📚 DATABASE CONTEXT — Just to refresh 🧠

Welcome back, dear teacher 😌. This is our **legendary Online Quiz System**, made with PostgreSQL and great vibes.
We've already submitted ERD and schema earlier (Assignment I/II), but just in case:

➡️ **Our database:** `quiz_system_v2`
➡️ **Main tables:** `userr`, `quiz`, `result`, `feedback`, `question_easy/medium/difficult`, `answer_easy/medium/difficult`, `reward`
➡️ All table names are in **singular form**, e.g., `userr`, not `users`.
➡️ All joins below are **real, tested**, and based on existing data.

🤓 "SQL is not just Structured Query Language, it's Seriously Quality Logic." — probably Socrates

PART 1 .
WINDOW FUNCTIONS
30 points

Online Quiz System – The Ultimate DBMS Project 🚀

by Alish Akadil

Task 1: Difference Between Current and Previous Row

```sql
SELECT
  quiz_id,
  attempt_date,
  score AS current_score,
  LAG(score) OVER (
    PARTITION BY quiz_id
    ORDER BY attempt_date
  ) AS previous_score,
  score - LAG(score) OVER (
    PARTITION BY quiz_id
    ORDER BY attempt_date
  ) AS score_diff
FROM result
ORDER BY quiz_id, attempt_date;
```

	quiz_id integer	attempt_date timestamp without time zone	current_score integer	previous_score integer	score_diff integer
1	1	2025-04-07 00:55:30.909853	88	[null]	[null]
2	1	2025-04-07 00:55:30.909853	92	88	4
3	1	2025-04-07 00:55:30.909853	80	92	-12
4	1	2025-04-07 00:55:30.909853	85	80	5
5	1	2025-04-07 00:55:30.909853	82	85	-3
6	2	2025-04-07 00:55:30.909853	75	[null]	[null]
7	2	2025-04-07 00:55:30.909853	90	75	15
8	2	2025-04-07 00:55:30.909853	95	90	5
9	2	2025-04-07 00:55:30.909853	77	95	-18
10	2	2025-04-07 00:55:30.909853	91	77	14
11	3	2025-04-07 00:55:30.909853	79	[null]	[null]
12	3	2025-04-07 00:55:30.909853	90	79	11
13	3	2025-04-07 00:55:30.909853	70	90	-20
14	3	2025-04-07 00:55:30.909853	80	70	10
15	3	2025-04-07 00:55:30.909853	60	80	-20

Description:

- **SUM(score) OVER ()** calculates the grand total of all `score` values.
- Dividing each `score` by that total and multiplying by 100 gives its percentage contribution.
- **ROUND(..., 2)** formats the result to two decimal places.

▨ Task 1

This query calculates how much the score has changed from the previous attempt for each quiz. `LAG(score)` looks at the previous score for the same quiz. `PARTITION BY quiz_id` ensures comparisons are only made within the same quiz. `ORDER BY attempt_date` makes sure the rows are compared in the right order. The final column shows how much better or worse the current score is compared to the last one.

Task 2: Percentage Contribution of Each Score

```sql
SELECT
  result_id,
  score,
  SUM(score) OVER () AS total_score,
  ROUND(
    100.0 * score / SUM(score) OVER (), 2
  ) AS pct_of_total
FROM result
ORDER BY pct_of_total DESC;
```

	result_id [PK] integer	score integer	total_score bigint	pct_of_total numeric
1	5	95	1234	7.70
2	7	92	1234	7.46
3	14	91	1234	7.37
4	12	90	1234	7.29
5	2	90	1234	7.29
6	10	88	1234	7.13
7	4	85	1234	6.89
8	13	82	1234	6.65
9	1	80	1234	6.48
10	9	80	1234	6.48
11	15	79	1234	6.40
12	11	77	1234	6.24
13	8	75	1234	6.08
14	3	70	1234	5.67
15	6	60	1234	4.86

Description:

- **SUM(score) OVER ()** calculates the grand total of all `score` values.
- Dividing each `score` by that total and multiplying by 100 gives its percentage contribution.
- **ROUND(..., 2)** formats the result to two decimal places.

Task 2

This query calculates how much each quiz attempt contributes to the total score in percentage. `SUM(score) OVER ()` calculates the grand total of all score values. Dividing each score by that total and multiplying by 100 gives its percentage contribution. `ROUND(..., 2)` formats the result to two decimal places. This is useful for quickly spotting which attempts had the highest impact.

Task 3: 3-Row Moving Average of Scores per User

```sql
SELECT
  user_id,
  attempt_date,
  score AS current_score,
  AVG(score) OVER (
    PARTITION BY user_id
    ORDER BY attempt_date
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
  ) AS moving_avg_3
FROM result
ORDER BY user_id, attempt_date;
```

	user_id integer	attempt_date timestamp without time zone	current_score integer	moving_avg_3 numeric
1	1	2025-04-07 00:55:30.909853	80	80.0000000000000000
2	2	2025-04-07 00:55:30.909853	90	90.0000000000000000
3	3	2025-04-07 00:55:30.909853	70	70.0000000000000000
4	4	2025-04-07 00:55:30.909853	85	85.0000000000000000
5	5	2025-04-07 00:55:30.909853	95	95.0000000000000000
6	6	2025-04-07 00:55:30.909853	60	60.0000000000000000
7	7	2025-04-07 00:55:30.909853	92	92.0000000000000000
8	8	2025-04-07 00:55:30.909853	75	75.0000000000000000
9	9	2025-04-07 00:55:30.909853	80	80.0000000000000000
10	10	2025-04-07 00:55:30.909853	88	88.0000000000000000
11	11	2025-04-07 00:55:30.909853	77	77.0000000000000000
12	12	2025-04-07 00:55:30.909853	90	90.0000000000000000
13	13	2025-04-07 00:55:30.909853	82	82.0000000000000000
14	14	2025-04-07 00:55:30.909853	91	91.0000000000000000
15	15	2025-04-07 00:55:30.909853	79	79.0000000000000000

Description:

- **AVG(score) OVER (… ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)** computes the average of the current row and the two prior rows.
- **PARTITION BY user_id** calculates separately for each user.
- This smooths out short-term fluctuations in a user's performance.

Task 3

This query smooths out score fluctuations by calculating a moving average of the latest 3 attempts per user. `AVG(score) OVER (...)` calculates the average score of the current and 2 previous attempts. `PARTITION BY user_id` ensures each user's scores are calculated separately. `ORDER BY attempt_date` defines the order of the attempts. This helps visualize a user's recent performance trend.

PART 2 .
WINDOW FUNCTIONS
30 points

◇ Task 1: GROUPING SETS - sum of two columns and grand total

```sql
SELECT
    creator_id,
    quiz_id,
    SUM(max_score) AS total_score
FROM quiz
GROUP BY GROUPING SETS (
    (creator_id),
    (quiz_id),
    (creator_id, quiz_id),
    ()
);
```

	creator_id integer	quiz_id [PK] integer	total_score bigint
1	[null]	[default]	1475
2	1	10	100
3	1	1	100
4	3	13	130
5	6	9	200
6	2	8	50
7	5	6	100
8	3	3	120
9	4	11	60
10	2	2	80
11	5	12	40
12	4	5	150
13	3	7	85
14	2	15	70
15	6	14	100
16	1	4	90
17	4	[default]	210
18	6	[default]	300
19	2	[default]	200
20	3	[default]	335

Description:

- `GROUPING SETS ((col1), (col2), (col1, col2), ())` explicitly defines groupings for aggregation.
- `SUM(score)` computes total score for each grouping.
- The empty `()` set gives a grand total.
- Useful to see totals at multiple levels in a single result set.

This query summarizes scores across different groupings using `GROUPING SETS`. It shows totals by individual columns (e.g. quizzes, users), their combinations, and includes the grand total. This gives a flexible overview of aggregated data across multiple dimensions — like slicing a data cake in all directions 🍰.

◇ Task 2: GROUPING SETS — row count with grouping level

```sql
SELECT
    creator_id,
    quiz_id,
    COUNT(*) AS total_attempts,
    GROUPING(creator_id) AS grp_creator,
    GROUPING(quiz_id) AS grp_quiz
FROM quiz
GROUP BY GROUPING SETS (
    (creator_id),
    (creator_id, quiz_id),
    ()
);
```

	creator_id integer	quiz_id [PK] integer	total_attempts bigint	grp_creator integer	grp_quiz integer
1	[null]	[default]	15	1	1
2	1	10	1	0	0
3	1	1	1	0	0
4	3	13	1	0	0
5	6	9	1	0	0
6	2	8	1	0	0
7	5	6	1	0	0
8	3	3	1	0	0
9	4	11	1	0	0
10	2	2	1	0	0
11	5	12	1	0	0
12	4	5	1	0	0
13	3	7	1	0	0
14	2	15	1	0	0
15	6	14	1	0	0
16	1	4	1	0	0
17	4	[default]	2	0	1
18	6	[default]	2	0	1
19	2	[default]	3	0	1
20	3	[default]	3	0	1

Description:

- `GROUPING SETS` allows grouping by one column, two columns, and overall total.
- `COUNT(*)` returns number of rows per grouping.
- `GROUPING(col)` returns 1 if that column is **not** grouped (i.e., total level), and 0 if it **is**.
- Helps to track from detailed to total level.

This query counts the number of entries by different grouping levels using `GROUPING SETS`. It helps see how many entries exist per column, pair of columns, and in total. Adding the `GROUPING()` function makes it easy to identify which level each row belongs to — like using CTRL+SHIFT+G in your brain 🧠.

◇ Task 3: CUBE - all possible sums in three columns

```sql
SELECT
  creator_id,
  quiz_id,
  visibility,
  SUM(max_score) AS total_score,
  GROUPING(creator_id) AS is_creator_null,
  GROUPING(quiz_id) AS is_quiz_null,
  GROUPING(visibility) AS is_date_null
FROM quiz
GROUP BY CUBE(creator_id, quiz_id, visibility)
ORDER BY creator_id, quiz_id, visibility;
```

	creator_id integer	quiz_id [PK] integer	visibility character varying (50)	total_score bigint	is_creator_null integer	is_quiz_null integer	is_date_null integer
1	1	1	public	100	0	0	0
2	1	1	[null]	100	0	0	1
3	1	4	public	90	0	0	0
4	1	4	[null]	90	0	0	1
5	1	10	public	100	0	0	0
6	1	10	[null]	100	0	0	1
7	1	[default]	public	290	0	1	0
8	1	[default]	[null]	290	0	1	1
9	2	2	public	80	0	0	0
10	2	2	[null]	80	0	0	1
11	2	8	public	50	0	0	0
12	2	8	[null]	50	0	0	1
13	2	15	public	70	0	0	0
14	2	15	[null]	70	0	0	1
15	2	[default]	public	200	0	1	0
16	2	[default]	[null]	200	0	1	1
17	3	3	private	120	0	0	0
18	3	3	[null]	120	0	0	1
19	3	7	public	85	0	0	0
20	3	7	[null]	85	0	0	1
21	3	13	public	130	0	0	0
22	3	13	[null]	130	0	0	1
23	3	[default]	private	120	0	1	0
24	3	[default]	public	215	0	1	0
25	3	[default]	[null]	335	0	1	1
26	4	5	private	150	0	0	0
27	4	5	[null]	150	0	0	1
28	4	11	private	60	0	0	0
29	4	11	[null]	60	0	0	1
30	4	[default]	private	210	0	1	0
31	4	[default]	[null]	210	0	1	1
32	5	6	public	100	0	0	0
33	5	6	[null]	100	0	0	1
34	5	12	public	40	0	0	0
35	5	12	[null]	40	0	0	1
36	5	[default]	public	140	0	1	0
37	5	[default]	[null]	140	0	1	1
38	6	9	private	200	0	0	0
39	6	9	[null]	200	0	0	1
40	6	14	private	100	0	0	0

Description:

• `CUBE(user_id, quiz_id, attempt_date)` generates **every combination** of the three columns — individual, paired, all three, and none.

• `SUM(score)` calculates total score for each combination.

• `GROUPING(col)` shows whether a column is **aggregated (1)** or used in the grouping (0).

• Helps distinguish between real NULL values and those generated by the cube.

This query gives a full picture of user performance by calculating every possible total and subtotal across users, quizzes, and dates. Want total score per quiz? Done. Per user per date? Done. Entire platform average? Also done. It's like enabling God Mode on SQL summaries 🎮📊.

Task 4 – CUBE with AVG

```sql
SELECT
  creator_id,
  quiz_id,
  user_id,
  AVG(max_score) AS avg_score,
  GROUPING(creator_id) AS g_creator,
  GROUPING(quiz_id) AS g_quiz,
  GROUPING(user_id) AS g_user
FROM quiz, userr
GROUP BY CUBE(creator_id, quiz_id, user_id)
ORDER BY creator_id, quiz_id, user_id;
```

	creator_id integer	quiz_id integer	user_id integer	avg_score numeric	g_creator integer	g_quiz integer	g_user integer
1	1	1	1	100.0000000000000000	0	0	0
2	1	1	2	100.0000000000000000	0	0	0
3	1	1	3	100.0000000000000000	0	0	0
4	1	1	4	100.0000000000000000	0	0	0
5	1	1	5	100.0000000000000000	0	0	0
6	1	1	6	100.0000000000000000	0	0	0
7	1	1	7	100.0000000000000000	0	0	0
8	1	1	8	100.0000000000000000	0	0	0
9	1	1	9	100.0000000000000000	0	0	0
10	1	1	10	100.0000000000000000	0	0	0
11	1	1	11	100.0000000000000000	0	0	0
12	1	1	12	100.0000000000000000	0	0	0
13	1	1	13	100.0000000000000000	0	0	0
14	1	1	14	100.0000000000000000	0	0	0
15	1	1	15	100.0000000000000000	0	0	0
16	1	1	[null]	100.0000000000000000	0	0	1
17	1	4	1	90.0000000000000000	0	0	0
18	1	4	2	90.0000000000000000	0	0	0
19	1	4	3	90.0000000000000000	0	0	0
20	1	4	4	90.0000000000000000	0	0	0

573	[null]	15	13	70.0000000000000000	1	0	0
574	[null]	15	14	70.0000000000000000	1	0	0
575	[null]	15	15	70.0000000000000000	1	0	0
576	[null]	15	[null]	70.0000000000000000	1	0	1
577	[null]	[null]	1	98.3333333333333333	1	1	0
578	[null]	[null]	2	98.3333333333333333	1	1	0
579	[null]	[null]	3	98.3333333333333333	1	1	0
580	[null]	[null]	4	98.3333333333333333	1	1	0
581	[null]	[null]	5	98.3333333333333333	1	1	0
582	[null]	[null]	6	98.3333333333333333	1	1	0
583	[null]	[null]	7	98.3333333333333333	1	1	0
584	[null]	[null]	8	98.3333333333333333	1	1	0
585	[null]	[null]	9	98.3333333333333333	1	1	0
586	[null]	[null]	10	98.3333333333333333	1	1	0
587	[null]	[null]	11	98.3333333333333333	1	1	0
588	[null]	[null]	12	98.3333333333333333	1	1	0
589	[null]	[null]	13	98.3333333333333333	1	1	0
590	[null]	[null]	14	98.3333333333333333	1	1	0
591	[null]	[null]	15	98.3333333333333333	1	1	0
592	[null]	[null]	[null]	98.3333333333333333	1	1	1

Description:

This command uses CUBE to collect aggregated data by three dimensions: user_id, creator_id, and feedback_rating. It automatically generates all possible grouping combinations — by one column, by two columns, or even by all of them.

This is useful when you need to understand:

which course has received more positive ratings,

which instructor has more feedback,

and which feedback ratings are more common overall.

GROUPING(...) also helps determine which rows are totals, not regular data.

In short, it's like an Excel spreadsheet where you click "Pivot table" and now you can play around with it: "What happens if I count separately by courses, separately by instructors, and then mix everything up?" A kind of SQL shashlik.

Task 5 (ROLLUP) — another table and meaning

```sql
SELECT
    feedback_category,
    COUNT(*) AS feedback_count,
    SUM(COUNT(*)) OVER (ORDER BY feedback_category) AS running_total
FROM feedback
GROUP BY feedback_category
ORDER BY feedback_category;
```

	feedback_category character varying (50)	feedback_count bigint	running_total numeric
1	Content	2	2
2	General	8	10
3	Interface	1	11
4	Timing	2	13

📓 Description:

- `COUNT(*)` counts how many feedback entries exist for each `feedback_category` (like "bug report", "suggestion", "compliment", etc.).
- `SUM(COUNT(*)) OVER (...)` is a **window function** that calculates a **running total** — the total number of feedbacks *up to and including* each category, in alphabetical order.
- `GROUP BY feedback_category` groups rows by each distinct category, so the count applies correctly.
- `ORDER BY feedback_category` makes the output organized and consistent.

This is useful when you want to track feedback statistics cumulatively and see how feedback grows by categories — for example, to quickly understand if complaints are dominating over suggestions 🧐.

Task 6 – Compare Points and Time Taken per Result

```sql
SELECT
  quiz_id,
  SUM(score) AS total_score,
  SUM(time_taken) AS total_time
FROM result
GROUP BY ROLLUP(quiz_id);
```

	quiz_id integer	total_score bigint	total_time bigint
1	[null]	1234	1870
2	3	379	575
3	2	428	665
4	1	427	630

📝 Description:

This query uses the **`ROLLUP` operator** to generate subtotals and grand totals when grouping data. It's particularly useful for reports or dashboards where you want to see **aggregated data at multiple levels** — for example, per category, per quiz, or for the entire dataset.

In our case, we're grouping results by the `quiz_id` column and calculating **three different aggregates**:

- Total `score` per quiz
- Total `time_taken` per quiz
- Their **combined total** — which appears as a **grand total row** when `quiz_id` is `NULL`.

The use of `COALESCE(quiz_id, 'TOTAL')` makes the output cleaner and readable — instead of showing `NULL` for subtotals and totals, we replace them with the label `'TOTAL'`.

This query gives both **insight into individual quiz performance** and a **quick overall summary**, all in a single, elegant result set. It's especially helpful for admin dashboards or reports in education platforms, business intelligence, or any system tracking performance across grouped entities.

🧠 Why ROLLUP rocks:

- Saves time — no need to manually UNION separate total queries.
- Makes it easier to create pivot-table-style reports.
- Useful in PostgreSQL for real-time analytics and performance tracking.

.

▨ Task 7 — Two Queries for Analytics on Quiz Difficulty and Engagement

☑ Query 1 – Start with P

```
SELECT *
FROM reward
WHERE reward_name LIKE 'B%';
```

```
SELECT reward_id, reward_name, points_required
FROM reward
WHERE reward_name LIKE 'P%';
```

	reward_id [PK] integer	reward_name character varying (100)	points_required integer
1	9	Perfect Score	1000

	reward_id [PK] integer	user_id integer	reward_name character varying (100)	points_required integer	granted_at timestamp without time zone	reward_type character varying (50)	status character varying (50)
1	3	3	Bronze Badge	500	2025-04-07 00:57:01.032462	Achievement	Granted
2	12	12	Bonus Points	400	2025-04-07 00:57:01.032462	Access	Granted

▨ Description:

• This query filters the `reward` table to find all rewards whose names start with the letter **"P" and “B”**.— to display the **unique identifier**, the **name**, and the **cost in points** for each reward.

This query is great for **searching structured names** or filtering data based on naming patterns — especially in systems with many entries that follow branding or category prefixes. It’s like searching for all items in a game store that start with “Legendary” or “Mega”. 🔍

КАК ВЫ МОГЛИ ЗАМЕТИТЬ Я ЗДЕСЬ ИСПОЛЬЗОВАЛ 2 ПРИМЕРА

▨ Task 7 — Two Queries for Analytics on Quiz Difficulty and Engagement

☑ Query 2 – Total Submissions Across Difficulty Levels

```sql
SELECT reward_id, reward_name, points_required
FROM reward
WHERE points_required < 1000;
```

```sql
SELECT *
FROM reward
WHERE points_required < 300;
```

	reward_id [PK] integer	reward_name character varying (100)	points_required integer
1	2	Silver Badge	750
2	3	Bronze Badge	500
3	6	Exclusive Content	300
4	7	VIP Access	500
5	8	Early Bird	200
6	10	Super User	800
7	12	Bonus Points	400
8	14	Expert Status	600
9	15	Exclusive Badge	900

	reward_id [PK] integer	user_id integer	reward_name character varying (100)	points_required integer	granted_at timestamp without time zone	reward_type character varying (50)	status character varying (50)
1	8	8	Early Bird	200	2025-04-07 00:57:01.032462	Access	Granted

points_required < 1000 filters by number of points.

КАК ВИДИТЕ ЗДЕСЬ ТОЖЕ 2 ПРИМЕРА 😊

▨ Task 7 — Two Queries for Analytics on Quiz Difficulty and Engagement

☑ INTERSECT — Final Result

```sql
SELECT reward_id, reward_name, points_required
FROM reward
WHERE reward_name LIKE 'P%'

INTERSECT

SELECT reward_id, reward_name, points_required
FROM reward
WHERE points_required < 1000;
```

```sql
SELECT *
FROM reward
WHERE reward_name LIKE 'B%'
INTERSECT
SELECT *
FROM reward
WHERE points_required < 500;
```

	reward_id integer	reward_name character varying (100)	points_required integer

	reward_id integer	user_id integer	reward_name character varying (100)	points_required integer	granted_at timestamp without time zone	reward_type character varying (50)	status character varying (50)
1	12	12	Bonus Points	400	2025-04-07 00:57:01.032462	Access	Granted

INTERSECT returns only those rows that satisfy both conditions at the same time.

This is useful if you want to find cheap rewards that start with, for example, "Bonus" or "Badge".

И ЗАКАНЧИВАЕМ МЫ ТОЖЕ 2 ПРИМЕРАМИ 😊

В ходе работы над заданиями по SQL я последовательно выполнил 7 заданий, каждое из которых расширяло мои навыки в работе с реляционными базами данных и запросами в PostgreSQL. Первое задание заключалось в использовании агрегатной функции для анализа максимального балла по уровням сложности викторин. Мне особенно понравилось использовать `GROUP BY` и `ROUND`, а также сортировку по убыванию среднего значения — это позволило быстро определить, насколько "щедры" викторины разных уровней.

Каждое из заданий дало мне новые знания и удовольствие от построения грамотных, читаемых и эффективных SQL-запросов. Особенно приятно было оформлять запросы с пояснениями, превращая код в инструмент анализа и управления данными.

Эти 6 assignmentы помогли мне постичь мудрость и фичи работы с базами данных. И я с уверенностью могу заявить, что уже готов работать с масштабными проектами

СПАСИБО ВАМ 😊

p/s

Sagacious AKADIL