**SQL Bug Fixing: Fix the QUERY - Totaling**

Oh no! Timmys been moved into the database divison of his software company but as we know Timmy loves making mistakes. Help Timmy keep his job by fixing his query...

Timmy works for a statistical analysis company and has been given a task of totaling the number of sales on a given day grouped by each department name and then each day.

Resultant table:

day (type: date) {group by} [order by asc]

department (type: text) {group by} [In a real world situation it is bad practice to name a column after a table]

sale\_count (type: int)

Tables and relationship below:

A diagram of a data flow

Description automatically generated with medium confidence

**Solution:**

**SELECT DISTINCT DATE (s.transaction\_date) AS day, d.name AS department, COUNT(s.id) AS sale\_count**

**FROM department d**

**INNER JOIN sale s ON d.id = s.department\_id**

**GROUP BY d.name, day**

**ORDER BY day ASC**

**Calculating Batting Average**

In baseball, the batting average is a simple and most common way to measure a hitter's performace. Batting average is calculated by taking all the players hits and dividing it by their number of at\_bats, and it is usually displayed as a 3 digit decimal (i.e. 0.300).

Given a yankees table with the following schema,

-player\_id STRING

-player\_name STRING

-primary\_position STRING

-games INTEGER

-at\_bats INTEGER

-hits INTEGER

return a table with player\_name, games, and batting\_average.

We want batting\_average to be rounded to the nearest thousandth, since that is how baseball fans are used to seeing it. Format it as text and make sure it has 3 digits to the right of the decimal (pad with zeroes if neccesary).

Next, order our resulting table by batting\_average, with the highest average in the first row.

Finally, since batting\_average is a rate statistic, a small number of at\_bats can change the average dramatically. To correct for this, exclude any player who doesn't have at least 100 at bats.

Expected Output Table

-player\_name STRING

-games INTEGER

-batting\_average STRING

**Solution:**

**SELECT player\_name, games, CAST(ROUND(hits::NUMERIC / at\_bats, 3) AS DECIMAL(10,3))::TEXT AS batting\_average**

**FROM yankees**

**WHERE at\_bats >= 100**

**ORDER BY batting\_average DESC**

**SQL: Disorder**

You are given a table numbers with just one column, number. It holds some numbers that are already ordered.

You need to write a query that makes them un-ordered, as in, every possible ordering should appear equally often.

**Solution:**

**SELECT \***

**FROM numbers**

**ORDER BY RANDOM()**

**SQL with Pokemon: Damage Multipliers**

You have arrived at the Celadon Gym to battle Erika for the Rainbow Badge.

She will be using Grass-type Pokemon. Any fire pokemon you have will be strong against grass, but your water types will be weakened. The multipliers table within your Pokedex will take care of that.

Using the following tables, return the pokemon\_name, modifiedStrength and element of the Pokemon whose strength, after taking these changes into account, is greater than or equal to 40, ordered from strongest to weakest.

pokemon schema

id

pokemon\_name

element\_id

str

multipliers schema

id

element

multiplier

**Solution:**

**SELECT p.pokemon\_name, (p.str \* m.multiplier) AS modifiedStrength, m.element**

**FROM pokemon p**

**LEFT JOIN multipliers m ON p.element\_id = m.id**

**WHERE modifiedStrength >= 40**

**ORDER BY modifiedStrength DESC**

**SQL Basics: Simple table totaling**

For this challenge you need to create a simple query to display each unique clan with their total points and ranked by their total points.

people table schema

name

points

clan

You should then return a table that resembles below

select on

rank

clan

total\_points

total\_people

The query must rank each clan by their total\_points, you must return each unqiue clan and if there is no clan name (i.e. it's an empty string) you must replace it with [no clan specified], you must sum the total\_points for each clan and the total\_people within that clan.

##Note The data is loaded from the live leaderboard, this means values will change but also could cause the kata to time out retreiving the information.

**Solution:**

**SELECT DISTINCT ROW\_NUMBER() OVER (ORDER BY SUM(points) DESC) AS rank,**

**COALESCE(NULLIF(clan,''), '[no clan specified]') AS clan,**

**SUM(points) AS total\_points,**

**COUNT(name) AS total\_people**

**FROM people**

**GROUP BY clan**

**ORDER BY SUM(points) DESC**

**SQL Basics - Monsters using CASE**

You have access to two tables named top\_half and bottom\_half, as follows:

top\_half schema

id

heads

arms

bottom\_half schema

id

legs

tails

You must return a table with the format as follows:

output schema

id

heads

legs

arms

tails

species

The IDs on the tables match to make a full monster. For heads, arms, legs and tails you need to draw in the data from each table.

For the species, if the monster has more heads than arms, more tails than legs, or both, it is a 'BEAST' else it is a 'WEIRDO'. This needs to be captured in the species column.

All rows should be returned (10).

Tests require the use of CASE. Order by species.

**Solution:**

**SELECT t.id, t.heads, t.arms, b.legs, b.tails,**

**(CASE**

**WHEN t.heads > t.arms OR b.tails > b.legs THEN 'BEAST'**

**ELSE 'WEIRDO'**

**END) AS species**

**FROM top\_half t JOIN bottom\_half b ON t.id = b.id**

**ORDER BY species**

**SQL Bug Fixing: Fix the JOIN**

Oh no! Timmys been moved into the database divison of his software company but as we know Timmy loves making mistakes. Help Timmy keep his job by fixing his query...

Timmy works for a statistical analysis company and has been given a task of calculating the highest average salary for a given job, the sample is compiled of 100 applicants each with a job and a salary. Timmy must display each unique job, the total average salary, the total people and the total salary and order by highest average salary. Timmy has some bugs in his query, help Timmy fix his query so he can keep his job!

people table schema

id

name

job table schema

id

people\_id

job\_title

salary

resultant table schema

job\_title (unique)

average\_salary (float, 2 dp)

total\_people (int)

total\_salary (float, 2 dp)

**Solution:**

**SELECT DISTINCT j.job\_title**

**,CAST(ROUND((SUM(j.salary) / COUNT(p)), 2) AS FLOAT) AS average\_salary**

**,COUNT(p.id) AS total\_people**

**,CAST(ROUND(SUM(j.salary), 2) AS FLOAT) AS total\_salary**

**FROM people p**

**INNER JOIN job j ON p.id = j.people\_id**

**GROUP BY j.job\_title**

**ORDER BY average\_salary DESC**

**SQL Basics: Simple PIVOTING data WITHOUT CROSSTAB**

This kata is inspired by SQL Basics: Simple PIVOTING data by matt c.

You need to build a pivot table WITHOUT using CROSSTAB function. Having two tables products and details you need to select a pivot table of products with counts of details occurrences (possible details values are ['good', 'ok', 'bad'].

Results should be ordered by product's name.

Model schema for the kata is:

your query should return table with next columns

name

good

ok

bad

Compare your table to the expected table to view the expected results.

**Solution:**

**SELECT p.name,**

**(SELECT COUNT(d.detail) FROM details d WHERE detail = 'good' AND p.id = d.product\_id ) as good,**

**(SELECT COUNT(d.detail) FROM details d WHERE detail = 'ok' AND p.id = d.product\_id) as ok,**

**(SELECT COUNT(d.detail) FROM details d WHERE detail = 'bad' AND p.id = d.product\_id) as bad**

**FROM products p INNER JOIN details d ON p.id = d.product\_id**

**GROUP BY p.name, p.id**

**ORDER BY p.name**

**SQL Basics: Simple Hierarchical structure**

NOTE: Most difficult query at the moment, required further investigation.

For this challenge you need to create a RECURSIVE Hierarchical query. You have a table employees of employees, you must order each employee by level. You must use a WITH statement and name it employee\_levels after that has been defined you must select from it.

A Level is in correlation what manager managers the employee. e.g. an employee with a manager\_id of NULL is at level 1 and then direct employees with the employee at level 1 will be level 2.

employees table schema

id

first\_name

last\_name

manager\_id (can be NULL)

resultant schema

level

id

first\_name

last\_name

manager\_id (can be NULL)

**Solution:**

**WITH RECURSIVE employee\_levels AS (**

**SELECT 1 AS level, id, first\_name, last\_name, manager\_id**

**FROM employees**

**WHERE manager\_id IS NULL**

**UNION ALL**

**SELECT (el.level + 1) AS level, e.id, e.first\_name, e.last\_name, e.manager\_id**

**FROM employees e**

**INNER JOIN employee\_levels el ON e.manager\_id = el.id**

**)**

**SELECT \***

**FROM employee\_levels**

**ORDER BY level, id;**

**SQL Basics: Group By Day**

There is an events table used to track different key activities taken on a website. For this task you need to:

find the entries whose name equals "trained"

group them by the day the activity happened (the date part of the created\_at timestamp) and their description's

the 2 aforementioned fields should be returned together with the number of grouped entries in a column called count

the result should also be sorted by day

"events" table schema

id (bigint)

name (text)

created\_at (timestamp)

description (text)

expected result schema

day (date)

description (text)

count (numeric)

**Solution:**

**SELECT DATE(created\_at) AS day, description, COUNT(description) AS count**

**FROM events**

**WHERE name = 'trained'**

**GROUP BY DATE(created\_at), description**

**ORDER BY DATE(created\_at)**

**SQL Basics: Simple HAVING**

For this challenge you need to create a simple HAVING statement, you want to count how many people have the same age and return the groups with 10 or more people who have that age.

people table schema

id

name

age

return table schema

age

total\_people

**Solution:**

**SELECT age, COUNT(age) AS total\_people**

**FROM people**

**GROUP BY age**

**HAVING COUNT(age) >= 10**

**Youngest Team Members**

You are working with a database that stores information about employees in a tech firm. The database includes a table named employees with the following columns:

employee\_id: A unique integer identifier for each employee.

full\_name: A string representing the employee's full name.

team: A string that specifies which team the employee is part of. The team can be one of the following four: "backend", "frontend", "devops", or "design".

birth\_date: A date that represents the employee's birthdate.

The company is planning an event where the youngest employee from each team will be given a chance to share their vision of future technology trends.

Your task is to write an SQL query that retrieves the complete record for the youngest member of each team. You should consider the person with the latest birthdate as the youngest. Let's assume for this task that the are no youngest employees who share the same birthdate.

The classical solution of using aggregate function and group by is forbidden. Can you come up with something more witty?

The result should be ordered by team in asc alphabetical order.

Good luck!

Desired Output

The desired output should look like this:

employee\_id full\_name team birth\_date

11 John Doe backend 1980-12-01

7 Jane Smith design 1985-05-03

24 Bob Jones devops 1990-04-15

54 Dana Smith frontend 1995-05-03

**Solution:**

**SELECT DISTINCT e.employee\_id, e.full\_name, e.team, e.birth\_date**

**FROM employees e**

**WHERE e.birth\_date >= ALL (**

**SELECT b.birth\_date**

**FROM employees b**

**WHERE b.team = e.team)**

**ORDER BY e.team, e.birth\_date**

**Dealing With Messy Data**

Your company has an internal policy to determine your customers' credit limit, but this procedure has been questioned recently by the board as being too conservative.

Your CEO wants to increase the current customer base credit limits in order to upsell a new line of products. In order to do that, the company hired several external consultancies to produce new credit limit estimates.

The problem is that each agency has produced the report in its own format. Some use the format "First-name Last-name" to identify a person, others use the format "Last-name, First-name". There is also no consensus on how to capitalize each word, so some used all uppercase, others used all lowercase, and some used mixed-case.

Also, some names are titled, for example: "Dr. Hannibal Lecter", "Robert Downey Jr." etc, so you will need to pay attention to any such or similar cases.

Internally, the data is structured as follows:

Table: customers

================

id: INT

first\_name: TEXT

last\_name: TEXT

credit\_limit: FLOAT

The data you've received from all agencies was consolidated in the following table:

Table: prospects

================

full\_name: TEXT

credit\_limit: FLOAT

Keep in mind that the agencies had access only to a ***partial*** customer base. There is also the possibility of more than one agency prospecting the same customer, so it's highly likely that there will be duplicates. Finally, they've prospected customers that were not in your customer base as well.

For this task you are interested in the prospected customers that are already in your customer base and the prospected credit limit is higher than your internal estimate. When more than one agency prospected the same customer, chose the highest estimate.

You have to produce a report with the following fields:

first\_name

last\_name

old\_limit [the current credit\_limit]

new\_limit [the highest credit\_limit found]

Good luck!

Notes:

* only list the customers that a higher credit limit was found.

**Solution: (not my solution ☹)**

**CREATE INDEX ON customers (lower(first\_name || ' ' || last\_name), lower(first\_name || ',' || last\_name));**

**CREATE INDEX ON prospects (lower(full\_name));**

**SELECT a.first\_name,**

**a.last\_name,**

**a.credit\_limit AS old\_limit,**

**max(b.credit\_limit) AS new\_limit**

**FROM customers a JOIN prospects b**

**ON lower(full\_name) IN (**

**lower(a.first\_name || ' ' || a.last\_name),**

**lower(a.last\_name || ', ' || a.first\_name)**

**)**

**GROUP BY a.id**

**HAVING max(b.credit\_limit) > a.credit\_limit**

**ORDER BY first\_name, last\_name**

My partial solution:

**SELECT DISTINCT c.first\_name, c.last\_name, c.credit\_limit AS old\_limit, MAX(p.credit\_limit) AS new\_limit**

**FROM customers c**

**JOIN prospects p ON UPPER(c.first\_name || ' ' || c.last\_name) = UPPER(p.full\_name)**

**WHERE p.credit\_limit IS NOT NULL**

**GROUP BY 1,2,3**

**HAVING MAX(p.credit\_limit) > c.credit\_limit**

**SQL Basics: Simple PIVOTING data**

For this challenge you need to PIVOT data. You have two tables, products and details. Your task is to pivot the rows in products to produce a table of products which have rows of their detail. Group and Order by the name of the Product.

Tables and relationship below:

A screen shot of a cell phone

Description automatically generated

products table schema

- id - integer

- name - text

details table schema

- id - integer

- product\_id - integer

- detail - text

You must use the CROSSTAB statement to create a table that has the schema as below:

CROSSTAB table schema

- name - text

- bad - bigint

- good - bigint

- ok - bigint

If the values aren't assigned to the last three columns within the query directly, it's assumed they will be presented in the lexicographical order (i.e. if we have three values, a, b and c, then bad, good and ok will have these values respectively).

Compare your table to the expected table to view the expected results.

**Solution:**

**CREATE EXTENSION tablefunc;**

**SELECT \***

**FROM CROSSTAB (**

**'SELECT p.name, d.detail, COUNT(\*)**

**FROM products p**

**JOIN details d ON p.id = d.product\_id**

**GROUP BY 1, 2**

**ORDER BY 1, 2',**

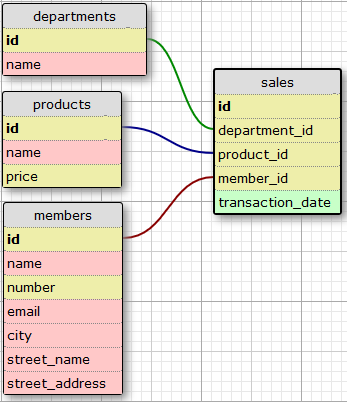
**'VALUES (''bad''::text), (''good''::text), (''ok''::text)'**

**) AS product\_pivot (name text, bad bigint, good bigint, ok bigint);**

**SQL Basics: Simple VIEW**

For this challenge you need to create a VIEW. This VIEW is used by a sales store to give out vouches to members who have spent over $1000 in departments that have brought in more than $10000 total ordered by the members id. The VIEW must be called members\_approved\_for\_voucher then you must create a SELECT query using the view.

**Tables and relationship below:**



resultant table schema

* id
* name
* email
* total\_spending

**Solution:**

**CREATE VIEW members\_approved\_for\_voucher**

**AS**

**SELECT m.id**

**,m.name**

**,m.email**

**,SUM(p.price) AS total\_spending**

**FROM members m**

**INNER JOIN sales s ON s.member\_id = m.id**

**RIGHT JOIN products p ON s.product\_id = p.id**

**RIGHT JOIN departments d ON s.department\_id = d.id**

**WHERE 10000 <= (**

**SELECT SUM(p2.price)**

**FROM products p2**

**INNER JOIN sales s2 ON p2.id = s2.product\_id**

**INNER JOIN departments d2 ON s2.department\_id = d2.id**

**WHERE d2.id = s.department\_id**

**)**

**GROUP BY 1**

**,2**

**,3**

**HAVING SUM(p.price) > 1000**

**ORDER BY m.id;**

**SELECT \***

**FROM members\_approved\_for\_voucher;**

**Challenge: Two actors who cast together the most**

Given the the schema presented below find two actors who cast together the most and list titles of only those movies they were casting together. Order the result set alphabetically by the movie title.

**Table film\_actor**

Column | Type | Modifiers

------------+-----------------------------+----------

actor\_id | smallint | not null

film\_id | smallint | not null

...

**Table actor**

Column | Type | Modifiers

------------+-----------------------------+----------

actor\_id | integer | not null

first\_name | character varying(45) | not null

last\_name | character varying(45) | not null

...

**Table film**

Column | Type | Modifiers

------------+-----------------------------+----------

film\_id | integer | not null

title | character varying(255) | not null

...

**The desired output:**

first\_actor | second\_actor | title

------------+--------------+--------------------

John Doe | Jane Doe | The Best Movie Ever

...

* first\_actor - Full name (First name + Last name separated by a space)
* second\_actor - Full name (First name + Last name separated by a space)
* title - Movie title

**Note:** actor\_id of the first\_actor should be lower then actor\_id of the second\_actor

**Solution:**

**WITH full\_actor AS ( --~ Name in the proper form**

**SELECT actor\_id, CONCAT(first\_name,' ',last\_name) AS name**

**FROM actor**

**),**

**full\_film AS ( --~ All films with their actors**

**SELECT a.actor\_id, a.name, f.film\_id, f.title**

**FROM full\_actor a**

**INNER JOIN film\_actor fa USING(actor\_id)**

**INNER JOIN film f USING(film\_id)**

**),**

**movie\_counts AS ( --~ Count coincidences between actors**

**SELECT f1.actor\_id AS actor1\_id, f2.actor\_id AS actor2\_id, COUNT(\*) AS movie\_count**

**FROM full\_film f1**

**JOIN full\_film f2 ON f1.film\_id = f2.film\_id AND f1.actor\_id < f2.actor\_id**

**GROUP BY f1.actor\_id, f2.actor\_id**

**)**

**SELECT actor1.name AS first\_actor, actor2.name AS second\_actor, actor1.title**

**FROM full\_film actor1**

**JOIN full\_film actor2 ON actor1.film\_id = actor2.film\_id AND actor1.actor\_id < actor2.actor\_id**

**JOIN movie\_counts mc ON actor1.actor\_id = mc.actor1\_id AND actor2.actor\_id = mc.actor2\_id**

**WHERE mc.movie\_count = (SELECT MAX(movie\_count) FROM movie\_counts)**

**ORDER BY actor1.title;**

**Using Window Functions To Get Top N per Group**

**Description**

Given the schema presented below write a query, which uses a **window function**, that returns two most viewed posts for every category.

Order the result set by:

1. category name alphabetically
2. number of post views largest to lowest
3. post id lowest to largest

Note:

* Some categories may have less than two or no posts at all.
* Two or more posts within the category can be tied by (have the same) the number of views. Use post id as a tie breaker - a post with a lower id gets a higher rank.

**Schema**

**categories**

Column | Type | Modifiers

------------+-----------------------------+----------

id | integer | not null

category | character varying(255) | not null

**posts**

Column | Type | Modifiers

------------+-----------------------------+----------

id | integer | not null

category\_id | integer | not null

title | character varying(255) | not null

views | integer | not null

**Desired Output**

The desired output should look like this:

category\_id | category | title | views | post\_id

------------+----------+-----------------------------------+-------+--------

5 | art | Most viewed post about Art | 9234 | 234

5 | art | Second most viewed post about Art | 9234 | 712

2 | business | NULL | NULL | NULL

7 | sport | Most viewed post about Sport | 10 | 126

...

* category\_id - category id
* category - category name
* title - post title
* views - the number of post views
* post\_id - post id

**Solution:**

**-- Common Table Expression (CTE) to rank posts within each category**

**WITH RankedPosts AS (**

**SELECT c.id AS category\_id,**

**c.category,**

**p.id AS post\_id,**

**p.views AS views,**

**p.title,**

**-- Using ROW\_NUMBER() to assign a rank to each post within its category**

**ROW\_NUMBER() OVER (PARTITION BY p.category\_id ORDER BY p.views DESC, p.id) AS row\_num**

**FROM categories c**

**LEFT JOIN posts p ON c.id = p.category\_id**

**GROUP BY c.id, c.category, p.title, p.id**

**)**

**SELECT category\_id,**

**category,**

**title,**

**views,**

**post\_id**

**FROM RankedPosts**

**WHERE row\_num <= 2 -- two most viewed posts**

**ORDER BY category, views DESC, post\_id;**

**Calculating Running Total:**

**Description**

Given a posts table that contains a created\_at timestamp column write a query that returns date (without time component), a number of posts for a given date and a running (cumulative) total number of posts up until a given date. The resulting set should be ordered chronologically by date.

**Desired Output**

The resulting set should look similar to the following:

date | count | total

-----------+-------+-------

2017-01-26 | 20 | 20

2017-01-27 | 17 | 37

2017-01-28 | 7 | 44

2017-01-29 | 8 | 52

...

* date - (DATE) date
* count - (INT) number of posts for a date
* total - (INT) a running (cumulative) number of posts up until a date

**Solution:**

**SELECT created\_at::date AS date, COUNT(\*) AS count,**

**CAST (SUM(COUNT(\*)) OVER (ORDER BY created\_at::date) AS integer) AS total**

**FROM posts**

**GROUP BY 1;**

**Maximum Daily Sales by Product and Store:**

Imagine a company that has many stores, and each store sells many different products. Each row in the sales table represents a single sale transaction in a specific store on a specific date, including the product sold, the quantity sold, and the unit price.

The table has the following columns:

* id (integer): A unique identifier for each record in the table.
* store\_id (integer): The identifier of the store where the transaction took place.
* product\_id (integer): The identifier of the product that was sold.
* quantity (integer): The number of units of the product that were sold in this transaction.
* transaction\_date (date): The date when the transaction took place.
* price (float): The price per unit of the product for this transaction.

Your task is to write a SQL query that, for each product and store combination, finds the date of the highest single-transaction sale (i.e., the transaction in which the most units of that product were sold in one go), and the total quantity of that product sold on that day. If there is a tie for the maximum\_quantity, we need to select the earliest date when this maximum quantity happened.

The query should return a result set with the following columns:

* store\_id: The identifier of the store.
* product\_id: The identifier of the product.
* transaction\_date: The date of the highest single-transaction sale for this store and product. If there are multiple dates with the highest single-transaction sale, we need to take the earliest.
* max\_quantity: The quantity sold in the highest single-transaction sale for this store and product.
* total\_quantity\_on\_max\_day: The total quantity of this product sold in the store on the date of the highest single-transaction sale.

The result set should be ordered by store\_id in ascending order, and then by product\_id in ascending order.

Let's provide a concrete example. Here's a simplified version of what the sales table might look like:

+----+----------+------------+----------+-------------------+-------+

| id | store\_id | product\_id | quantity | transaction\_date | price |

+----+----------+------------+----------+-------------------+-------+

| 1 | 1 | 1 | 3 | 2023-06-01 | 5 |

| 2 | 1 | 1 | 7 | 2023-06-01 | 5 |

| 3 | 1 | 1 | 5 | 2023-06-02 | 5 |

| 4 | 2 | 1 | 2 | 2023-06-01 | 7 |

| 5 | 2 | 1 | 10 | 2023-06-02 | 7 |

| 6 | 2 | 2 | 4 | 2023-06-01 | 3 |

| 7 | 2 | 2 | 6 | 2023-06-01 | 3 |

+----+----------+------------+----------+-------------------+-------+

Let's break down the query result:

* For store 1 and product 1, the highest single-transaction sale day was 2023-06-01 with a quantity of 7. On that day, the total quantity sold was 10 (7 from the highest single sale and 3 from the other transaction).
* For store 2 and product 1, the highest single-transaction sale day was 2023-06-02 with a quantity of 10.
* For store 2 and product 2, the highest single-transaction sale day was 2023-06-01 with a quantity of 6. On that day, the total quantity sold was 10 (6 from the highest single sale and 4 from the other transaction).

So the result of the query would be:

+----------+------------+-------------------+--------------+---------------------------+

| store\_id | product\_id | transaction\_date | max\_quantity | total\_quantity\_on\_max\_day |

+----------+------------+-------------------+--------------+---------------------------+

| 1 | 1 | 2023-06-01 | 7 | 10 |

| 2 | 1 | 2023-06-02 | 10 | 10 |

| 2 | 2 | 2023-06-01 | 6 | 10 |

+----------+------------+-------------------+--------------+---------------------------+

**Solution:**

**WITH ranked\_sales AS (**

**SELECT DISTINCT store\_id,**

**product\_id,**

**transaction\_date,**

**quantity,**

**RANK() OVER (PARTITION BY store\_id, product\_id ORDER BY quantity DESC, transaction\_date) AS rnk**

**FROM sales**

**),**

**total\_quantity AS (**

**SELECT \*, (SUM(quantity) OVER (PARTITION BY store\_id, product\_id, transaction\_date)) AS total\_quantity\_on\_max\_day**

**FROM sales**

**)**

**SELECT DISTINCT r.store\_id,**

**r.product\_id,**

**r.transaction\_date,**

**MAX(r.quantity) AS max\_quantity,**

**t.total\_quantity\_on\_max\_day**

**FROM ranked\_sales r**

**INNER JOIN total\_quantity t ON (r.store\_id = t.store\_id AND r.product\_id = t.product\_id AND r.transaction\_date = t.transaction\_date)**

**WHERE r.rnk = 1**

**GROUP BY r.store\_id, r.product\_id, r.transaction\_date, t.total\_quantity\_on\_max\_day**

**ORDER BY r.store\_id, r.product\_id;**

**--SIMPLER SOLUTION:  
SELECT DISTINCT ON (store\_id, product\_id)**

**store\_id,**

**product\_id,**

**transaction\_date,**

**MAX(quantity) AS max\_quantity,**

**SUM(quantity) AS total\_quantity\_on\_max\_day**

**FROM sales**

**GROUP BY store\_id, product\_id, transaction\_date**

**ORDER BY store\_id, product\_id, max\_quantity DESC, transaction\_date ASC**

**SQL Statistics: MIN, MEDIAN, MAX**

For this challenge you need to create a simple SELECT statement. Your task is to calculate the MIN, MEDIAN and MAX scores of the students from the results table.

Tables and relationship below:

A close-up of a graph

Description automatically generated

**Solution:**

**SELECT MIN(score),**

**PERCENTILE\_CONT(0.5) WITHIN GROUP (ORDER BY (score)) AS median,**

**MAX(score)**

**FROM result;**