

ANSWER KEY

SQL, Databases & Pandas - Evaluation (Attempt 2)

32 questions · 8 topics · correct answers with explanations

CodeWithMunnaX

TOPICS

1. SQL Window Functions Q1, Q3, Q7, Q8, Q25, Q28
2. Joins & Set Operations Q2, Q4, Q10, Q21, Q30
3. Filtering, Sorting & Basic Queries Q5, Q6, Q9
4. CTEs & Query Structure Q15, Q18
5. Normalization Q11, Q12, Q14, Q27
6. Keys, Constraints & Data Types Q17, Q20, Q26, Q29
7. Pandas Operations Q13, Q16, Q19, Q22, Q23, Q24
8. Case Studies (Subjective) Q31, Q32

TOPIC 1 SQL Window Functions

6 Qs

Q1

Single choice

A dataset records each student's monthly test score across several months. An analyst wants to compute, for every student separately, the change in score compared to that same student's previous month -- without mixing one student's previous score into another student's row. Which window function specification correctly achieves this?

A AVG(score) OVER (PARTITION BY student_name) with no ordering applied at all

B LAG(score, 1) OVER (ORDER BY test_month) with no PARTITION BY clause at all

C LAG(score, 1) OVER (PARTITION BY student_name ORDER BY test_month)

✓ Correct

D GROUP BY student_name, test_month, collapsing all rows per group

EXPLANATION

PARTITION BY student_name confines the LAG calculation to each student's own rows, so the previous-month lookup never crosses into another student's data. Omitting PARTITION BY would let the previous row belong to a completely different student, an AVG window computes an average rather than a prior-month lag value, and a plain GROUP BY would collapse the monthly rows instead of preserving row-by-row comparisons.

Q3

Single choice

A teacher lists exam scores in descending order: 88, 85, 85, 82, 80. If `DENSE_RANK()` is applied over this ordering, what rank is assigned to the student who scored 82?

☐ A 2☐ B 4☒ C 3

✓ Correct

☐ D 5**EXPLANATION**

`DENSE_RANK` assigns consecutive ranks without skipping any after a tie: 88 gets rank 1, both students scoring 85 tie for rank 2, and the next distinct score (82) gets rank 3 -- not rank 4, which is what `RANK` would assign because it skips a rank after a tie.

Q7

Single choice

A retail analyst wants to compute month-over-month sales growth for a store by comparing this month's sales to last month's sales, using a monthly sales table ordered by month. Which window function retrieves the previous month's sales value into the same row?

☐ A GROUP BY month, which aggregates rows☐ B LEAD(sales, 1) OVER (ORDER BY month)☐ C RANK() OVER (ORDER BY month) ranking☒ D LAG(sales, 1) OVER (ORDER BY month)

✓ Correct

EXPLANATION

`LAG` looks backward in the time sequence and pulls in the value from the previous row, which is exactly what is needed to compare this month against last month. `LEAD` instead looks forward to a future row, `RANK` only assigns an ordering position rather than retrieving a sales value, and `GROUP BY` would collapse the monthly rows entirely rather than aligning a prior month's value alongside the current one.

Q8

Single choice

An analyst wants to show each employee's individual salary alongside their department's average salary, in the same row, without collapsing the individual employee rows. Which approach fits this requirement?

- ☐ A Use only SELECT DISTINCT on the salary column, without any aggregation logic
- ☐ B Use AVG(salary) with GROUP BY department, since it always preserves every employee row
- ☒ C Use AVG(salary) with OVER and PARTITION BY, appended as a new column ✓ Correct
- ☐ D Use ORDER BY salary DESC and read off the top row per department

EXPLANATION

The OVER clause with PARTITION BY computes the aggregate per group while appending it as a new column, keeping every original employee row intact. GROUP BY, by contrast, collapses rows into one per department, so it cannot preserve individual employee rows; SELECT DISTINCT only removes duplicate salary values and does not compute an average; sorting by salary does not compute or attach any departmental average.

Q25

Multiple select

An analyst wants to smooth a noisy daily rainfall time series by computing, for each day, the average of that day plus the 2 days before and 2 days after it (a 5-day window). Assume the table has exactly one row for every calendar day in the period, with no missing dates. Which statements about this task are correct? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

- ☐ A This clause requires the data to first be aggregated using GROUP BY before it can be applied
- ☒ B This technique is used to remove sharp fluctuations and highlight the underlying trend in the time series ✓ Correct
- ☒ C This clause requires an ORDER BY inside the OVER() clause to define which rows count as PRECEDING and FOLLOWING ✓ Correct
- ☒ D AVG(rainfall) OVER (ORDER BY day ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING) correctly computes this 5-day centered window ✓ Correct

EXPLANATION

The ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING window correctly captures the current day plus the two days on either side, and moving averages exist precisely to smooth out sharp fluctuations and reveal the

underlying trend. This clause does require an ORDER BY inside OVER() to define the row sequence PRECEDING/FOLLOWING refers to, and it does not require any prior GROUP BY aggregation to be applied.

Q28

Multiple select

A company wants to rank students within each batch separately (not overall across all batches). Which of the following correctly achieve this batch-specific ranking? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

- ☒ **A** RANK() OVER (PARTITION BY batch ORDER BY score DESC) -- ranks restart for each batch ✓ **Correct**
- ☐ **B** RANK() OVER (PARTITION BY score ORDER BY batch DESC) -- ranks students by score within each batch
- ☐ **C** RANK() OVER (ORDER BY score DESC) without PARTITION BY -- ranks students only within their own batch automatically
- ☒ **D** DENSE_RANK() OVER (PARTITION BY batch ORDER BY score DESC) -- using consecutive ranks with no gaps after ties ✓ **Correct**

EXPLANATION

Using PARTITION BY batch correctly restarts the ranking for each batch, and this works whether RANK() (which skips ranks after ties) or DENSE_RANK() (which does not skip ranks) is used. Leaving out PARTITION BY entirely ranks across the whole dataset rather than per batch, and swapping the roles of batch and score in the partition/order clauses ranks by batch within each score value instead of the intended batch-specific score ranking.

TOPIC 2 Joins & Set Operations

5 Qs

Q2

Single choice

A Students table is LEFT JOINed to a Marks table (Students as the left table), so every student appears even if they have no marks recorded yet. For a student with no matching row in Marks, what will the Marks-table columns show in the joined result?

- ☒ **A** NULL values for every Marks column ✓ **Correct**
- ☐ **B** Zero values for every Marks column

C The row is excluded entirely from the output

D The most recent mark from another student

EXPLANATION

A LEFT JOIN keeps every row from the left table (Students); when there is no matching row in the right table (Marks), the right table's columns are filled with NULL instead of being dropped or replaced with a numeric placeholder like zero, and no other student's data is ever substituted in.

Q4

Single choice

A streaming platform stores `shows` and `watch_history` tables. To find every show that has never been watched by any user, which querying approach correctly identifies them?

A LEFT JOIN shows to watch_history on show_id, then filter for NULL watch_history rows

✓ Correct

B INNER JOIN shows to watch_history on show_id, then filter for the earliest watch date

C CROSS JOIN shows with watch_history and count the resulting rows

D UNION ALL shows and watch_history and remove duplicate show names

EXPLANATION

This is the anti-join pattern: a LEFT JOIN keeps every show, and shows with no matching watch_history row will have NULL in the joined watch_history columns, so filtering for those NULLs isolates never-watched shows. An INNER JOIN would only return shows that already have at least one watch record, a CROSS JOIN produces every possible show-watch combination without filtering for absence, and combining the two tables with UNION ALL does not test for a missing relationship at all.

Q10

Single choice

Two tables, `logins_day1` and `logins_day2`, share the same structure and contain some overlapping user IDs from users who logged in on both days. To build a list of unique users who logged in across the two days, with no duplicate entries, which set operation should be used?

A the UNION operation

✓ Correct

B the UNION ALL operation

☐ C the INTERSECT operation

☐ D the EXCEPT operation

EXPLANATION

UNION combines rows from both tables and removes duplicates, producing exactly the distinct set of users who logged in on either day. UNION ALL would keep the overlapping users twice, INTERSECT would return only the users common to both days, and EXCEPT would return only users present in one table but missing from the other.

Q21

Multiple select

A retailer wants to generate every possible combination of t-shirt type and t-shirt size to pre-populate a size chart. Which statements about the SQL operation suited for this task are correct? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

☒ A A CROSS JOIN produces this result, since it returns the Cartesian product of both tables ✓ Correct

☒ B A CROSS JOIN typically produces a large number of rows and should be used deliberately, not as a default choice ✓ Correct

☐ C An INNER JOIN with no ON condition specified is the standard, well-documented way to generate this result

☒ D The number of output rows equals the number of types multiplied by the number of sizes ✓ Correct

EXPLANATION

A CROSS JOIN returns the Cartesian product of both tables, so its row count is naturally the number of types multiplied by the number of sizes, and because that output can grow very large it should be used deliberately rather than by default. Omitting the ON condition from an INNER JOIN is not the standard way to build a Cartesian product -- CROSS JOIN is the construct built for exactly this purpose.

Q30

Multiple select

A team wants to combine cricket_players (student_id, name, joined_date) and music_club_members (name, student_id, active_status) using a set operation such as UNION. Both tables have 3 columns, but the column order differs and the third column differs in both name and data type. Which statements about this situation are correct? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

- A** The tables must first be adjusted so column names, order, and data types align before UNION, INTERSECT, or EXCEPT can be safely applied ✓ **Correct**
- B** The fact that both tables have exactly 3 columns guarantees the set operation will work correctly, regardless of column names or order
- C** The third columns (joined_date vs active_status) likely have different data types, which the set operation requires to be compatible before combining ✓ **Correct**
- D** The mismatched column order (student_id first vs name first) will combine unrelated values from the same position, even though column counts match ✓ **Correct**

EXPLANATION

Set operations require matching column count, identical column names in the same order, and compatible data types; here the order differs (student_id first vs name first) and the third columns differ in both name and likely data type, so simply matching the column count is not enough -- the tables must be aligned first. It is not enough to just have the same number of columns for the operation to be guaranteed correct.

TOPIC 3 Filtering, Sorting & Basic Queries

3 Qs

Q5

Single choice

A query runs `SELECT name, age FROM students ORDER BY age;` on a SQLite database, and some students have a NULL age value. Where will these students appear in the ascending output?

- A** Excluded from the result entirely, since ORDER BY cannot process NULL values
- B** At the very end of the result, since NULL is treated as the largest value in ascending order
- C** Randomly scattered throughout the result depending on the database engine
- D** At the very beginning of the result, since NULL is treated as the smallest value ✓ **Correct**

EXPLANATION

In SQLite (as in MySQL, PostgreSQL, and SQL Server), NULL values are treated as the smallest possible value in ascending order, so rows with a NULL age sort to the very beginning rather than the end -- they are neither scattered randomly nor dropped from the output. (Note: not every SQL engine agrees -- Oracle sorts NULLs last in ascending order by default.)

Q6

Single choice

A restaurant analyst wants to list every city that appears in a restaurants table, with each city name shown only once even though many restaurants share the same city. Which SQL keyword accomplishes this?

A the BETWEEN clause

B the DISTINCT clause

✓ Correct

C the LIMIT clause

D the ORDER BY clause

EXPLANATION

DISTINCT removes duplicate values so each unique city appears only once. BETWEEN filters values within a numeric range, LIMIT restricts the number of rows returned (without removing duplicates), and ORDER BY only changes the sort order.

Q9

Single choice

A student writes `SELECT price * quantity AS total_cost FROM orders WHERE total_cost > 1000;` expecting it to filter orders where the computed total exceeds 1000, but the database reports that `total_cost` is an unknown column in the WHERE clause. What causes this, and how should the query be fixed?

A SQL does not support computed columns at all, so `price * quantity` must be removed entirely

B WHERE runs before SELECT, so the alias isn't defined yet; use the raw expression directly

✓ Correct

C The AS keyword is invalid syntax in SQL and must be replaced with an equals sign

D WHERE clauses can only filter on primary key columns, never on computed values

EXPLANATION

SQL's execution order runs FROM and WHERE before SELECT, so any alias defined in SELECT (like `total_cost`) does not exist yet when WHERE is evaluated -- the fix is to repeat the full expression directly inside WHERE. SQL does support computed columns via expressions in SELECT, AS is valid alias syntax, and WHERE can filter on any expression, not only primary key columns.

TOPIC 4 CTEs & Query Structure

2 Qs

Q15

Single choice

Consider this query:

```
WITH regional_sales AS (  
  SELECT region, SUM(amount) AS total  
  FROM sales  
  GROUP BY region  
)  
top_regions AS (  
  SELECT region, total  
  FROM regional_sales  
  WHERE total > 100000  
)  
SELECT region FROM top_regions ORDER BY total DESC;
```

What does this query ultimately return?

- ☐ A The single region with the maximum total sales across the entire table, ignoring the 100000 threshold
- ☐ B Every region's raw sales rows from the base table, without any aggregation applied
- ☐ C A syntax error, since a CTE is not allowed to reference another CTE in this query
- ☒ D The names of regions whose total sales exceed 100000, ordered from highest to lowest total sales ✓ Correct

EXPLANATION

The first CTE aggregates sales by region, the second CTE filters that aggregated result down to regions above 100000, and the final SELECT orders those filtered region names by total sales descending -- returning multiple qualifying regions, not raw unaggregated rows, not a syntax error (chaining CTEs where one builds on another is valid), and not only a single top region.

Q18

Single choice

A query needs to (1) filter transactions from the last quarter, (2) aggregate them by region, and (3) rank regions by total revenue -- three sequential steps a teammate will later need to review and modify. Which SQL construct best fits this multi-step, readable requirement?

- ☐ A A single WHERE clause with all three conditions combined using AND, run as one flat filter

B A Common Table Expression (CTE) chain, where each step builds on the previous result

✓ **Correct**

C A CROSS JOIN between the transactions table and itself, producing a Cartesian product

D A DISTINCT clause applied after the final SELECT, with no grouping performed

EXPLANATION

Chaining CTEs lets each sequential step (filtering, then aggregating, then ranking) be written as its own named, readable stage that the next stage builds on, which is exactly the multi-step maintainability CTEs are used for. A single WHERE clause cannot perform aggregation or ranking, a self CROSS JOIN produces a Cartesian product unrelated to sequential filtering and ranking, and a trailing DISTINCT only removes duplicate rows without organizing any of the three steps.

TOPIC 5 Normalization

4 Qs

Q11

Single choice

A campus canteen stores every order in one giant spreadsheet with columns for student name, student phone, dish name, dish price, and order date -- repeating each student's phone number and each dish's price on every order row. When a dish's price changes, the owner must edit hundreds of rows, and typos have already produced two different phone numbers for the same student. Which redesign resolves both problems at their root?

A Split into separate Students, Dishes, and Orders tables linked by keys, storing each fact once

✓ **Correct**

B Keep the single table but add a notes column to record known typos for manual review later

C Add a second identical spreadsheet as a backup so mistakes in one can be cross-checked against the other

D Sort the spreadsheet alphabetically by student name so duplicate phone numbers are easier to spot visually

EXPLANATION

Splitting a single overloaded table into separate logical tables linked by primary and foreign keys stores each phone number and each price exactly once, eliminating both the redundant repeated writes that cause update anomalies and the risk of the same fact being recorded inconsistently. A notes column only documents the problem without preventing it, a duplicate spreadsheet doubles the redundancy instead of removing it, and sorting only makes an existing inconsistency easier to notice, not impossible to create.

Q12

Single choice

A `students` table has a column `favorite_subjects` that stores comma-separated text like 'Math, Science, Art' inside a single cell for each student. Which normalization rule does this design violate, and what fixes it?

- ☐ A Third Normal Form; remove the `favorite_subjects` column entirely since subjects cannot be normalized
- ☒ B First Normal Form; break the comma-separated list into separate rows so each cell holds exactly one value ✓ **Correct**
- ☐ C Referential integrity; add a foreign key from `students` to a `subjects` table without changing the cell structure
- ☐ D Second Normal Form; merge `favorite_subjects` into the primary key to form a composite key

EXPLANATION

First Normal Form requires every cell to hold exactly one atomic value; a comma-separated list violates this, and the fix is to break the list into separate rows. Removing the column loses information rather than fixing the structure, adding a foreign key alone does not make the existing cell atomic, and merging the column into the primary key does not address non-atomic values at all.

Q14

Single choice

A table uses a composite primary key of (`Order_ID`, `Item_Name`), with additional columns `Order_Date` and `Item_Price`. `Order_Date` depends only on `Order_ID`, and `Item_Price` depends only on `Item_Name` -- neither depends on the full composite key. Which normalization issue does this table have, and what fixes it?

- ☐ A A referential integrity violation; add a foreign key from `Order_Date` to `Item_Price`
- ☐ B A First Normal Form violation from non-atomic values; merge `Order_Date` and `Item_Price` into a single column
- ☒ C A Second Normal Form violation; move each column into a table keyed by what it depends on ✓ **Correct**
- ☐ D No issue at all; composite keys automatically satisfy every normal form on their own

EXPLANATION

When a non-key column depends on only part of a composite primary key rather than the whole key, that is a partial dependency and a Second Normal Form violation, fixed by moving each such column into a table keyed by the specific part of the composite key it depends on. There is no non-atomic value here to trigger a First Normal Form issue, no reference between two unrelated attributes to enforce via a foreign key, and composite keys do not automatically guarantee any normal form.

Q27

Multiple select

A students_courses table stores Student_ID, Course_ID, Course_Name, and Department_Head_Name, where Course_Name and Department_Head_Name both depend only on Course_ID, not on the full Student_ID + Course_ID combination and not on Student_ID at all. Which statements about this table's normalization are correct? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

- ☒ **A** The fix is to move Course_Name and Department_Head_Name into a separate Courses table keyed by Course_ID ✓ **Correct**
- ☒ **B** Course_Name and Department_Head_Name depending on Course_ID rather than the full key is a partial dependency, violating 2NF ✓ **Correct**
- ☐ **C** Splitting the data this way increases redundancy compared to keeping everything in one table
- ☐ **D** This table already satisfies Third Normal Form as written, since it has a defined primary key

EXPLANATION

A non-key column depending on only part of a composite primary key (here, Course_ID, which is part of the Student_ID + Course_ID composite key) rather than the whole key is a partial dependency and a Second Normal Form violation -- not a transitive dependency, which requires a non-key column to depend on another non-key column instead of on part of the key. The fix is to move Course_Name and Department_Head_Name into their own Courses table keyed by Course_ID. Having a defined primary key alone does not guarantee 2NF compliance, and separating dependent data into its own table reduces redundancy rather than increasing it.

TOPIC 6 Keys, Constraints & Data Types

4 Qs

Q17

Single choice

A bank wants to guarantee that every customer row in its database always has a recorded balance value, never a missing one. Which SQL constraint enforces this?

- ☐ **A** DEFAULT constraint
- ☐ **B** UNIQUE constraint
- ☒ **C** NOT NULL constraint ✓ **Correct**

D FOREIGN KEY constraint

EXPLANATION

NOT NULL ensures a column can never have a missing value, which is exactly the guarantee needed for a balance column. UNIQUE only prevents duplicate values, DEFAULT only supplies a fallback when no value is given (it doesn't block an explicit NULL), and FOREIGN KEY only links a column to another table's primary key.

Q20

Single choice

In a relational database, which of the following best defines a Primary Key?

- A** A key used exclusively to connect two unrelated databases across servers
- B** A column used only for storing large blocks of text, unrelated to identifying rows
- C** A column that must always contain duplicate values across the table
- D** A column whose value uniquely identifies each record within a table ✓ **Correct**

EXPLANATION

A Primary Key is a column (or set of columns) whose value uniquely identifies each record in a table, similar to how an Aadhaar or passport number uniquely identifies a person. It is not about storing large text, it cannot hold duplicate values, and it is not a mechanism for linking separate, unrelated databases.

Q26

Multiple select

Which of the following are numeric SQL data types used to store numbers with varying size or precision? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

- A** DECIMAL ✓ **Correct**
- B** VARCHAR
- C** BIGINT ✓ **Correct**
- D** INTEGER ✓ **Correct**

EXPLANATION

INTEGER stores whole numbers, DECIMAL stores exact numeric values with decimals, and BIGINT stores integers larger than what a standard integer type can hold -- all three are numeric types. VARCHAR is a non-numeric, variable-length string type used for text, not numbers.

Q29

Multiple select

Which of the following statements correctly describe types of relational integrity constraints? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

- ☐ A Domain integrity guarantees that a foreign key can reference a row that has already been deleted
- ☒ B Entity integrity requires that every table have a primary key whose value is never null ✓ **Correct**
- ☒ C Domain integrity restricts a column to its appropriate data type and format, e.g., numeric-only for age ✓ **Correct**
- ☒ D Referential integrity requires that every foreign key value match an existing primary key value in the related table ✓ **Correct**

EXPLANATION

Referential integrity requires foreign keys to match an existing primary key, entity integrity requires a non-null primary key on every table, and domain integrity restricts each column to its appropriate data type and format. Allowing a foreign key to point to an already-deleted row is precisely what referential integrity is designed to prevent, not something domain integrity permits.

TOPIC 7 Pandas Operations

6 Qs

Q13

Single choice

A data scientist runs `df['dept_avg_salary'] = df.groupby('department')['salary'].transform('mean')` on a DataFrame with 500 employee rows spread across 5 departments. How many rows will the resulting DataFrame have, and what will the new column contain?

- ☐ A 5 rows total; one single average salary value computed per department overall
- ☒ B 500 rows; each row shows the average salary of its own department, repeated across matching rows ✓ **Correct**

- ☒ C 500 rows; each row shows the overall average salary across the entire DataFrame, ignoring department
- ☐ D 2,500 rows; one row produced for every department-employee combination pair

EXPLANATION

`transform()` creates a new column without reducing the row count, so all 500 rows remain, and each row's new column holds the average salary of the department that row belongs to, not a single collapsed table, not an overall dataset-wide average, and not an expanded cross-product of departments and employees.

Q16

Single choice

A pandas DataFrame `employees` needs to be filtered to rows where salary is above 80000 AND department is 'Sales'. A student writes: `employees[employees['salary'] > 80000 and employees['department'] == 'Sales']`. What is the most likely outcome, and what should be corrected?

- ☒ A It raises an error; use ``&`` with parentheses instead of ``and`` ✓ Correct
- ☐ B It works correctly and returns only the Sales employees earning above 80000
- ☐ C It silently returns the full, unfiltered DataFrame with no rows removed
- ☐ D It returns only the department column instead of filtered rows

EXPLANATION

Combining multiple row conditions in pandas requires the `&` operator with each condition wrapped in parentheses, not Python's `and` keyword, because `and` is not defined for element-wise comparison across an entire Series and raises an error rather than silently succeeding, returning everything, or returning only a column.

Q19

Single choice

In pandas, `pd.merge(orders, customers, on='customer_id', how='left')` combines two DataFrames. Which statement correctly describes the behavior of `how='left'`?

- ☐ A Keep every row from both DataFrames regardless of whether the keys match
- ☐ B Keep only rows where `customer_id` exists in both DataFrames
- ☐ C Keep every row from customers, filling unmatched orders columns with NaN

D Keep every row from orders, filling unmatched customers columns with NaN

✓ Correct

EXPLANATION

A left merge keeps every row from the left DataFrame (orders here) and fills in NaN wherever no matching customer_id exists in customers. Keeping only matching rows describes an inner merge, preserving every row of the right table instead describes a right merge, and keeping every row from both regardless of matches describes an outer merge.

Q22

Multiple select

Which of the following statements about combining two DataFrames in pandas are correct? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

A pd.concat() combines two DataFrames vertically, matching SQL's UNION ALL behavior by default

✓ Correct

B Calling drop_duplicates() after pd.concat() reproduces SQL's UNION by removing duplicate rows

✓ Correct

C isin() is used to find elements common to both DataFrames, similar to SQL's INTERSECT

✓ Correct

D pd.merge() is the pandas equivalent of SQL's EXCEPT operation for finding non-matching rows

EXPLANATION

pd.concat() stacks rows from both DataFrames and behaves like UNION ALL by default (keeping duplicates), calling drop_duplicates() afterward removes those duplicates to reproduce UNION, and isin() identifies elements common to both DataFrames the way INTERSECT does. The EXCEPT-style comparison (rows present in one table but not the other) is instead achieved with filtering logic, not with pd.merge().

Q23

Multiple select

A dataset needs a new salary_band column: 'High' for salary above 100000, 'Senior' for 70000-100000, 'Mid' for 40000-69999, and 'Junior' otherwise. Which statements about implementing this with np.select() are correct? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

A np.select() can only evaluate exactly two conditions at a time, so a chained apply or lambda is required for anything beyond that

- B** `np.where()` is a simpler alternative that fits binary, two-outcome conditions better, while `np.select()` fits multiple-condition category creation ✓ **Correct**
- C** A default value can be supplied for rows that don't match any listed condition, covering the 'Junior' case ✓ **Correct**
- D** `np.select()` takes a list of boolean conditions and a matching list of values, applying the value for the first condition that is True for each row ✓ **Correct**

EXPLANATION

`np.select()` evaluates a list of conditions against a matching list of values and applies the first one that is True, and it supports a default value for rows matching none of them -- exactly what handling the 'Junior' fallback needs. It is designed to handle more than two conditions in one call (that is its whole purpose versus the simpler, binary-condition `np.where()`), so it is not limited to only two conditions.

Q24

Multiple select

A dataset has a city column, and an analyst wants to filter for cities that start with 'B' or contain 'pur' anywhere in the name, in a case-insensitive way. Which of the following are valid building blocks for this task? (Select all that apply)

Note: This is a Multiple Select Question (MSQ). The answer is marked incorrect if any correct option is missed or any incorrect option is selected.

- A** `df['city'].str.lower()` used to normalize case before comparison ✓ **Correct**
- B** `df['city'].str.contains('pur', case=False)`, a case-insensitive check ✓ **Correct**
- C** `df['city'].str.startswith('B')`, a simple prefix check ✓ **Correct**
- D** `df['city'].isna()` to check which cities start with 'B'

EXPLANATION

`str.startswith('B')` checks for cities beginning with 'B', `str.contains('pur', case=False)` finds cities containing 'pur' case-insensitively, and `str.lower()` is a valid way to normalize case before comparing strings. `isna()` checks for missing values and has nothing to do with detecting a starting letter.

TOPIC 8 Case Studies (Subjective)

2 Qs

Q31

Subjective

Problem Statement

A college's academic office maintains two tables: `students` (every enrolled student) and `marks` (every exam attempt a student has made). Some students have not yet appeared in `marks` at all -- they have no exam records yet. The academic office wants a single set of SQL queries that gives them a complete, accurate picture of exam performance across the entire student body, including the students who have not taken any exam yet.

Dataset

Run the following in any SQL environment (SQLite, MySQL, or PostgreSQL syntax all work with minor adjustments) to set up the tables:

```
CREATE TABLE students (  
    student_id INTEGER PRIMARY KEY,  
    name TEXT NOT NULL,  
    city TEXT NOT NULL  
);  
  
INSERT INTO students VALUES (1, 'Priya', 'Delhi');  
INSERT INTO students VALUES (2, 'Rahul', 'Mumbai');  
INSERT INTO students VALUES (3, 'Anjali', 'Delhi');  
INSERT INTO students VALUES (4, 'Ketan', 'Pune');  
INSERT INTO students VALUES (5, 'Vamsi', 'Mumbai');  
INSERT INTO students VALUES (6, 'Meera', 'Delhi');  
  
CREATE TABLE marks (  
    marks_id INTEGER PRIMARY KEY,  
    student_id INTEGER,  
    course_id INTEGER,  
    score INTEGER,  
    attendance INTEGER  
);  
  
INSERT INTO marks VALUES (101, 1, 10, 78, 90);  
INSERT INTO marks VALUES (102, 1, 11, 85, 88);  
INSERT INTO marks VALUES (103, 2, 10, 62, 70);  
INSERT INTO marks VALUES (104, 3, 10, 91, 95);  
INSERT INTO marks VALUES (105, 3, 11, 74, 80);  
INSERT INTO marks VALUES (106, 5, 11, 55, 60);
```

Notice that Ketan (`student_id` 4) and Meera (`student_id` 6) have no rows in `marks` at all -- your queries must handle both correctly.

Tasks

1. Write a single SQL query that lists every student's name alongside their exam score, ensuring that students with no marks record still appear in the result (their score should show as NULL rather than the student being dropped).
2. Write a query that computes, for each `course_id`, the total number of exam attempts and the average score, using the appropriate aggregate functions.

3. Using a Common Table Expression (CTE), first build the joined student+marks result from Task 1 as a named CTE, then write a main query on top of that CTE that computes the average score for each city (again, students with no marks should still appear in their city's grouping (their row will have a NULL score, which AVG() automatically skips when computing that city's average)).

Submission Format

Paste all three SQL queries in your response, clearly labeled Task 1, Task 2, and Task 3.

Note: You may use any IDE (online: Google Colab, Jupyter Notebook; offline: VS Code, PyCharm, or similar) to write, run, and test your code. You must **disable all AI-based extensions** (GitHub Copilot, Tabnine, Codeium, etc.) before starting. AI tools of any kind are not permitted and will lead to disqualification. **Submit your complete code as a copy-pasted snippet in the input box — do not submit a URL, notebook link, GitHub repository, or Google Drive file.**

MODEL ANSWER

Question Explanation

This question tests whether a student can combine three separate SQL skills into one coherent workflow: preserving unmatched rows with a LEFT JOIN, summarizing data per group with GROUP BY and aggregate functions, and organizing a multi-step query for readability with a CTE.

Short Explanation

Use a LEFT JOIN with students as the left table so every student appears even without exam records, use GROUP BY with COUNT and AVG for the per-course summary, then wrap the Task 1 join in a CTE and re-aggregate by city on top of it.

Long Explanation

For Task 1, LEFT JOIN students to marks on student_id so every student row is preserved; any student without a matching marks row will show NULL for the score column instead of being excluded. For Task 2, GROUP BY course_id on the marks table and apply COUNT() to count attempts and AVG() to compute the mean score per course. For Task 3, wrap the Task 1 join inside a CTE using WITH ... AS (...), selecting city and score from the joined result; then in the main query, SELECT FROM that CTE, GROUP BY city, and apply AVG(score) -- this keeps the query readable as two clearly separated stages (build the joined base, then aggregate) instead of one large nested query.

Model Solution

```
-- Task 1: every student with their score, including students with no marks
SELECT s.name, m.score
FROM students AS s
LEFT JOIN marks AS m
  ON s.student_id = m.student_id;

-- Task 2: attempts and average score per course
SELECT course_id,
       COUNT(student_id) AS total_attempts,
       AVG(score) AS avg_score
FROM marks
GROUP BY course_id;

-- Task 3: average score per city, using a CTE built on Task 1's join
WITH student_scores AS (
  SELECT s.city, m.score
  FROM students AS s
  LEFT JOIN marks AS m
    ON s.student_id = m.student_id
)
SELECT city, AVG(score) AS avg_city_score
FROM student_scores
GROUP BY city;
```

Q32

Subjective

Problem Statement

A retail chain tracks monthly sales for each of its stores. The data team wants to build feature-engineered columns from this raw monthly sales data to eventually feed a forecasting model. You are asked to reproduce, in pandas, the row-preserving feature engineering that would otherwise be done in SQL with `PARTITION BY` and `LAG`.

Dataset

store_id	month	sales
S1	2026-01	100000
S1	2026-02	110000
S1	2026-03	105000
S1	2026-04	120000
S2	2026-01	80000
S2	2026-02	78000
S2	2026-03	95000
S2	2026-04	90000

Ready-to-run initializer:

```
import pandas as pd

sales_data = pd.DataFrame({
    'store_id': ['S1', 'S1', 'S1', 'S1', 'S2', 'S2', 'S2', 'S2'],
    'month':    ['2026-01', '2026-02', '2026-03', '2026-04', '2026-01', '2026-02', '2026-03', '2026-04'],
    'sales':    [100000, 110000, 105000, 120000, 80000, 78000, 95000, 90000]
})
```

Tasks

1. Add a column `store_avg_sales` containing each store's average sales across all 4 of its months, using `groupby().transform()`, so the row count stays at 8 (no reduction).
2. Add a column `prev_month_sales` containing the previous month's sales for that same store (a missing value for each store's first month), using `shift()` applied within each store's own group so no store's lag value ever comes from a different store.
3. Add a column `pct_change_from_prev` computing the percentage change of `sales` versus `prev_month_sales` for each row, leaving it as a missing value wherever `prev_month_sales` is missing.

Expected Output

The final DataFrame should have 8 rows and the columns: `store_id`, `month`, `sales`, `store_avg_sales`, `prev_month_sales`, `pct_change_from_prev`.

Submission Format

Paste your complete pandas code as a copy-pasted snippet in the input box.

Note: You may use any IDE (online: Google Colab, Jupyter Notebook; offline: VS Code, PyCharm, or similar) to write, run, and test your code. You must **disable all AI-based extensions** (GitHub Copilot, Tabnine, Codeium, etc.) before starting. AI tools of any kind are not permitted and will lead to disqualification. **Submit your complete code as a copy-pasted snippet in the input box — do not submit a URL, notebook link, GitHub repository, or Google Drive file.**

MODEL ANSWER

Question Explanation

This question tests whether a student can chain two pandas feature-engineering techniques -- `groupby().transform()` for a per-group aggregate column, and a grouped `shift()` for a lag feature -- which together reproduce what SQL would express with `OVER/PARTITION BY` and `LAG`.

Short Explanation

Use `groupby('store_id')['sales'].transform('mean')` for the per-store average, `groupby('store_id')['sales'].shift(1)` for the previous month's sales, then compute the percentage change from the shifted column using vectorized arithmetic.

Long Explanation

For Task 1, group by `store_id` and call `.transform('mean')` on the `sales` column; `transform` preserves the original row count by broadcasting the group's average back onto every row of that group, unlike a plain `groupby().mean()` which would collapse to one row per store. For Task 2, group by `store_id` again and call `.shift(1)` on `sales`; because the shift happens after grouping, each store's own previous-month value is retrieved without ever crossing into another store's rows, and the first month of each store naturally becomes

a missing value since there is no prior row to shift from. For Task 3, compute $(\text{sales} - \text{prev_month_sales}) / \text{prev_month_sales}$ for each row using ordinary vectorized pandas arithmetic; wherever `prev_month_sales` is a missing value, the arithmetic naturally propagates a missing result rather than raising an error.

Model Solution

```
import pandas as pd

sales_data = pd.DataFrame({
    'store_id': ['S1', 'S1', 'S1', 'S1', 'S2', 'S2', 'S2', 'S2'],
    'month':    ['2026-01', '2026-02', '2026-03', '2026-04', '2026-01', '2026-02', '2026-03', '2026-04'],
    'sales':    [100000, 110000, 105000, 120000, 80000, 78000, 95000, 90000]
})

# Task 1: per-store average sales, row count preserved
sales_data['store_avg_sales'] = sales_data.groupby('store_id')['sales'].transform('mean')

# Task 2: previous month's sales, computed within each store's own group
sales_data['prev_month_sales'] = sales_data.groupby('store_id')['sales'].shift(1)

# Task 3: percentage change vs previous month
sales_data['pct_change_from_prev'] = (
    (sales_data['sales'] - sales_data['prev_month_sales']) /
    sales_data['prev_month_sales']
) * 100

print(sales_data)
```

SQL, Databases & Pandas - Evaluation (Attempt 2) · Answer Key · CodeWithMunnaX