10-31	[NULL]	1
3	2025-11-02	android	2
4	2025-11-02	web	1
5	2025-11-03	web	2
6	2025-11-04	android	1
7	2025-11-04	ios	1
8	2025-11-04	web	1
9	2025-11-04	[NULL]	1
10	2025-11-07	[NULL]	1

Investigate and Answer

- Why is JSONB acceptable for analytics metadata?

  =>beacuse these type of data is changes frequently so doesnt need strict schema

- Why would JSONB be a poor choice for high-QPS transactional filtering?
  =>JSONB is slower to filter data compared normal index coloumn
- What indexing challenges exist with JSONB?
  =>complex JSON query indexes can became large

Problem 4: Table Partitioning – Conceptual Understanding

Context

The `orders` table is expected to grow to **hundreds of millions of rows**.

Most queries:

- filter by time range
- are analytics/reporting heavy

Tasks

Answer conceptually (no SQL yet):

1. What problem does **partitioning** solve?

2. What is partitioning in Postgres?
3. Why indexes alone are not sufficient?
4. Why `created_at` is the best partition key?
5. Why `customer_id` or `city` are poor partition keys?

Partitioning means splitting one big table into smaller smaller parts based on some column. It helps because when table becomes very large, database does not scan full table, it only scan needed partition. In PostgreSQL, partitioning means one logical table is divided into many child tables internally.Indexes alone are not enough because even index can become very big and still need to scan lot of data. `created_at` is best partition key because most queries filter by time range, so database can easily skip old partitions. `customer_id` or `city` are bad keys because queries usually not filter by them in range and data can become uneven distributed.

Output Expected

A concise explanation (6–8 lines total).

Problem 5: Partitioned Orders Table

Tasks

1. Create a new table `orders_partitioned`:
    - Same schema as `orders`
    - Partitioned by **RANGE on `created_at`**

- Monthly partitions (at least 3)

```sql
CREATE TABLE orders_partitioned (
    order_id UUID DEFAULT gen_random_uuid(),
    customer_id UUID DEFAULT gen_random_uuid(),
    city TEXT,
    order_amount NUMERIC,
    order_status order_status_enum,
    created_at TIMESTAMPTZ NOT NULL,
    metadata JSONB
) PARTITION BY RANGE (created_at);
```

Statistics 1

Name	Value
Updated Rows	0
Execute time	0.025s
Start time	Sun Feb 08 18:19:42 IST 2026
Finish time	Sun Feb 08 18:19:42 IST 2026
Query	CREATE TABLE orders_partitioned (order_id UUID DEFAULT gen_random_uuid(), customer_id UUID DEFAULT gen_random_uuid(), city TEXT, order_amount NUMERIC, order_status order_status_enum, created_at TIMESTAMPTZ NOT NULL,

2. Insert sample data into multiple partitions.

3. Run a query to fetch orders from **one month only**.

```sql
SELECT * FROM orders_partitioned
WHERE created_at >= '2026-01-01' AND created_at < '2026-02-01';
```

orders_partitioned 1

SELECT * FROM orders_partitioned WHER | Enter a SQL expression to filter results (use Ctrl+Space)

	order_id	customer_id	city	order_amount	order_status	created_a
1	d4c6b736-660d-4dc3-b144-6ff0ffb9295a	618a8d2f-94ac-4ebe-a5a5-2c03ac4db169	Delhi	1,000	PAID	2026-01-10 0
2	efeb1ebd-6e6b-4003-bbed-eda3da80657a	bfb657e7-278b-427e-9349-0b750175ab6d	Delhi	1,000	PAID	2026-01-10 0

4. Run EXPLAIN.

```sql
explain SELECT * FROM orders_partitioned
WHERE created_at >= '2026-01-01' AND created_at < '2026-02-01';
```

Results 1

explain SELECT * FROM orders_partitione | Enter a SQL expression to filter results (use Ctrl+Space)

	QUERY PLAN
1	Seq Scan on orders_jan_2026 orders_partitioned (cost=0.00..17.20 rows=2 width=140)
2	Filter: ((created_at >= '2026-01-01 00:00:00+05:30'::timestamp with time zone) AND (created_at < '2026-02-01 00:00:00+05:30'::timestamp with time zone))

Investigate and Answer

- Which partitions are scanned?
  => partitions split big table by data
- What is partition pruning?
  =>automatically skips partitions that do not match the WHERE condition.
- Why does this improve performance for time-bounded queries?
  =>Because instead of scanning millions of rows in the whole table, the database scans only one small partition.

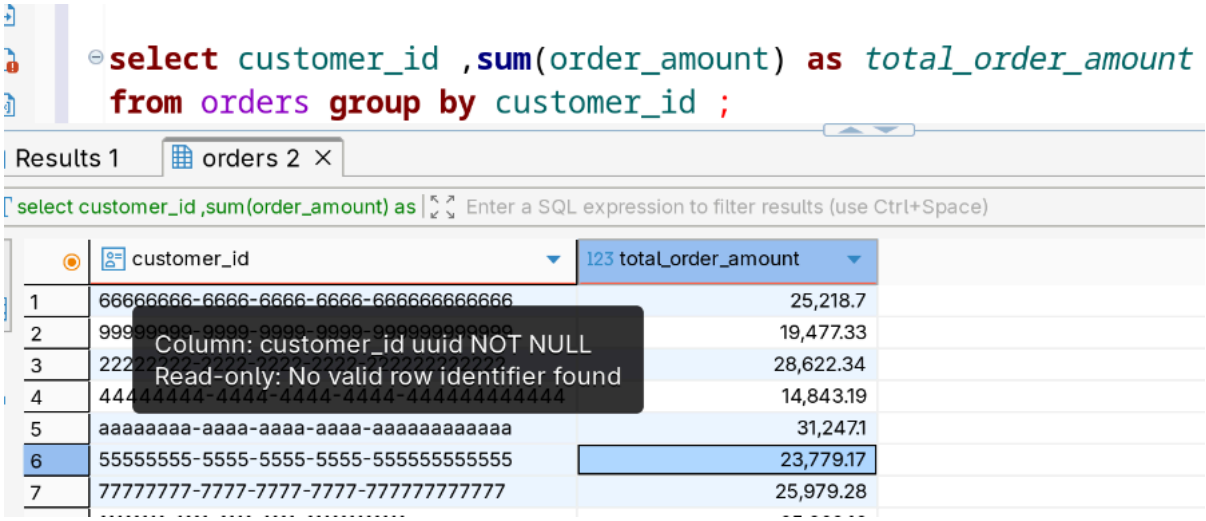
Problem 6: Window Functions vs GROUP BY

Context

A developer uses `GROUP BY` and loses row-level detail required by analytics.

Tasks

1. Using `GROUP BY`, compute:
   - total order amount per customer



2. Using a **window function**, compute:
   - each order

- along with the customer's total spend

```sql
select order_id,customer_id ,sum(order_amount) over
(partition by customer_id ) as total_spend from orders ;
```

sults 1 | orders 2 ×

ct order_id,customer_id ,sum(order_an | Enter a SQL expression to filter results (use Ctrl+Space)

order_id	customer_id	123 total_spend
fdaa5722-39ea-4728-b4f7-1e847962f10a	11111111-1111-1111-1111-111111111111	25,869.16
5762caf8-c128-4d6a-87a3-bc4c8b7012ff	11111111-1111-1111-1111-111111111111	25,869.16
6692c54b-721a-49a1-b5fe-aa49280784ee	22222222-2222-2222-2222-2222:	28,622.34
c393c3f9-3ae5-47f2-a435-2e09888ab922	22222222-2222-2222-2222-2222:	28,622.34
195b52b4-2da7-4f5d-80f5-8a4541b8f44c	22222222-2222-2222-2222-2222:	28,622.34
ebb8cb63-a52c-480f-ad5a-41f6627495f0	22222222-2222-2222-2222-2222:	28,622.34
0808266e-7916-4af1-b272-7d4183ef6360	22222222-2222-2222-2222-2222:	28,622.34
ba56e8df-9870-4c70-bc76-de619f410390	22222222-2222-2222-2222-2222:	28,622.34
5a1b1c68-a38e-4dba-879c-098183519d2d	22222222-2222-2222-2222-2222:	28,622.34
d7fa824d-97dd-4cc3-a321-745074692b37	22222222-2222-2222-2222-2222:	28,622.34

Investigate and Answer

- Why does `GROUP BY` collapse rows?

  => group by combines all rows of the same group into one result row.

- Why do window functions preserve rows?
  =>Because window functions calculate values over a group but do not combine rows
- What does `OVER()` represent conceptually?
  =>OVER() defines the "window" or group of rows used for calculation

---

Problem 7: Practical Analytics with Window Functions

Tasks

Write queries to compute:

1. Rank orders **per customer** by order_amount

```sql
select order_id,customer_id ,order_amount,
RANK() over (partition by customer_id order by order_amount DESC )
as order_rank from orders ;
```

Results 1 | orders 2 ×

select order_id,customer_id ,order_amount | Enter a SQL expression to filter results (use Ctrl+Space)

	order_id	customer_id	order_amount	order_rank
1	dfef2df8-8ba1-4a8d-853f-4e99e5d490c9	11111111-1111-1111-1111-111111111111	4,813.2	1
2	3e032468-bdb7-44a4-a5f2-cbbf17a28a97	11111111-1111-1111-1111-111111111111	4,544.46	2
3	145b4712-cbee-4bd1-a019-7cedd862a044	11111111-1111-1111-1111-111111111111	4,335.27	3
4	81a8c385-ccb9-418f-b496-90729a171ee7	11111111-1111-1111-1111-111111111111	4,247.93	4
5	271654d9-a6ca-4281-95a0-f1e7fc668943	11111111-1111-1111-1111-111111111111	4,067.38	5
6	fdaa5722-39ea-4728-b4f7-1e847962f10a	11111111-1111-1111-1111-111111111111	978.23	6
7	5762caf8-c128-4d6a-87a3-bc4c8b7012ff	11111111-1111-1111-1111-111111111111	967.99	7
8	ac4028f4-c360-4cda-bd2a-c5676481eee4	11111111-1111-1111-1111-111111111111	678.83	8
9	b713f80e-b0b4-4f6e-ae2a-cea43a0cdffe	11111111-1111-1111-1111-111111111111	491.89	9
10	10d3c82f-d8e4-4366-9006-fde29e12e8f7	11111111-1111-1111-1111-111111111111	379.56	10

2. Running total of spend per customer ordered by `created_at`

```sql
select order_id,customer_id ,SUM(order_amount)
over (partition by customer_id order by created_at)
as running_spend from orders ;
```

Results 1 | orders 2 ×

select order_id,customer_id ,SUM(order_ar | Enter a SQL expression to filter results (use Ctrl+Space)

	order_id	customer_id	running_spend
1	fdaa5722-39ea-4728-b4f7-1e847962f10a	11111111-1111-1111-1111-111111111111	978.23
2	5762caf8-c128-4d6a-87a3-bc4c8b7012ff	11111111-1111-1111-1111-111111111111	1,946.22
3	dfef2df8-8ba1-4a8d-853f-4e99e5d490c9	11111111-1111-1111-1111-111111111111	6,759.42
4	9745059e-fea9-4f82-b42c-9505f102bf8a	11111111-1111-1111-1111-111111111111	7,123.84
5	81a8c385-ccb9-418f-b496-90729a171ee7	11111111-1111-1111-1111-111111111111	11,371.77
6	145b4712-cbee-4bd1-a019-7cedd862a044	11111111-1111-1111-1111-111111111111	15,707.04
7	3e032468-bdb7-44a4-a5f2-cbbf17a28a97	11111111-1111-1111-1111-111111111111	20,251.5
8	10d3c82f-d8e4-4366-9006-fde29e12e8f7	11111111-1111-1111-1111-111111111111	20,631.06
9	b713f80e-b0b4-4f6e-ae2a-cea43a0cdffe	11111111-1111-1111-1111-111111111111	21,122.95
10	ac4028f4-c360-4cda-bd2a-c5676481eee4	11111111-1111-1111-1111-111111111111	21,801.78

3. Percentage contribution of each order to customer's total spend

```sql
select order_id,customer_id ,order_amount
,Round(100.0*order_amount/sum(order_amount)
over (partition by customer_id ),2)
as percent_contri from orders ;
```

Results 1 | orders 2

select order_id,customer_id ,order_amount | Enter a SQL expression to filter results (use Ctrl+Space)

	order_id	customer_id	order_amount	percent_contri
1	ac4028f4-c360-4cda-bd2a-c5676481eee4	11111111-1111-1111-1111-11111111111	678.83	2.62
2	9745059e-fea9-4f82-b42c-9505f102bf8a	11111111-1111-1111-1111-11111111111	364.42	1.41
3	145b4712-cbee-4bd1-a019-7cedd862a044	11111111-1111-1111-1111-11111111111	4,335.27	16.76
4	3e032468-bdb7-44a4-a5f2-cbbf17a28a97	11111111-1111-1111-1111-11111111111	4,544.46	17.57
5	271654d9-a6ca-4281-95a0-f1e7fc668943	11111111-1111-1111-1111-11111111111	4,067.38	15.72
6	b713f80e-b0b4-4f6e-ae2a-cea43a0cdffe	11111111-1111-1111-1111-11111111111	491.89	1.9
7	81a8c385-ccb9-418f-b496-90729a171ee7	11111111-1111-1111-1111-11111111111	4,247.93	16.42
8	dfef2df8-8ba1-4a8d-853f-4e99e5d490c9	11111111-1111-1111-1111-11111111111	4,813.2	18.61
9	10d3c82f-d8e4-4366-9006-fde29e12e8f7	11111111-1111-1111-1111-11111111111	379.56	1.47
10	fdaa5722-39ea-4728-b4f7-1e847962f10a	11111111-1111-1111-1111-11111111111	978.23	3.78

4. Rank coupons by total revenue contribution

```sql
select metadata->>'coupon' AS coupon,
    SUM(order_amount) AS total_revenue,
    RANK() OVER ( ORDER BY SUM(order_amount) DESC
    ) AS coupon_rank from orders where order_status='PAID'
    and metadata?'coupon' group by metadata->>'coupon';
```

Results 1 | Results 2

select metadata->>'coupon' AS coupon, S | Enter a SQL expression to filter results (use Ctrl+Space)

	coupon	total_revenue	coupon_rank
1	[NULL]	42,044.87	1
2	NEWUSER	13,461.71	2

Investigate and Answer

- Why window functions are ideal here
  =>Because we need to see each order and also calculate things like rank, running total, and percentage. Window functions let us calculate these without removing rows.

- Why these queries do not belong in latency-sensitive APIs
  =>they process many rows and use sorting and calculations. If many users call them database can become slow. They are better for reports not real-time APIs

---

Problem 8: Usage Boundaries & Performance Intuition

Answer briefly (3–4 lines each):

1. Why CTEs are discouraged in high-QPS CRUD queries
   =>CTEs can sometimes add extra processing. In high traffic APIs, even small extra work can slow down database. For simple CRUD, direct queries are better and faster.
2. Why partitioning is about **data management**, not just speed
   =>Partitioning helps split very big table into smaller parts. It makes data easier to manage and delete old data. Speed is a benefit, but main goal is handling large data properly.
3. Why window functions are analytics-first constructs
   =>Window functions are mainly used for ranking, running totals, and percentage. These are needed in reports and analytics, not in normal insert or update operations.
4. Why JSONB should be used carefully in transactional paths
   =>JSONB is flexible but not as fast as normal columns for filtering. In high-traffic systems, too much JSONB usage can slow queries. So it should be used carefully.

Expected Learning Outcomes

By completing this assignment, participants will:

- Understand **CTEs, partitioning, and window functions** deeply
- Know **where each construct belongs**
- Build intuition for **analytics-heavy SQL**
- Learn how schema evolution supports real-world querying
- Stop misusing advanced SQL in CRUD APIs
