From Questions to Queries: An AI-powered Multi-Agent Framework for Spatial Text-to-SQL

Ali Khosravi Kazazi[1], Zhenlong Li[1], M. Naser Lessani[1], Guido Cervone[2]

[1]Geoinformation and Big Data Research Laboratory, Department of Geography, The Pennsylvania State University, University Park, PA, USA

[2]Institute for Computational and Data Sciences and Department of Geography, The Pennsylvania State University, University Park, PA, USA

Appendix I

Sample prompts and responses of the five agents within the Spatial Text-to-SQL Pipeline

Sample input natural language question: ***Group counties by state and calculate the average perimeter in kilometers.***

Entity Extraction Agent

Prompt

Role:

You are a Query Understanding & Information Retrieval Expert that only outputs valid JSON. You specialize in analyzing user questions to extract meaningful information for retrieval systems.

Your Goal:

Identify key retrieval points from a user's question, including the underlying intent, important entities, and relevant operations.

Your Task:

Intent Detection (information retrieval focus): Determine what the user is truly asking for. Summarize it in a concise sentence.

Keyword Extraction (linguistic and spatial focus): Identify important keywords or phrases from the question. Treat multi-word named entities and geographic references as single keywords.

Keyword Classification: Classify extracted keywords into:

Entities: tangible or named items such as places, datasets, or thematic subjects.

Operations: words or phrases describing spatial, analytical, or logical relationships.

Input Context:

You are given a user question:

Group counties by state and calculate the average perimeter in kilometers.

Constraints:

Do not provide explanations, reasoning, or any text outside the required JSON structure.

Collaboration Protocol:

Provide structured JSON that can be used directly by downstream agents or systems.

Output Format:

{

"intent": "<brief description of the user's underlying goal>",

"entities": ["<entity1>", "<entity2>", "..."],

"operations": ["<operation1>", "<operation2>", "..."]

}

Evaluation Criteria:

The intent accurately reflects the user's underlying question.

All relevant keywords are correctly extracted.

Keywords are correctly classified into entities and operations.

Reasoning Mode:

Think step-by-step, combining linguistic and spatial analysis. Treat multi-word named entities and geographic references as single phrases.

Sample output of Entity Extraction Agent


```
{'intent': 'Group counties by state and perform calculations',

 'entities': ['counties', 'state'],

 'operations': ['group', 'calculate', 'average']}
```


Metadata Retrieval Agent

Prompt

Role:

You are the Metadata Retrieval Agent, responsible for bridging natural language queries and database schemas. You take extracted entities and keywords from the Entity Extraction Agent and identify the most relevant database tables and columns for downstream query generation.

Your Goal:

Given a set of query entities, select and rank database columns and tables by semantic similarity to the user's question, and generate an enriched, structured mapping that can be used by the Query Logic Agent and SQL Generation Agent.

Your Task:

Candidate Column Selection using the tool that you have access to called "Similarity_Check". The tool computes similarity scores between each query entity and all candidate columns using cosine distance and detect natural breaks in the similarity distribution to determine high possibility candidate columns.

Avoid over-inclusion (irrelevant columns) and under-inclusion (missing relevant columns).

Grouping and Deduplication:

Group selected columns by their respective tables using "to_markdown" function.

Remove duplicate columns:

Produce a structured mapping of each entity to its corresponding tables and columns.

Enrichment:

For each candidate column, get its human-readable descriptions generated by an LLM using "column_abvr_json". Also, include representative sample values from stored metadata in the Knowledge Base using "column_values_json"

Specialized Function Lookup:

When abstract operations require database functions (e.g., spatial predicates in PostGIS), retrieve the most relevant functions and examples from the Knowledge Base using "Supportive_doucments"

Output Subset Schema:

Produce a "trimmed" schema subset containing only relevant tables and columns, to reduce query complexity for the SQL Generation Agent.

Input Context:

Query entities and keywords extracted by the Entity Extraction Agent.

Constraints:

Only output structured JSON or data structures compatible with downstream agents.

Avoid including irrelevant tables, columns, or functions.

Ensure the mapping is clear, deduplicated, and enriched for interpretability.

Collaboration Protocol:

Provide structured outputs consumable by the Query Logic Agent and SQL Generation Agent.

Evaluation Criteria:

Columns and tables accurately match the semantic meaning of the query entities.

Mapping is deduplicated, structured, and enriched with human-readable descriptions and sample values.

Relevant database functions are correctly retrieved for abstract operations.

Reasoning Mode:

Use reasoning to produce contextually appropriate outputs.

Treat all candidate columns, tables, and functions in relation to the query entities for maximal relevance.

Sample output of Metadata Retrieval Agent

Potential table columns

```
{"counties": {
        "state": "State to which the county belongs",
        "geom": "Geometric data describing the county's location"
    }
"states": {
        "gid": "Unique identifier for each state entry",
        "geom": "Geometric data describing the state's location"
}
```

AI-generated descriptions

```
{"counties": {
     "state": "State of the county, stored as a numeric string code   (e.g., '04', '24', '29', '37', '50'). VARCHAR(80), nullable, with 56 distinct non-null codes; the specific coding scheme is not specified.",
     "geom": "County boundary geometry as a MULTIPOLYGON in SRID 4326 (WGS 84). Nullable if the county’s geometry is unknown or not provided."
    }
"states": {
     "gid": "gid is a non-null INTEGER surrogate identifier auto-incremented by the states_gid_seq sequence; all 56 values are unique (e.g., 53, 8, 17). It is described as a unique identifier for the county.",
      "geom": "Boundary geometry stored as a WGS 84 MULTIPOLYGON (geometry(MULTIPOLYGON, 4326)); the column is nullable."
    }
}
```

Sample values

Table counties

```
| state | geom |
|---|---|
| 48 | 0105000020E61 ... |
| 19 | 0105000020E61 ... |
| 13 | 0105000020E61 ... |
```

Table states

```
geom | gid |
|---|---|
0105000020E61 ... | 19 |
0105000020E61 ... | 53 |
0105000020E61 ... | 45 |
```

Spatial functions

```
{"ST_LengthSpheroid": "Returns the 2D or 3D length/perimeter of a lon/lat geometry on a spheroid."
"ST_Perimeter": "Returns the length of the boundary of a polygonal geometry or geography. "
"ST_Perimeter2D": "Returns the 2D perimeter of a polygonal geometry. Alias for ST_Perimeter."
"ST_3DPerimeter": "Returns the 3D perimeter of a polygonal geometry."}
```

Query Logic Agent

Prompt

Role:

You are an expert in SQL and data modeling. Your role is to analyze natural-language questions and table schemas, and produce precise query logic plans.

Goal:

Your ultimate objective is to generate a comprehensive, step-by-step logic plan that clearly specifies how to answer the user's question using the provided tables, columns, and sample data.

Task:

Your immediate task is to examine the question and the table schemas, determine which columns are relevant, how tables should be joined, what filters are required, and how the final output should be structured. You must produce a structured plan that downstream agents or humans could directly use to write SQL queries.

Input Context:

You will be provided with:

A natural-language question ({question})

One or more table schemas with column names and sample rows ({smaple_values_str})

Constraints:

Do not modify the table schemas.

Classify all columns explicitly as join key, filter criterion, output field, or unused.

Columns representing outcomes that the question depends on must be classified as filter criterion.

Follow the required format exactly; do not add extra commentary.

Collaboration Protocol:

Receive the natural-language question and table schemas as input. Your output will be consumed by SQL generation or query validation agents, so structure it for clarity and completeness.

Output Format:

Your answer must follow these sections in order:

<u>Tables & Columns:</u>

Column Name

Role: join key / filter criterion / output field / unused

Explanation if unused, or reasoning for role assignment

If filter criterion, specify the exact logical filter

If join key, specify the table.column it joins to and why

<u>Join Strategy:</u>

Left table.column = Right table.column

Join type (INNER, LEFT, etc.)

Justification for the join type

Any null-handling or deduplication considerations

Filter Conditions:

Status flags, outcome-based filters, date ranges, question-specific conditions

Output Definition:

Include aliases where appropriate

Explain purpose of each column

Specify aggregates and group-by keys if applicable

High-Level Algorithm – Step-by-step plan in pseudocode or plain English:

Order of table joins

When filters are applied

Aggregation/grouping steps

Final projection and ordering

Evaluation Criteria:

Success is measured by completeness, correctness, and clarity. The plan must include all relevant tables and columns, specify all joins and filters, and allow direct translation into SQL.

Reasoning Mode:

Use analytical, step-by-step logical reasoning to identify column roles, join strategy, filters, output structure, and data processing flow. Always justify your choices clearly in the structured plan.

Question: Group counties by state and calculate the average perimeter in kilometers

Table Schemas:

```
Table counties

| state | geom |
|---|---|
```

48	0105000020E61 ...
19	0105000020E61 ...
13	0105000020E61 ...

Table states

geom	gid
0105000020E61 ...	19
0105000020E61 ...	53
0105000020E61 ...	45

Sample output of Query Logic Agent

Abstract problem

Spatial relationship: Containment — counties are contained within states.
Spatial operation: Aggregation based on containment.
Attribute of interest: Perimeter — a geometric property derived from shape.
Unit conversion: Conversion of perimeter to kilometers (spatial measurement issue)

Query Logical plan

Tables & Columns

Table: counties

- state

Role: join key
Joins to states.gid to associate each county with its state.

- geom

Role: output field
Used to calculate the perimeter of each county using spatial functions.

Table: states

- geom

Role: unused
Not required for this query since perimeter is computed per county; state geometry is not needed.

- gid

Role: join key
Joins to counties.state to group counties by state.

Join Strategy

```
counties.state = states.gid
Join type: INNER JOIN
Reason: Only counties that belong to a valid state are needed; ensures all selected counties have matching states. No need for outer joins as we are only aggregating counties that exist.
```

Filter Conditions

```
No additional filters required for this question. All counties are included in the calculation.
```

Output Definition

```
states.gid AS state_id — identifies the state for grouping.
AVG(ST_Perimeter(counties.geom)::numeric / 1000) AS avg_perimeter_km — computes the average perimeter of counties in kilometers.
ST_Perimeter calculates the perimeter in the unit of the spatial reference system; dividing by 1000 converts meters to kilometers.
```

High-Level Algorithm

```
1. Join counties with states on counties.state = states.gid using an INNER JOIN.
2. For each county, compute the perimeter using ST_Perimeter(counties.geom).
3. Group the results by states.gid.
4. Calculate the average perimeter for each state.
5. Return the state identifier and average perimeter in kilometers.
```

SQL Generation Agent

Prompt

Role:

You are the SQL Generation Agent, the code-producing component of a multi-agent system. Your role is to synthesize executable SQL statements based on the logical plans provided by the Query Logic Agent, together with schema descriptions and sample data from the Metadata Retrieval Agent.

Goal:

Your ultimate objective is to produce correct, fully executable SQL queries that conform to the syntax and constraints of the primary database while capturing the user's intent as expressed in the logical plan.

Task:

Your immediate task is to consume the logical plan and table schemas, request trimmed table views with sample values when needed, select appropriate casting, handle nulls safely, apply parameterization for literals, resolve column ambiguities using aliases, and construct join clauses with the correct keys. You also decide the most appropriate predicate shapes for filtering, such as exact matches or ranges, based on sample values and column types.

Input Context:

You will be provided with the logical plan (including tables, columns, join strategies, filters, and output definitions), trimmed schema subsets with sample values from the Metadata Retrieval Agent, and any relevant metadata or documentation for spatial or non-spatial functions.

Constraints:

Do not perform reasoning about query logic or column relevance; rely entirely on the provided plan. Ensure SQL adheres to database syntax and semantic correctness. Avoid generating queries that could produce ambiguous results due to unclear aliases or missing joins.

Collaboration Protocol:

Receive logical plans from the Query Logic Agent and schema samples from the Metadata Retrieval Agent. Provide your SQL statement and a structured manifest to the Reviewer Agent for validation and provenance tracking.

Output Format:

Return a complete SQL statement along with a manifest in JSON format. The manifest must enumerate the columns returned, predicates applied, expected coordinate reference system or units for spatial queries, and any potential issues or assumptions such as ambiguous column names or approximate spatial metrics.

Evaluation Criteria:

Success is measured by the correctness and executability of the SQL, adherence to the logical plan, clarity of join and filter clauses, safe handling of nulls and data types, and completeness of the manifest for review and provenance tracking.

Reasoning Mode:

Use analytical and detail-oriented reasoning to translate a validated logical plan into syntactically correct SQL, applying database best practices for joins, filtering, aggregation, and spatial computations, while recording all assumptions and potential issues in the manifest.

Sample output of SQL Generation Agent

```
SQL Query

SELECT
    c.state,
```

```sql
    AVG(ST_Perimeter(c.geom::geography)) / 1000.0 AS avg_perimeter_km
FROM counties AS c
GROUP BY c.state
ORDER BY c.state;
```

Manifest


```json
{
  "columns_returned": [
    {
      "name": "state",
      "source": "counties.state",
      "description": "Identifier of the state used for grouping counties"
    },
    {
      "name": "avg_perimeter_km",
      "source": "counties.geom",
      "description": "Average perimeter of counties in kilometers"
    }
  ],
  "predicates_applied": [],
  "spatial_info": {
    "geometry_column": "counties.geom",
    "operation": "ST_Perimeter cast to geography",
    "unit_assumption": "meters from ST_Perimeter, converted to kilometers"
  },
  "assumptions": [
    "All counties have valid geometries.",
```

```
    "Perimeter computation uses geography casting for correct measurement in meters.",
    "No additional filters applied; all counties are included.",
    "Grouping is done by state identifier."
  ],
  "potential_issues": [
    "No explicit join to the states table; assumes state ID alone is sufficient for grouping.",
    "Null geometries would cause errors; assumes geom is always valid."
  ]
}
```

Review Agent

Prompt

Role:

You are the Review Agent, a self-verifying, LLM-assisted quality control module responsible for validating and improving SQL queries before they are returned to the end user. You serve as the final checkpoint to ensure semantic correctness, schema alignment, and safe execution.

Goal:

Your ultimate objective is to confirm that the generated SQL query accurately answers the user's natural-language question, respects the database schema, and produces meaningful, representative results. You must detect missing columns, incorrect joins, mismatched filters, or any spatial/metric inconsistencies, and correct them as needed.

Task:

Your immediate task is to evaluate a generated SQL statement by performing multiple programmatic checks. First, use the LogicChecker tool to determine whether the SQL matches the intent of the natural-language query, returning a JSON verdict with "ok": true or "ok": false" and a "reason" for the assessment.

Next, execute a sandboxed dry run using the QueryExecution function with a LIMIT 10 clause to produce a representative sample of rows. Review this sample for completeness and correctness using your LLM-assisted evaluation. If the output is missing required columns, invoke the AddColumn function to inject missing columns into the SELECT clause and revise the SQL accordingly.

Additionally, use QueryInfo to generate a machine-readable manifest of tables, columns, joins, and predicates, and apply DoubleCheck to validate that all tables and columns exist, literal values match their declared types, and spatial operations (CRS, units, ST functions) are correct.

Input Context:

You receive the natural-language question, the generated SQL statement, a schema description, and optionally a small sample of SQL output rows from a sandboxed execution. You also have access to programmatic tools including LogicChecker, QueryExecution, AddColumn, QueryInfo, and DoubleCheck.

Constraints:

Do not modify the original query arbitrarily; only apply changes to correct semantic or structural issues identified by your checks. Ensure the reviewed SQL remains syntactically valid, safe to execute, and consistent with the user's intent. Avoid returning results without a thorough validation process.

If the query includes spatial functions, call "spatial_ guide" which is a text file to support how to check spatial constraints.

Collaboration Protocol:

Receive the generated SQL and its manifest from the SQL Generation Agent. Use the LogicChecker, QueryExecution, QueryInfo, DoubleCheck, and AddColumn tools as needed. After validation and any corrections, return the approved SQL query and updated manifest to the end user or the system pipeline.

Output Format:

Return a JSON object containing the reviewed SQL statement under "SQL" and a rationale to show why the query is a good answer for the given question.

Evaluation Criteria:

Success is measured by whether the SQL statement correctly answers the user query, conforms to the database schema, handles spatial and non-spatial data appropriately, and passes all internal programmatic checks. The manifest should be complete and accurately reflect the query's tables, columns, predicates, joins, and spatial considerations.

Reasoning Mode:

Use a combination of deterministic programmatic checks and LLM-assisted reasoning to systematically verify SQL semantics, structure, and outputs. Identify issues, justify any corrections, and ensure the reviewed SQL fully aligns with both the natural-language question and schema constraints before returning it.

Sample output of Review Agent

```
Validated SQL Query
{
"SQL":
"SELECT
    c.state,
```

```
    AVG(ST_Perimeter(c.geom::geography)) / 1000.0 AS avg_perimeter_km
FROM counties AS c
GROUP BY c.state
ORDER BY c.state",

"Rationale": "This query accurately groups counties by state, calculates the correct geodesic perimeter of each county,
converts it to kilometers, and averages the values per state. Its structure and use of spatial functions ensure both semantic
correctness and real-world accuracy."
}
```

Appendix II

Spatial Guide that Supports the Review Agent

This context guides the Reviewer Agent validating and reasoning about PostGIS spatial queries, including function selection, CRS handling, geometry vs geography, units, and topology.

1. Coordinate Reference Systems (CRS)

Agent instructions:

- Always check the SRID of input geometries.
- Ensure consistent SRIDs between geometries when performing spatial operations.
- When SRIDs differ, recommend using ST_Transform:
  For example: ST_Intersects(a.geom, ST_Transform(b.geom, ST_SRID(a.geom)));
- CRS determines units (degrees, meters, feet). Units affect distance, area, buffer results.
- Advise the user to use projected CRS for metric calculations.

2. Geometry vs Geography

Agent instructions:

- Use geometry for local/topological operations or when custom CRS is required.
- Use geography for global distances or cross-continental queries.
- Example guidelines:

Use Case	Use Geometry	Use Geography
Global / cross-continental	Not Recommended	Recommended
Topological operations	Recommended	Use With Caution
Accurate Earth distances	Use With Caution	Recommended
Custom CRS	Recommended	Not Supported

- Verify function compatibility with type (geometry vs geography).

3. Function Semantics

Agent instructions:

- Categorize spatial functions and reason about expected input/output:

Category	Examples	Expected Output
Measurement	ST_Area, ST_Length, ST_Distance	Numeric (units depend on CRS/type)
Relationship tests	ST_Intersects, ST_Within, ST_Touches, ST_Contains	Boolean
Constructive	ST_Buffer, ST_Union, ST_Intersection, ST_Difference	Geometry
Transformative	ST_Transform, ST_Simplify, ST_SnapToGrid	Geometry
Accessors	ST_X, ST_Y, ST_NPoints	Numeric
Aggregation	ST_Union, ST_Collect	Geometry

- Always verify input type (geometry/geography) and expected output type.

4. Units and Measurements

Agent instructions:

- Geometry in EPSG:4326 → distances in degrees.
- Recommend ST_Transform to projected CRS for metric calculations:
- For example: SELECT ST_Area(ST_Transform(geom, 3857)) / 1000000 AS area_km2;
- Geography type → always returns meters for distance and area.

5. Precision and Validity

Agent instructions:

- If invalid, suggest fixing with:
  For example: SELECT ST_MakeValid(geom);
- Warn about distortions from simplification or transformation:
  For example: SELECT ST_SimplifyPreserveTopology(geom, tolerance);

6. Dimensionality

Agent instructions:

- Default functions are 2D; check if 3D functions are required:
  For example: ST_3DDistance, ST_3DIntersects
- Consider Z (elevation) and M (measure) only if analysis requires it.

7. Topology and Spatial Relationships

Agent instructions:

Understand and distinguish relationships for logical reasoning:

Function	Meaning
ST_Within(a, b)	a lies completely inside b
ST_Contains(a, b)	b lies inside a
ST_Touches(a, b)	Boundaries touch but do not overlap
ST_Intersects(a, b)	Any overlap
ST_Crosses(a, b)	Share some but not all points

Correctly interpret natural language queries in terms of relationships.

8. Combining Spatial and Non-Spatial Logic

Agent instructions:

- Spatial queries may include attribute filtering:

```
SELECT city.name
FROM city
JOIN country ON ST_Within(city.geom, country.geom)
WHERE country.name = 'France'
  AND city.population > 100000;
```

- Recognize that spatial relationship + attribute condition = combined logic.

9. Geometry Validity Across Functions

Agent instructions:

- Check for topology or SRID changes after functions like ST_Union.
- Advise explicit casting when mixing geometry and geography:

    For example: ST_Distance(geom::geography, other_geom::geography);

Appendix III
Evaluation Results of Spatial Queries (SpatialQueryBenchmark)

Level 1: Basic

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation
Find all points of interest (POIs) classified as a 'bench'.	SELECT name, geom FROM poi WHERE fclass = 'bench';	SELECT osm_id AS osm_id, name AS name FROM poi WHERE fclass = 'bench';	✅	Both queries correctly filter the POIs by fclass = 'bench', which satisfies the question's intent.	SELECT osm_id AS osm_id, name AS name FROM poi WHERE fclass = 'bench';	Reviewer didn't make any changes All three correctly filter POIs by fclass = 'bench'. The question doesn't specify required columns. All satisfy the intent.	✅
What is the area of the county named 'Winn' in square kilometers?	SELECT ST_Area(geom::geography) / 1000000 AS area_sq_km FROM counties WHERE name = 'Winn';	SELECT ST_Area(geom::geography) / 1e6 AS area_sq_km FROM counties WHERE name = 'Winn';	✅	Both queries correctly cast the geometry to geography, use ST_Area to get square meters, and divide by 1,000,000 to return square kilometers for the county named 'Winn'.	SELECT ST_Area(geom::geography) / 1e6 AS area_sq_km FROM counties WHERE name = 'Winn';	Reviewer didn't make any changes. All three correctly compute the county's area by using ST_Area on the geography cast (yielding square meters) and dividing by 1e6 (or 1000000) to return square kilometers, filtered by name = 'Winn'.	✅

List the names of all roads with a maximum speed limit greater than 50 mph.	SELECT name FROM roads WHERE maxspeed > 50;	SELECT DISTINCT
		  name AS road_name
		FROM
		  roads
		WHERE
		  maxspeed > 50
		  AND name IS NOT NULL
		  AND name <> '';
			✅	The System-Generated Query correctly returns road names for records with maxspeed > 50, excludes null/empty names, and uses DISTINCT to avoid duplicate names—aligned with “List the names of all roads,” which implies valid, unique names.
				The Benchmark-Proposed Query may include null/empty names and duplicates because it lacks filters and DISTINCT, which does not cleanly satisfy the request to list the road names.
					SELECT DISTINCT
					  name AS road_name
					FROM
					  roads
					WHERE
					  maxspeed > 50
					  AND name IS NOT NULL
					  AND name <> '';
						All correctly filter roads with maxspeed > 50 and return the road names, excluding NULL/empty names and removing duplicates, which aligns with "List the names" and avoids unnamed entries or repeated names.	✅

Show the WGS 84 geometry for the state of 'Nevada'.	SELECT geom FROM states WHERE name = 'Nevada';	SELECT states.geom AS geomFROM statesWHERE states.name = 'Nevada';	✅	The System-Generated Query is a valid SQL statement that returns the geometry for Nevada. Assuming the states.geom column is stored in WGS 84 (common for such datasets), it correctly answers the question.	SELECT states.geom AS geomFROM statesWHERE states.name = 'Nevada';	All three are equivalent: they return the geom for the row where name = 'Nevada'. If states.geom is stored in WGS 84, they correctly answer the question; B and C just alias the column.Reviewer didn't make any changes.	✅

What is the length of the road segment with the gid of 360036, in meters?	SELECT ST_Length(geom::geography) AS length_meters FROM roads WHERE gid = 360036;	SELECT ST_Length(geom::geography) AS length_m FROM roads WHERE gid = 360036;	✅	Both queries correctly compute the length in meters by casting the geometry to geography and using ST_Length, filtered by gid = 360036. The alias difference (length_m vs length_meters) does not affect correctness.	SELECT ST_Length(geom::geography) AS length_m FROM roads WHERE gid = 360036;	All three queries correctly cast geom to geography and use ST_Length to return length in meters, filtered by gid = 360036. Alias differences (length_meters vs length_m) do not affect correctness. None of the queries are incorrect; they are functionally identical.	✅
Find the location of the POI named 'Staples'.	SELECT geom FROM poi WHERE name = 'Staples';	SELECT geom AS location FROM poi WHERE name = 'Staples';	✅	The System-Generated Query is correct because it selects the geometry column (location) of POIs where the name is 'Staples', and it clearly aliases the result as "location," matching the question's intent.	SELECT geom AS location FROM poi WHERE name = 'Staples';	All correctly filter poi by name = 'Staples' and return the geom (the location). Using an alias as location in B and C does not change correctness. Reviewer didn't make any changes.	✅

What is the area of the protected area called 'Everglades' in square kilometers?	SELECT ST_Area(geom::geography) / 1000000 AS area_sq_km FROM ne_protected_areas WHERE name = 'Everglades';	SELECT ST_Area(ST_Transform(geom, 6933)) / 1000000.0 AS area_sq_kmFROM ne_protected_areasWHERE unit_name = 'Everglades';	❌	The Benchmark-Proposed Query correctly computes area on the spheroid using geography (square meters) and converts to square kilometers, and it filters by the likely name field ("name = 'Everglades'"), matching the question's phrasing.The System-Generated Query may return no result due to filtering on "unit_name" (likely not the correct column for this dataset) and relies on a specific projected CRS (EPSG:6933), which may be inappropriate or unnecessary, potentially introducing error.	SELECT SUM(ST_Area(geom::geography)) / 1000000.0 AS area_sq_kmFROM ne_protected_areasWHERE unit_name ILIKE '%Everglades%';	Reviewer correctly returns a single area in square kilometers by summing all matching geometries, uses geodesic area via geography, and matches likely real names (e.g., "Everglades National Park") with a case-insensitive partial match.Unreviewed query is incorrect because it filters on unit_name = 'Everglades' (likely too strict), and it also doesn't aggregate, potentially returning multiple rows instead of a single total area.	✅
List all weather stations located in the state of 'MS'.	SELECT name, geom FROM ghcn WHERE state = 'MS';	SELECT g.station_id AS station_id, g.name AS station_name, g.lat AS latitude, g.lon AS longitude, g.state AS state FROM ghcn AS g INNER JOIN states AS s ON g.state = s.stusps WHERE s.stusps = 'MS';	✅	The System-Generated Query correctly filters ghcn stations to state 'MS' (via join on states.stusps) and selects appropriate station fields (station_id, name, lat, lon, state).	SELECT g.station_id AS station_id, g.name AS station_name, g.lat AS latitude, g.lon AS longitude, g.state AS state FROM ghcn AS g INNER JOIN states AS s ON g.state = s.stusps WHERE s.stusps = 'MS';	All correctly list stations with state = 'MS'. Reviewer didn't make any changes. None are incorrect.	✅

What are the coordinates (longitude, latitude) of the weather station with the ID 'US1NCHR0026'?	SELECT ST_AsText(geom) FROM ghcn WHERE station_id = 'US1NCHR0026';	SELECT ghcn.lon AS longitude, ghcn.lat AS latitude FROM ghcn WHERE ghcn.station_id = 'US1NCHR0026';	✅	The System-Generated Query is correct because it explicitly returns two numeric columns labeled longitude and latitude, matching the requested format for the station ID.	SELECT DISTINCT lon AS longitude, lat AS latitude FROM ghcn WHERE station_id = 'US1NCHR0026' LIMIT 1;	All correctly return the longitude and latitude as separate, appropriately named columns filtered by the station_id. Reviewer additionally uses DISTINCT and LIMIT 1, which is acceptable.	✅
Find all roads that are marked as tunnels.	SELECT name FROM roads WHERE tunnel = 'T';	SELECT gid AS road_id, name AS road_name, fclass AS road_class, tunnel AS tunnel_flagFROM roadsWHERE tunnel = 'T';	✅	The System-Generated Query correctly filters roads where tunnel = 'T' and returns valid road records; it is a valid SQL statement that identifies roads marked as tunnels.	SELECT gid AS road_id, name AS road_name, fclass AS road_class, tunnel AS tunnel_flagFROM roadsWHERE tunnel = 'T';	Reviewer didn't make any changes. All three queries correctly filter roads where tunnel = 'T'.	✅
What is the area of the census tract with geoid '27027020700' in square meters?	SELECT ST_Area(geom::geography) AS area_sq_m FROM tracts WHERE geoid = '27027020700';	SELECT geoid, ST_Area(geom::geography) AS area_sqm FROM tracts WHERE geoid = '27027020700';	✅	Both queries correctly compute the area in square meters by casting geom to geography and filtering by the specified geoid. The System-Generated Query returns the geoid alongside the area; The Benchmark-Proposed Query returns just the area. Both satisfy the question.	SELECT geoid, ST_Area(geom::geography) AS area_sqm FROM tracts WHERE geoid = '27027020700';	All correctly compute area in square meters by casting geom to geography and filtering geoid = '27027020700'. Benchmark query returns just the area, which directly answers the question; System generated queries also include the geoid but still correctly answer it. System generated queries are identical to each other and equally correct.	✅

List the names of all protected areas that are a 'National Park'.	SELECT unit_name FROM ne_protected_areas WHERE unit_type = 'National Park';	SELECT ne_protected_areas.name AS protected_area_name FROM ne_protected_areas WHERE ne_protected_areas.unit_type = 'National Park';	✅	The System-Generated Query is correct because it filters rows where unit_type = 'National Park' and returns the protected area names from the name column, matching the question.	SELECT ne_protected_areas.name AS protected_area_name FROM ne_protected_areas WHERE ne_protected_areas.unit_type = 'National Park';	Reviewer didn't make any changes. System generated queries are identical and correctly return the protected area names (ne_protected_areas.name) filtered by unit_type = 'National Park'.	✅
Find the geometric centroid of the state with the USPS code 'DC', returned as a WGS 84 point.	SELECT ST_Centroid(geom) FROM states WHERE stusps = 'DC';	SELECT ST_AsText(ST_Centroid(ST_Transform(geom, 4326))) AS centroid_wgs84_wktFROM statesWHERE stusps = 'DC';	✅	The System-Generated Query is correct because it returns the centroid in WGS 84 (EPSG:4326). It transforms the geometry to 4326 and then computes the centroid, and outputs it as WKT. While best practice is typically to compute the centroid in the native/projected CRS and then transform, the question only requires the result to be returned as a WGS 84 point, which The System-Generated Query satisfies.	SELECT ST_AsText(ST_Centroid(ST_Transform(geom, 4326))) AS centroid_wgs84_wktFROM statesWHERE stusps = 'DC';	Reviewer didn't make any changesSystem generated queries correctly return the centroid as a WGS 84 (EPSG:4326) point by transforming to 4326 and outputting WKT.	✅

What is the geometry type for the features in the roads table?	SELECT DISTINCT ST_GeometryType(geom) FROM roads;	SELECT DISTINCT GeometryType(geom) AS geometry_type FROM roads WHERE geom IS NOT NULL;	✅	The System-Generated Query correctly returns the distinct geometry types of non-null geometries from the roads table, aligning with the intent to report actual feature types. Filtering out NULLs avoids an irrelevant NULL entry.	SELECT DISTINCT GeometryType(geom) AS geometry_type FROM roads WHERE geom IS NOT NULL;	Reviewer didn't make any changes. System generated queries correctly return the distinct geometry types for non-null features, aligning with the question.	✅
Retrieve the name of the county with the geoid '27117'.	SELECT name FROM counties WHERE geoid = '27117';	SELECT counties.name AS county_name FROM counties WHERE counties.geoid = '27117';	✅	The System-Generated Query correctly selects the county name from the counties table where geoid equals '27117', returning the requested name (optionally aliased as county_name).	SELECT counties.name AS county_name FROM counties WHERE counties.geoid = '27117';	Reviewer didn't make any changes	✅
What is the longitude of the POI with gid 9918?	SELECT ST_X(geom) FROM poi WHERE gid = 9918;	SELECT ST_X(poi.geom) AS longitude FROM poi WHERE poi.gid = 9918;	✅	The System-Generated Query correctly selects the X coordinate (longitude) from the geometry for the POI with gid 9918 and is valid SQL.	SELECT ST_X(poi.geom) AS longitude FROM poi WHERE poi.gid = 9918;	All three correctly select the X coordinate (longitude) of the geometry for gid = 9918 using ST_X and the same WHERE filter. Reviewer didn't make any changes.	✅

What is the latitude of the POI with gid 2240?	SELECT ST_Y(geom) FROM poi WHERE gid = 2240;	SELECT ST_Y(geom) AS latitudeFROM poiWHERE gid = 2240;	✅	The System-Generated Query is correct because it uses ST_Y(geom) to return the latitude for the POI with gid = 2240 and appropriately aliases the result as latitude.	SELECT ST_Y(geom) AS latitudeFROM poiWHERE gid = 2240;	Reviewer didn't make any changes. All are correct.	✅
Find the WGS 84 geometry for the road named 'Veronica Road'.	SELECT geom FROM roads WHERE name = 'Veronica Road';	SELECT CASE WHEN ST_SRID(geom) = 4326 THEN geom ELSE ST_Transform(geom, 4326) END AS geom_wgs84, name AS road_name FROM roads WHERE name = 'Veronica Road';	✅	The System-Generated Query correctly ensures the geometry is in WGS 84 (EPSG:4326) by checking the SRID and transforming when necessary, while filtering for the road named 'Veronica Road'. The Benchmark-Proposed Query may return the geometry in whatever SRID the table stores; it does not guarantee WGS 84, so it may not satisfy the requirement.	SELECT CASE WHEN ST_SRID(geom) = 4326 THEN geom ELSE ST_Transform(geom, 4326) END AS geom_wgs84, name AS road_name FROM roads WHERE name = 'Veronica Road';	Reviewer didn't make any changes. System generated queries correctly return the geometry in WGS 84 (EPSG:4326), transforming if needed, for the road named 'Veronica Road'.	✅
How many POIs are classified as 'convenience' stores?	SELECT COUNT(*) FROM poi WHERE fclass = 'convenience';	SELECT COUNT(DISTINCT p.gid) AS convenience_poi_count FROM poi AS p WHERE p.fclass = 'convenience';	✅	The System-Generated Query is correct because it counts distinct POIs (by gid) with fclass = 'convenience', ensuring one count per POI.	SELECT COUNT(DISTINCT p.gid) AS convenience_poi_count FROM poi AS p WHERE p.fclass = 'convenience';	All three correctly count POIs with fclass = 'convenience'. Benchamrk query counts rows; System generated queries count distinct gid values. SInce gid is a unique, non-null identifier (typical for POI tables), all return the same result.	✅

Find the area of the time zone named '-3' in square kilometers.	SELECT ST_Area(geom::geography) / 1000000 AS area_sq_km FROM ne_time_zones WHERE name = '-3';	SELECT name AS timezone_name, ST_Area(ST_Transform(geom, 6933)) / 1000000.0 AS area_sq_km FROM ne_time_zones WHERE name = '-3';	❌	The System-Generated Query lacks aggregation and would return one row per feature (and includes an extra column), not the single total area requested.	SELECT SUM(ST_Area(ST_Transform(geom, 6933))) / 1000000.0 AS area_sq_km FROM ne_time_zones WHERE name = '-3';	Query C correctly sums the areas of all geometries with name = '-3', transforms to an equal-area meter-based CRS (EPSG:6933), and returns a single area value in square kilometers—matching the intent to get the total area of that time zone. Uneriwed query lacks aggregation and would return one row per feature (and includes an extra column), not the single total area requested.	✅
List all counties located in the state with FIPS code '51'.	SELECT name FROM counties WHERE state = '51';	SELECT name AS county_name FROM counties WHERE state = '51';	✅	The System-Generated Query is correct because it selects county names from the counties table and filters by state = '51', returning the list of counties in the state with FIPS code '51'. The alias county_name is appropriate but not required.	SELECT name AS county_name FROM counties WHERE state = '51';	All three correctly filter counties where state = '51' and return the county name; B and C only add an alias. Reviewer didn't make any changes. No incorrect queries.	✅

Find the WGS 84 geometry of the protected area with the unit_code 'EVER'.	SELECT geom FROM ne_protected_areas WHERE unit_code = 'EVER';	SELECT geom AS geom_wgs84 FROM ne_protected_areas WHERE unit_code = 'EVER';	✅	Both queries correctly return the geometry for the protected area with unit_code 'EVER'. If the geom column is stored in WGS 84 (common for Natural Earth datasets), then both satisfy the request; The System-Generated Query simply aliases the column as geom_wgs84.	SELECT geom AS geom_wgs84 FROM ne_protected_areas WHERE unit_code = 'EVER';	Reviewer didn't make any changes. All three queries return the same geometry for unit_code = 'EVER'. Natural Earth geometries are in WGS 84 (EPSG:4326), so selecting geom (aliased or not) satisfies the request.	✅
Retrieve the WGS 84 geometry of the block group with the geoid '060710017031'.	SELECT geom FROM blockgroups WHERE geoid = '060710017031';	SELECT ST_Transform(geom, 4326) AS geom_wgs84 FROM blockgroups WHERE geoid = '060710017031';	✅	The System-Generated Query correctly returns the geometry in WGS 84 by applying ST_Transform(..., 4326), ensuring the result is in EPSG:4326 regardless of the source SRID.	SELECT ST_Transform(geom, 4326) AS geom_wgs84 FROM blockgroups WHERE geoid = '060710017031';	Reviewer didn't make any changes Correct: System-generated queries transform geom to EPSG:4326 (WGS 84) and filter by the given geoid.	✅

What is the cataloged water area (awater) of the state named 'Guam'? (Note: The units are as-is from the database.)	SELECT awater FROM states WHERE name = 'Guam';	SELECT states.awater AS awater FROM states WHERE states.name = 'Guam';	✅	The System-Generated Query correctly selects the awater value from the states table where name = 'Guam', matching the question. The Benchmark-Proposed Query is wrapped as a JSON-like array containing a string, which is not valid SQL as-is. While the inner SQL text is logically correct, the provided form would not execute as a SQL query.	SELECT states.awater AS awater FROM states WHERE states.name = 'Guam';	Reviewer didn't make any changes. All three queries correctly select the awater value from the states table for the row where name = 'Guam'.	✅
List all roads that are classified as bridges.	SELECT name FROM roads WHERE bridge = 'T';	SELECT roads.gid AS road_id, roads.name AS road_name FROM roads WHERE roads.bridge = 'T';	✅	Both queries correctly filter roads where bridge = 'T', which matches "roads that are classified as bridges." The System-Generated Query returns road_id and road_name; The Benchmark-Proposed Query returns just the name. Either satisfies the request to list the roads.	SELECT roads.gid AS road_id, roads.name AS road_name FROM roads WHERE roads.bridge = 'T';	Reviewer didn't make any changes. All three correctly filter roads where bridge = 'T', i.e., roads classified as bridges. System-Generated quereis are identical (returning road_id and road_name). Benchmark qury returns just the name, which still satisfies "List all roads."	✅

Show the boundary of the county named 'Glasscock' as a WGS 84 geometry.	SELECT ST_Boundary(geom) FROM counties WHERE name = 'Glasscock';	SELECT name AS county_name, ST_AsText(ST_Transform(geom, 4326)) AS wgs84_geom_wkt FROM counties WHERE name = 'Glasscock';	✗	The Benchmark-Proposed Query is correct because it computes the boundary of the county geometry. It would fully meet the requirement if the source geometry is already in WGS 84; otherwise, it needs ST_Transform to 4326. The System-Generated Query is incorrect because it returns the full polygon (not its boundary), even though it transforms to WGS 84.	SELECT name AS county_name, ST_AsText(ST_Transform(geom, 4326)) AS wgs84_geom_wkt FROM counties WHERE name = 'Glasscock';	Benchmark query is the only one that computes the boundary (ST_Boundary). If counties.geom is already stored in WGS 84 (SRID 4326), this satisfies the request. If not, the ideal form would be ST_Boundary(ST_Transform(geom, 4326)). System-generated queries transform to WGS 84 but do not compute the boundary; they return the full geometry as WKT text (not the boundary geometry), so they do not answer the question.	✗
What is the recorded elevation for the weather station named 'Howell Canyon'?	SELECT elev FROM ghcn WHERE name = 'Howell Canyon';	SELECT DISTINCT elev AS elevation FROM ghcn WHERE name = 'HOWELL CANYON' LIMIT 1;	✗	The System-Generated query is incorrect because it filters on 'HOWELL CANYON' (uppercase), which may not match in a case-sensitive collation, and it uses DISTINCT LIMIT 1, which could arbitrarily choose one elevation if duplicates exist.	SELECT g.elev AS elevation FROM ghcn AS g WHERE g.name ILIKE '%Howell%';	Reviewer query is incorrect because it uses a partial match with ILIKE '%Howell%'	✗

Find the WGS 84 geometry for the time zone where 'New Zealand' is listed in the places column.	SELECT geom FROM ne_time_zones WHERE places = 'New Zealand';	SELECT geom AS wgs84_geom FROM ne_10m_time_zones WHERE tz_name = 'New Zealand';	❌	The System-Generated Query is incorrect because it uses a different table (ne_10m_time_zones) and filters by tz_name instead of the places column specified in the question.	SELECT geom AS wgs84_geom FROM ne_time_zones WHERE places = 'New Zealand';	Reviewer uses the correct table (ne_time_zones), filter on the correct column (places = 'New Zealand'), and return the geometry	✅
What is the geometry type for features in the poi table?	SELECT DISTINCT ST_GeometryType(geom) FROM poi;	SELECT DISTINCT ST_GeometryType(geom) AS geometry_type FROM poi;	✅	Both queries correctly use ST_GeometryType(geom) with DISTINCT to return the geometry type(s) present in the poi table, matching the question's intent.	SELECT DISTINCT ST_GeometryType(geom) AS geometry_type FROM poi;	Reviewer didn't make any changes. All are correct.	✅
Find the name of the state with the statefp code '41'.	SELECT name FROM states WHERE statefp = '41';	SELECT states.name AS state_name FROM states WHERE states.statefp = '41';	✅	Both queries correctly filter the states table by statefp = '41' and return the state's name. The System-Generated Query returns the column as state_name; The Benchmark-Proposed Query returns it as name.	SELECT states.name AS state_name FROM states WHERE states.statefp = '41';	All three queries correctly select the state name from states where statefp = '41'.	✅

Level 2: Intermediate

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation
Find all points of interest within the county named 'Winn'.	SELECT p.* FROM poi p JOIN counties c ON ST_Within(p.geom, c.geom) WHERE c.name = 'Winn';	WITH target_county AS (SELECT geom FROM counties WHERE name = 'Winn' AND geom IS NOT NULL) SELECT p.gid AS poi_id, p.name AS poi_name, p.fclass AS poi_class, p.geom AS poi_geom FROM poi AS p INNER JOIN target_county AS c ON ST_Within(p.geom, c.geom);	✅	The system-generated query is correct because ST_Within(p.geom, c.geom) returns POIs strictly inside the county geometry named 'Winn', which matches the intent of "within the county."	WITH target_county AS (SELECT geom FROM counties WHERE name = 'Winn' AND geom IS NOT NULL) SELECT p.gid AS poi_id, p.name AS poi_name, p.fclass AS poi_class, p.geom AS poi_geom FROM poi AS p INNER JOIN target_county AS c ON ST_Within(p.geom, c.geom);	System-generated queries are correct because they use ST_Within(p.geom, c.geom), which strictly returns POIs located inside the county polygon named 'Winn'. They also guard against NULL county geometries.	✅

Which roads are in the block group with geoid '421010336004'?	SELECT r.* FROM roads r JOIN blockgroups b ON ST_Within(r.geom, b.geom) WHERE b.geoid = '421010336004';	WITH target_bg AS (
		  SELECT geom
		  FROM blockgroups
		  WHERE geoid = '421010336004'
		)
		SELECT
		  r.gid AS road_id,
		  COALESCE(r.name, '') AS road_name,
		  r.fclass AS road_class,
		  r.maxspeed AS maxspeed
		FROM roads r
		JOIN target_bg b
		  ON ST_Within(r.geom, b.geom);
			✅	Both queries correctly identify roads whose geometries intersect the block group geometry for geoid '421010336004', which aligns with the typical interpretation of "in" as at least partially within the block group.	WITH target_bg AS (
					  SELECT geom
					  FROM blockgroups
					  WHERE geoid = '421010336004'
					)
					SELECT
					  r.gid AS road_id,
					  COALESCE(r.name, '') AS road_name,
					  r.fclass AS road_class,
					  r.maxspeed AS maxspeed
					FROM roads r
					JOIN target_bg b
					  ON ST_Within(r.geom, b.geom);
						Reviewer didn't make any changes.
						Benchmark and unreviewed queries correctly return roads that lie in the block group by using ST_Intersects with the specified geoid, which matches the usual intent of “in” (any overlap within the polygon).
						Reviewer uses ST_Within, requiring roads to be entirely contained within the block group, excluding roads that cross the boundary; this is too restrictive for the question.
							✅

List the GHCN stations located within the protected area named 'Sleeping Bear Dunes NL'.	SELECT g.* FROM ghcn g JOIN ne_protected_areas pa ON ST_Within(g.geom, pa.geom) WHERE pa.name = 'Sleeping Bear Dunes NL';	SELECT
		  g.station_id AS station_id,
		  g.name AS station_name
		FROM ne_protected_areas AS p
		JOIN ghcn AS g
		  ON ST_Contains(p.geom, g.geom)
		WHERE p.unit_name = 'Sleeping Bear Dunes NL';
			✅	The system-gnerated is correct because it filters the protected area by the expected name field (pa.name = 'Sleeping Bear Dunes NL') and uses a suitable spatial predicate (ST_Intersects) to return stations inside or on the boundary of the area.	SELECT DISTINCT
					  g.station_id,
					  g.name
					FROM ne_protected_areas AS p
					JOIN ghcn AS g
					  ON ST_Contains(p.geom, g.geom)
					WHERE p.unit_name = 'Sleeping Bear Dunes NL';
						Reviewer didn't make any changes.
						System-generated queries correctly implement “located within” by using ST_Contains(p.geom, g.geom) and filter the protected area by unit_name = 'Sleeping Bear Dunes NL'. Query C adds DISTINCT to prevent potential duplicates; both answer the question.
							✅

Find all counties that intersect the state named 'Guam'.	SELECT c.* FROM counties c JOIN states s ON ST_Intersects(c.geom, s.geom) WHERE s.name = 'Guam';	SELECT
		  c.geoid AS county_geoid,
		  c.name AS county_name
		FROM counties AS c
		INNER JOIN states AS s
		  ON ST_Intersects(c.geom, s.geom)
		WHERE s.name = 'Guam';
			✅	Both queries correctly join counties to states using ST_Intersects and filter for the state named 'Guam', returning all counties whose geometries intersect Guam.	SELECT
					  c.geoid AS county_geoid,
					  c.name AS county_name
					FROM counties AS c
					INNER JOIN states AS s
					  ON ST_Intersects(c.geom, s.geom)
					WHERE s.name = 'Guam';
						All three correctly join counties to states using ST_Intersects and filter by s.name = 'Guam', which directly answers the question.
						The only difference is selected columns (c.* vs specific fields); the question doesn't constrain the output schema, so all are valid. Reviewer didn't make any changes.
							✅

Which points of interest are within 10 kilometers of the GHCN station with station_id 'USS0005J27S'?	SELECT p.* FROM poi p JOIN ghcn g ON ST_DWithin(p.geom::geography, g.geom::geography, 10000) WHERE g.station_id = 'USS0005J27S';	WITH target_station AS (
		  SELECT geom::geography AS geom
		  FROM ghcn
		  WHERE station_id = 'USS0005J27S'
		  LIMIT 1
		)
		SELECT
		  p.gid AS poi_id,
		  p.name AS poi_name,
		  p.fclass AS poi_class,
		  ST_Distance(ts.geom, p.geom::geography) AS distance_meters
		FROM target_station ts
		JOIN poi p
		  ON ST_DWithin(ts.geom, p.geom::geography, 10000)
		ORDER BY distance_meters ASC;
			✅	The system-generated query correctly casts both geometries to geography, filters POIs within 10,000 meters of the specified station, and returns a clear POI list with distances, ordered by proximity.	WITH target_station AS (
					  SELECT geom::geography AS geom
					  FROM ghcn
					  WHERE station_id = 'USS0005J27S'
					  LIMIT 1
					)
					SELECT
					  p.gid AS poi_id,
					  p.name AS poi_name,
					  p.fclass AS poi_class,
					  ST_Distance(ts.geom, p.geom::geography) AS distance_meters
					FROM target_station ts
					JOIN poi p
					  ON ST_DWithin(ts.geom, p.geom::geography, 10000)
					ORDER BY distance_meters ASC;
						Reviewer didn't make any changes.
						All three correctly use ST_DWithin with geography and a 10,000-meter radius to return POIs near the specified station. Benchamrk returns all POI columns; system-generated queries (identical) return selected fields plus distance and order by proximity.
							✅

List the roads that intersect the tract with geoid '27027020700'.	SELECT r.* FROM roads r JOIN tracts t ON ST_Intersects(r.geom, t.geom) WHERE t.geoid = '27027020700';	WITH target_tract AS (
		  SELECT geom
		  FROM tracts
		  WHERE geoid = '27027020700'
		)
		SELECT DISTINCT
		  r.gid AS road_id,
		  r.name AS road_name,
		  r.fclass AS road_class
		FROM roads r
		JOIN target_tract t
		  ON ST_Intersects(t.geom, r.geom);
			✅	Both queries correctly identify roads whose geometries intersect the geometry of the tract with geoid '27027020700' using ST_Intersects and filtering on that geoid.	WITH target_tract AS (
					  SELECT geom
					  FROM tracts
					  WHERE geoid = '27027020700'
					)
					SELECT DISTINCT
					  r.gid AS road_id,
					  r.name AS road_name,
					  r.fclass AS road_class
					FROM roads r
					JOIN target_tract t
					  ON ST_Intersects(t.geom, r.geom);
						Reviewer didn't make any changes.
						All three correctly return roads whose geometries intersect the tract geometry with geoid '27027020700'.
							✅

Find all block groups that overlap with the time zone where zone = -7.0.	SELECT b.* FROM blockgroups b JOIN ne_time_zones tz ON ST_Intersects(b.geom, tz.geom) WHERE tz.zone = -7.0;	WITH tz AS (
		  SELECT geom
		  FROM ne_time_zones
		  WHERE zone = -7.0
		),
		bg AS (
		  SELECT geoid, geom
		  FROM blockgroups
		  WHERE geom IS NOT NULL
		)
		SELECT DISTINCT b.geoid
		FROM bg b
		JOIN tz t
		  ON ST_Intersects(b.geom, t.geom);
			✅	The system-generated query correctly returns the set of block groups that intersect the -7.0 time zone and deduplicates them (DISTINCT geoid), aligning with “find all block groups” as a unique set.	WITH tz AS (
					  SELECT geom
					  FROM ne_time_zones
					  WHERE zone = '-7.0'
					),
					bg AS (
					  SELECT geoid, geom
					  FROM blockgroups
					  WHERE geom IS NOT NULL
					)
					SELECT DISTINCT b.geoid
					FROM bg b
					JOIN tz t
					  ON ST_Intersects(b.geom, t.geom);
						reviewed queriy is valid since return the same result as when filter using WHERE zone=-7.0. The postgresql database engine treats both queries identically after the implicit cast, they produce the exact same result.	✅

Which protected areas intersect the county with geoid '48109'?	SELECT pa.* FROM ne_protected_areas pa JOIN counties c ON ST_Intersects(pa.geom, c.geom) WHERE c.geoid = '48109';	WITH target_county AS (SELECT geom FROM counties WHERE geoid = '48109' AND geom IS NOT NULL) SELECT p.gid AS protected_area_id, p.unit_name AS protected_area_name, p.unit_type AS protected_area_type FROM target_county c JOIN ne_protected_areas p ON p.geom IS NOT NULL AND ST_Intersects(c.geom, p.geom);	✅	Both queries correctly identify protected areas whose geometries intersect the geometry of the county with geoid '48109'. They use ST_Intersects between ne_protected_areas.geom and the target county geometry filtered by geoid.	WITH target_county AS (SELECT geom FROM counties WHERE geoid = '48109' AND geom IS NOT NULL) SELECT p.gid AS protected_area_id, p.unit_name AS protected_area_name, p.unit_type AS protected_area_type FROM target_county c JOIN ne_protected_areas p ON p.geom IS NOT NULL AND ST_Intersects(c.geom, p.geom);	All three queries correctly return protected areas whose geometries intersect the county geometry with geoid '48109' using ST_Intersects with the county filtered to that GEOID.	✅

List the points of interest within 5 kilometers of the road named 'Veronica Road'.	SELECT p.* FROM poi p JOIN roads r ON ST_DWithin(p.geom::geography, r.geom::geography, 5000) WHERE r.name = 'Veronica Road';	WITH target_road AS (
		  SELECT geom::geography AS g
		  FROM roads
		  WHERE name = 'Veronica Road'
		)
		SELECT
		  p.gid AS poi_id,
		  p.name AS poi_name,
		  p.fclass AS poi_type,
		
		ST_Distance(p.geom::geography, r.g) AS distance_meters
		FROM poi p
		CROSS JOIN target_road r
		WHERE
		ST_DWithin(p.geom::geography, r.g, 5000)
		ORDER BY distance_meters;
			✅	The system-generated query correctly casts to geography, filters POIs within 5,000 meters of the geometry of roads named "Veronica Road," and returns relevant POI fields (with distance). This answers the question.	WITH target_road AS (
					  SELECT geom::geography AS g
					  FROM roads
					  WHERE name = 'Veronica Road'
					)
					SELECT
					  p.gid AS poi_id,
					  p.name AS poi_name,
					  p.fclass AS poi_type,
					
					ST_Distance(p.geom::geography, r.g) AS distance_meters
					FROM poi p
					CROSS JOIN target_road r
					WHERE
					ST_DWithin(p.geom::geography, r.g, 5000)
					ORDER BY distance_meters;
						All three correctly filter POIs within 5,000 meters of the geometry of roads named 'Veronica Road' using ST_DWithin on geography, which matches the 5 km requirement.
						None are incorrect; all satisfy the prompt.
							✅

Find all GHCN stations that intersect the state with statefp '41'.	SELECT g.* FROM ghcn g JOIN states s ON ST_Intersects(g.geom, s.geom) WHERE s.statefp = '41';	SELECT
		  g.station_id AS station_id,
		  g.name AS station_name
		FROM ghcn AS g
		JOIN states AS s
		  ON ST_Intersects(g.geom, s.geom)
		WHERE s.statefp = '41';
			✅	Both queries correctly join ghcn to states using ST_Intersects and filter by s.statefp = '41', returning the GHCN stations that intersect that state.	SELECT
					  g.station_id AS station_id,
					  g.name AS station_name
					FROM ghcn AS g
					JOIN states AS s
					  ON ST_Intersects(g.geom, s.geom)
					WHERE s.statefp = '41';
						Reviewer didn't make any changes.
						All three correctly join ghcn to states using ST_Intersects and filter s.statefp = '41', returning the stations that intersect that state.
						Benchmark returns all station columns; system-generateded queries (identical) return only station_id and name.
							✅

Which tracts intersect the protected area with unit_code 'MOJA'?	SELECT t.* FROM tracts t JOIN ne_protected_areas pa ON ST_Intersects(t.geom, pa.geom) WHERE pa.unit_code = 'MOJA';	SELECT DISTINCT
		  t.geoid AS tract_geoid
		FROM tracts AS t
		INNER JOIN ne_protected_areas AS p
		  ON ST_Intersects(t.geom, p.geom)
		WHERE p.unit_code = 'MOJA';
			✅	The system-generated query correctly identifies tracts that intersect MOJA by using ST_Intersects and returns a deduplicated list of tract identifiers (geoid), ensuring each tract is listed once even if multiple MOJA polygons intersect it.	SELECT DISTINCT
					  t.geoid AS tract_geoid
					FROM tracts AS t
					INNER JOIN ne_protected_areas AS p
					  ON ST_Intersects(t.geom, p.geom)
					WHERE p.unit_code = 'MOJA';
						Reviewer didn't make any changes
						Correct: System-generated queries return the distinct set of tract identifiers (t.geoid) that intersect protected areas with unit_code = 'MOJA', matching “which tracts” as a unique list.
							✅

List the counties within 20 kilometers of the points of interest with name 'Staples'.	SELECT c.* FROM counties c JOIN poi p ON ST_DWithin(c.geom::geography, p.geom::geography, 20000) WHERE p.name = 'Staples';	SELECT DISTINCT c.name AS county_name
		FROM counties AS c
		CROSS JOIN LATERAL (
		  SELECT geom
		  FROM poi
		  WHERE name = 'Staples'
		) AS p
		WHERE c.geom IS NOT NULL
		  AND p.geom IS NOT NULL
		  AND
		ST_DWithin(c.geom::geography, p.geom::geography, 20000);
			✅	System-generated query correctly finds all counties whose geometries are within 20,000 meters of any POI named 'Staples' using ST_DWithin on geography and filters by name. It returns the counties as requested.	SELECT DISTINCT c.name AS county_name
					FROM counties AS c
					CROSS JOIN LATERAL (
					  SELECT geom
					  FROM poi
					  WHERE name = 'Staples'
					) AS p
					WHERE c.geom IS NOT NULL
					  AND p.geom IS NOT NULL
					  AND
					ST_DWithin(c.geom::geography, p.geom::geography, 20000);
						Reviewer doesn't make any changes to the unreviwed query .	✅

Find all roads that are within a 1 kilometer buffer of the block group with gid 11548.	SELECT r.* FROM roads r JOIN blockgroups b ON ST_Within(r.geom, ST_Buffer(b.geom::geography, 1000)::geometry) WHERE b.gid = 11548;	SELECT r.gid AS road_gid, r.name AS road_name, r.fclasses AS road_class, r.geom AS road_geom FROM roads r JOIN (SELECT ST_Buffer(bg.geom::geography, 1000)::geometry AS buf FROM blockgroups bg WHERE bg.gid = 11548) b ON ST_Within(r.geom, b.buf);	❌	The system-gnerated query uses a wrong column name 'fclasses' that must be 'fclass' instead.	SELECT r.gid AS road_gid, r.name AS road_name, r.fclass AS road_class, r.geom AS road_geom FROM roads r JOIN (SELECT ST_Buffer(bg.geom::geography, 1000)::geometry AS buf FROM blockgroups bg WHERE bg.gid = 11548) b ON ST_Within(r.geom, b.buf);	The query construct a 1 km buffer around the block group geometry using ST_Buffer(geom::geography, 1000) and then find roads that intersect that buffer via ST_Intersects, which captures roads within (or touching) the buffer area.	✅

Which time zones intersect the state named 'Nevada'?	SELECT tz.* FROM ne_time_zones tz JOIN states s ON ST_Intersects(tz.geom, s.geom) WHERE s.name = 'Nevada';	SELECT DISTINCT tz.name AS timezone_name, tz.zone AS timezone_offset_numeric, tz.utc_format AS timezone_utc_label, tz.iso_8601 AS timezone_iso8601_example FROM states s INNER JOIN ne_time_zones tz ON ST_Intersects(s.geom, tz.geom) WHERE s.name = 'Nevada' AND tz.featurecla = 'Timezone';	✅	The system-generated query is correct because it joins time zones to the Nevada state geometry using ST_Intersects, filters for the Nevada record, restricts to timezone features (featurecla = 'Timezone'), and uses DISTINCT to avoid duplicates—returning a clean list of intersecting time zones.	SELECT DISTINCT tz.name AS timezone_name, tz.zone AS timezone_offset_numeric, tz.utc_format AS timezone_utc_label, tz.iso_8601 AS timezone_iso8601_example FROM states s INNER JOIN ne_time_zones tz ON ST_Intersects(s.geom, tz.geom) WHERE s.name = 'Nevada' AND tz.featurecla = 'Timezone';	Reviwed query is identical to unreviwed query and is correct for the same reason.	✅
List the points of interest that intersect the county in state '51' named 'Glasscock'.	SELECT p.* FROM poi p JOIN counties c ON ST_Intersects(p.geom, c.geom) WHERE c.state = '51' AND c.name = 'Glasscock';	SELECT p.gid AS poi_id, p.name AS poi_name FROM poi AS p INNER JOIN counties AS c ON ST_Intersects(p.geom, c.geom) WHERE c.state = '50' AND c.name = 'Glasscock';	❌	System-generated query is incorrect because it filters state = '50' instead of '51'.	SELECT p.gid AS poi_id, p.name AS poi_name FROM poi AS p INNER JOIN counties AS c ON ST_Intersects(p.geom, c.geom) WHERE c.state = '51' AND c.name = 'Glasscock';	Reviwer correctly join POIs to counties using ST_Intersects and filter the county by state = '51' and name = 'Glasscock', listing the intersecting POIs (Benchmark returns all POI columns; Reviwer returns id and name).	✅

Find all GHCN stations within 15 kilometers of the protected area with nps_region 'Intermountain'.	SELECT g.* FROM ghcn g JOIN ne_protected_areas pa ON ST_DWithin(g.geom::geography, pa.geom::geography, 15000) WHERE pa.nps_region = 'Intermountain';	WITH intermountain_areas AS (SELECT geom FROM ne_protected_areas WHERE nps_region = 'Intermountain' AND geom IS NOT NULL) SELECT DISTINCT g.station_id AS station_id, g.name AS station_name, g.state AS state FROM ghcn AS g JOIN intermountain_areas AS a ON ST_Boundary(g.geom::geography, a.geom::geography, 15000);	❌	The system-generated query is incorrect because it uses ST_Boundary instead of ST_DWithin.	WITH intermountain_areas AS (SELECT geom FROM ne_protected_areas WHERE nps_region = 'Intermountain' AND geom IS NOT NULL) SELECT DISTINCT g.station_id AS station_id, g.name AS station_name, g.state AS state FROM ghcn AS g JOIN intermountain_areas AS a ON ST_DWithin(g.geom::geography, a.geom::geography, 15000);	Reviewer correctly filter protected areas to nps_region = 'Intermountain' and use ST_DWithin with geography and a 15,000-meter threshold to return GHCN stations within 15 km. Reviwer adds DISTINCT and selects specific columns, but answer the question.	✅
Which roads intersect the time zone with tz_name1st 'America'?	SELECT r.* FROM roads r JOIN ne_time_zones tz ON ST_Intersects(r.geom, tz.geom) WHERE tz.tz_name1st = 'America';	SELECT DISTINCT r.name AS road_name FROM roads AS r JOIN ne_time_zones AS tz ON ST_Intersects(r.geom, tz.geom) WHERE tz.tz_name1st LIKE 'America%';	✅	The system-generated query is correct because it matches all time zones whose tz_name1st begins with 'America'.	SELECT DISTINCT r.name AS road_name FROM roads AS r JOIN ne_time_zones AS tz ON ST_Intersects(r.geom, tz.geom) WHERE tz.tz_name1st = 'America';	All queries are correct and return the same output.	✅

List the tracts that overlap with the county named 'Conecuh'.	SELECT t.* FROM tracts t JOIN counties c ON ST_Intersects(t.geom, c.geom) WHERE c.name = 'Conecuh';	SELECT DISTINCT t.geoid AS tract_geoid FROM tracts AS t INNER JOIN counties AS c ON ST_Intersects(t.geom, c.geom) WHERE c.name = 'Conecuh';	✅	Both queries correctly identify tracts whose geometries intersect the geometry of the county named 'Conecuh' using ST_Intersects, which is appropriate for "overlap" in this context (including full containment).	SELECT DISTINCT t.geoid AS tract_geoid FROM tracts AS t INNER JOIN counties AS c ON ST_Intersects(t.geom, c.geom) WHERE c.name = 'Conecuh';	All three correctly join tracts to the county via ST_Intersects and filter on c.name = 'Conecuh', returning the tracts that overlap the county.	✅
Find all points of interest within a 2 kilometer buffer of the GHCN station named 'LAKE PLACID 4.8 NW'.	SELECT p.* FROM poi p JOIN ghcn g ON ST_Intersects(p.geom, ST_Buffer(g.geom::geography, 2000)::geometry) WHERE g.name = 'LAKE PLACID 4.8 NW';	WITH station AS (SELECT geom FROM ghcn WHERE name = 'LAKE PLACID 4.8 NW' LIMIT 1) SELECT p.gid AS gid, p.name AS poi_name, p.fclass AS fclass, p.code AS code, p.osm_id AS osm_id FROM poi p CROSS JOIN station s WHERE ST_DWithin(p.geom::geography, s.geom::geography, 2000);	✅	The system-generated query is correct because it uses ST_DWithin on geography with a 2000-meter threshold, which accurately returns POIs within 2 km of the specified station point.	WITH station AS (SELECT geom FROM ghcn WHERE name = 'LAKE PLACID 4.8 NW' LIMIT 1) SELECT p.gid AS gid, p.name AS poi_name, p.fclass AS fclass, p.code AS code, p.osm_id AS osm_id FROM poi p CROSS JOIN station s WHERE ST_DWithin(p.geom::geography, s.geom::geography, 2000);	All three correctly return POIs within 2 km of the station. Benchamrk query uses a geodesic 2 km buffer on the station (geography) and intersects POIs (geometry), which is topologically equivalent to a 2 km distance filter. Sustem-generated queris use ST_DWithin on geography with a 2000 m threshold, which directly implements the requirement.	✅

Which block groups intersect the state with geoid '47'?	SELECT b.* FROM blockgroups b JOIN states s ON ST_Intersects(b.geom, s.geom) WHERE s.geoid = '47';	SELECT DISTINCT b.geoid AS blockgroup_geoid FROM blockgroups AS b INNER JOIN states AS s ON ST_Intersects(b.geom, s.geom) WHERE s.geoid = '47';	✅	The system-generated query is correct because it returns the unique GEOIDs of block groups whose geometries intersect the state geometry with geoid '47', directly answering "which block groups" by identifier.	SELECT DISTINCT b.geoid AS blockgroup_geoid FROM blockgroups AS b INNER JOIN states AS s ON ST_Intersects(b.geom, s.geom) WHERE s.geoid = '47';	All three correctly join blockgroups to the state via ST_Intersects and filter on s.geoid = '47'. Benchmark query returns full block group rows. Reviewer didn't make any changes. None are incorrect.	✅
List the protected areas within 50 kilometers of the point of interest with fclass 'bench'.	SELECT pa.* FROM ne_protected_areas pa JOIN poi p ON ST_DWithin(pa.geom::geography, p.geom::geography, 50000) WHERE p.fclass = 'bench';	SELECT DISTINCT npa.gid AS protected_area_id, npa.name AS protected_area_name FROM poi p JOIN ne_protected_areas npa ON ST_Intersects(p.geom::geography, npa.geom::geography, 50000) WHERE p.fclass = 'bench';	❌	The system-generated query is incorrect because it returns protected areas that intersects the buffer while question directly asks for 'within'.	SELECT DISTINCT npa.gid AS protected_area_id, npa.name AS protected_area_name FROM poi p JOIN ne_protected_areas npa ON ST_Within(p.geom::geography, npa.geom::geography, 50000) WHERE p.fclass = 'bench';	Reviewer incorrectly uses ST_Within fucntion instead of ST_DWithin	❌

Find all roads that intersect the tract with gid 21463.	SELECT r.* FROM roads r JOIN tracts t ON ST_Intersects(r.geom, t.geom) WHERE t.gid = 21463;	WITH target_tract AS (
		  SELECT geom
		  FROM tracts
		  WHERE gid = 21463
		)
		SELECT
		  r.gid AS road_gid,
		  r.name AS road_name,
		  r.fclass AS road_class,
		  r.maxspeed AS road_maxspeed
		FROM roads r
		JOIN target_tract t
		  ON ST_Intersects(r.geom, t.geom);
			✅	Both queries correctly identify roads whose geometries intersect the geometry of the tract with gid 21463 using ST_Intersects, returning the set of intersecting roads.	WITH target_tract AS (
					  SELECT geom
					  FROM tracts
					  WHERE gid = 21463
					)
					SELECT
					  r.gid AS road_gid,
					  r.name AS road_name,
					  r.fclass AS road_class,
					  r.maxspeed AS road_maxspeed
					FROM roads r
					JOIN target_tract t
					  ON ST_Intersects(r.geom, t.geom);
						All three correctly join roads to the tract geometry and use ST_Intersects to return roads that intersect the tract with gid = 21463.	✅

Which GHCN stations are within 10 kilometers of the time zone with name '+14'?	SELECT g.* FROM ghcn g JOIN ne_time_zones tz ON ST_DWithin(g.geom::geography, tz.geom::geography, 10000) WHERE tz.name = '+14';	WITH tz AS (
		  SELECT geom
		  FROM ne_time_zones
		  WHERE name = '+14'
		)
		SELECT g.station_id AS station_id,
		       g.name AS station_name
		FROM ghcn AS g
		JOIN tz
		  ON ST_DWithin(
		       ST_Transform(g.geom, 3857),
		       ST_Transform(tz.geom, 3857),
		       10000
		     );
			✕	The Benchmark-Proposed Query correctly measures 10 kilometers using geodesic distance by casting both geometries to geography and applying ST_DWithin with a 10,000-meter threshold. It also filters the time zone by name = '+14', satisfying the intent precisely.
				The system-generated query measures distance after transforming to EPSG:3857 (Web Mercator), which introduces scale distortion and makes the 10 km threshold inaccurate, especially away from the equator. While it will run, it does not reliably answer a geodesic “within 10 km” question.
					WITH tz AS (
					  SELECT geom
					  FROM ne_time_zones
					  WHERE name = '+14'
					)
					SELECT g.station_id AS station_id,
					       g.name AS station_name
					FROM ghcn AS g
					JOIN tz
					  ON ST_DWithin(
					       ST_Transform(g.geom, 3857),
					       ST_Transform(tz.geom, 3857),
					       10000
					     );
						Benchamrk query is correct because it measures distance using geography (meters) via ST_DWithin(...::geography, ...::geography, 10000) and properly filters the time zone by name = '+14'. This yields GHCN stations within 10 km geodesically of the '+14' time zone polygon, including those inside it (distance 0). Reviewed is incorrect because it computes distance on geometry in EPSG:3857, which introduces planar distortion and can be especially wrong near the dateline where the '+14' zone is located. This can misclassify stations relative to the 10 km threshold.	✕

List the counties that overlap with the protected area named 'Everglades'.	SELECT c.* FROM counties c JOIN ne_protected_areas pa ON ST_Intersects(c.geom, pa.geom) WHERE pa.name = 'Everglades';	WITH everglades AS (
		  SELECT geom
		  FROM ne_protected_areas
		  WHERE name = 'Everglades'
		)
		SELECT DISTINCT
		  c.geoid AS county_geoid,
		  c.name AS county_name,
		  c.state AS state_fips
		FROM counties c
		INNER JOIN everglades e
		  ON ST_Intersects(c.geom, e.geom);
			✅	The system-generated query correctly returns a de-duplicated list of counties whose geometries intersect the geometry(ies) of the protected area named 'Everglades'. Using DISTINCT prevents duplicate counties when the protected area has multiple geometry records.	WITH everglades AS (
					  SELECT geom
					  FROM ne_protected_areas
					  WHERE name = 'Everglades'
					)
					SELECT DISTINCT
					  c.geoid AS county_geoid,
					  c.name AS county_name,
					  c.state AS state_fips
					FROM counties c
					INNER JOIN everglades e
					  ON ST_Intersects(c.geom, e.geom);
						Reviewer didn't make any changes.
						System-generated queries correctly return the counties that intersect the Everglades geometry and use DISTINCT to avoid duplicates when multiple Everglades geometries exist; they present a clean county list.
							✅

Find all points of interest that intersect the block group in county '053'.	SELECT p.* FROM poi p JOIN blockgroups b ON ST_Intersects(p.geom, b.geom) WHERE b.county = '053';	SELECT
		  p.gid AS poi_id,
		  p.name AS poi_name,
		  p.fclass AS poi_class,
		  p.geom AS poi_geom,
		  bg.county AS county,
		  bg.geom AS blockgroup_geom
		FROM poi AS p
		INNER JOIN blockgroups AS bg
		  ON ST_Intersects(p.geom, bg.geom)
		WHERE bg.county = '053';
			✅	Both queries correctly join points of interest to block groups using ST_Intersects and filter to county = '053', returning POIs that intersect block groups in that county.	SELECT
					  p.gid AS poi_id,
					  p.name AS poi_name,
					  p.fclass AS poi_class,
					  p.geom AS poi_geom,
					  bg.county AS county,
					  bg.geom AS blockgroup_geom
					FROM poi AS p
					INNER JOIN blockgroups AS bg
					  ON ST_Intersects(p.geom, bg.geom)
					WHERE bg.county = '053';
						Reviewer didn't make any changes. All three correctly join POIs to blockgroups using ST_Intersects and filter to county '053', thereby returning POIs that intersect block groups in that county. Differences are only in selected columns, not logic. No query is incorrect.	✅

Which states intersect the time zone with zone -3.0?	SELECT s.* FROM states s JOIN ne_time_zones tz ON ST_Intersects(s.geom, tz.geom) WHERE tz.zone = -3.0;	WITH target_zone AS (
		  SELECT geom
		  FROM ne_time_zones
		  WHERE zone = -3.0
		)
		SELECT DISTINCT s.name AS state_name
		FROM states s
		INNER JOIN target_zone tz
		  ON ST_Intersects(s.geom, tz.geom);
			✅	The system-generated query is correct because it filters the time zone to zone = -3.0, intersects states with those geometries, and returns a deduplicated list of state names, matching “Which states intersect...”.	WITH target_zone AS (
					  SELECT geom
					  FROM ne_time_zones
					  WHERE zone = -3.0
					)
					SELECT DISTINCT s.name AS state_name
					FROM states s
					INNER JOIN target_zone tz
					  ON ST_Intersects(s.geom, tz.geom);
						System-generated queries correctly return a distinct list of state names that intersect any time zone polygon with zone = -3.0, matching the intent of “Which states.” Benchmark returns all columns from states (not just names)	✅

List the roads within 5 kilometers of the GHCN station with elev 112.0.	SELECT r.* FROM roads r JOIN ghcn g ON ST_DWithin(r.geom::geography, g.geom::geography, 5000) WHERE g.elev = 112.0;	WITH station AS (
		  SELECT geom
		  FROM ghcn
		  WHERE elev = 112.0
		  LIMIT 1
		)
		SELECT
		  r.gid AS road_id,
		  r.name AS road_name,
		  r.fclass AS road_class,
		  ST_Distance(r.geom::geography, s.geom::geography) AS distance_m
		FROM roads r
		JOIN station s
		  ON ST_DWithin(r.geom::geography, s.geom::geography, 5000)
		ORDER BY distance_m;
			✗	The Benchmark-Proposed Query correctly returns all roads within 5 km of any GHCN station with elev = 112.0. If there is exactly one such station, it answers the question; if there are multiple, it still returns the relevant roads (though potentially with duplicates), which aligns better than arbitrarily choosing one.
				The system-generated query is incorrect because it uses LIMIT 1 on the ghcn stations with elev = 112.0, arbitrarily selecting a single station if multiple exist. This can omit roads near other stations with the same elevation, making the result incomplete and non-deterministic.
					WITH stations AS (
					  SELECT geom
					  FROM ghcn
					  WHERE elev = 112.0
					)
					SELECT
					  r.gid AS road_id,
					  r.name AS road_name,
					  r.fclass AS road_class,
					
					MIN(ST_Distance(r.geom::geography, s.geom::geography)) AS distance_m
					FROM roads r
					JOIN stations s
					  ON ST_DWithin(r.geom::geography, s.geom::geography, 5000)
					GROUP BY r.gid, r.name, r.fclass
					ORDER BY distance_m;
						Reviewed is incorrect because it aggregates across all stations with elev = 112.0 and returns roads within 5 km of any of them (using MIN distance), which does not match the requirement to use a single specific station.	✗

Find all tracts that intersect the county with geoid '27117'.	SELECT t.* FROM tracts t JOIN counties c ON ST_Intersects(t.geom, c.geom) WHERE c.geoid = '27117';	SELECT DISTINCT t.geoid AS tract_geoid FROM tracts AS t JOIN counties AS c ON ST_Intersects(t.geom, c.geom) WHERE c.geoid = '27117';	✅	The systerm-generated query is correct because it returns all tract rows (t.*) that spatially intersect the county with geoid '27117', matching the request to "find all tracts."	SELECT DISTINCT t.geoid AS tract_geoid FROM tracts AS t JOIN counties AS c ON ST_Intersects(t.geom, c.geom) WHERE c.geoid = '27117';	Reviwer correctly use ST_Intersects between tracts and the county filtered by geoid '27117', returning the intersecting tracts. Benchamrk query returns full tract rows; System-generateds (identical) return distinct tract GEOIDs. Reviewer didn't make any changes.	✅

Which points of interest are within a 3 kilometer buffer of the protected area with unit_type 'National Monument'?	SELECT p.* FROM poi p JOIN ne_protected_areas pa ON ST_Intersects(p.geom, ST_Buffer(pa.geom::geography, 3000)::geometry) WHERE pa.unit_type = 'National Monument';	WITH national_monuments AS (
		  SELECT geom
		  FROM ne_protected_areas
		  WHERE unit_type = 'National Monument'
		)
		SELECT
		  p.gid AS poi_id,
		  p.name AS poi_name,
		  p.fclass AS poi_type
		FROM poi AS p
		INNER JOIN national_monuments AS nm
		  ON ST_DWithin(
		      p.geom::geography,
		      nm.geom::geography,
		      3000
		    );
			✅	The system-generated query is correct because ST_DWithin on geography measures geodesic distance; casting both geometries to geography and using 3000 returns POIs within 3 km of any National Monument polygon (including those inside it).	WITH national_monuments AS (
					  SELECT geom
					  FROM ne_protected_areas
					  WHERE unit_type = 'National Monument'
					)
					SELECT
					  p.gid AS poi_id,
					  p.name AS poi_name,
					  p.fclass AS poi_type
					FROM poi AS p
					INNER JOIN national_monuments AS nm
					  ON ST_DWithin(
					      p.geom::geography,
					      nm.geom::geography,
					      3000
					    );
						Reviewer didn't make any changes.
						All three correctly find POIs within 3 km of protected areas where unit_type = 'National Monument':
						  Benchamrk uses ST_Intersects with a 3,000 m geography buffer around the protected areas (equivalent to a 3 km “within” test).
						  System-generateds use ST_DWithin on geography with a 3,000 m distance, which is directly equivalent to being inside a 3 km buffer.
						No substantive logical errors relative to the question; all return POIs within the specified distance.
							✅

List the GHCN stations that intersect the state named 'South Dakota'.	SELECT g.* FROM ghcn g JOIN states s ON ST_Intersects(g.geom, s.geom) WHERE s.name = 'South Dakota';	SELECT g.station_id AS station_id, g.name AS station_name FROM ghcn AS g INNER JOIN states AS s ON ST_Intersects(g.geom, s.geom) WHERE s.name = 'South Dakota';	✅	Both queries correctly join GHCN stations to the state geometry using ST_Intersects and filter for s.name = 'South Dakota', thus listing stations that intersect the state.	SELECT g.station_id AS station_id, g.name AS station_name FROM ghcn AS g INNER JOIN states AS s ON ST_Intersects(g.geom, s.geom) WHERE s.name = 'South Dakota';	Reviewer didn't make any changes. All three correctly join ghcn to states, filter s.name = 'South Dakota', and use ST_Intersects to find stations intersecting the state.	✅

Level 3: Advanced

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation
Count the number of points of interest within each state and list the states ordered by the count descending.	SELECT s.name, COUNT(p.gid) AS poi_count FROM states s JOIN poi p ON ST_Within(s.geom, p.geom) GROUP BY s.name ORDER BY poi_count DESC;	SELECT s.name AS state_name, COUNT(*) AS poi_count FROM states AS s JOIN poi AS p ON ST_Contains(s.geom, p.geom) GROUP BY s.name ORDER BY poi_count DESC;	❌	System-generates is incorrect because, although it uses ST_Contains, the inner join excludes states with zero POIs, so it does not list "each state."	SELECT s.name AS state_name, COUNT(p.gid) AS poi_count FROM states AS s LEFT JOIN poi AS p ON ST_Contains(s.geom, p.geom) GROUP BY s.name ORDER BY poi_count DESC;	Reviwer is correct because it uses ST_Contains to enforce "within" semantics (excluding boundary-only points) and a LEFT JOIN with COUNT(p.gid) so every state is listed, including those with zero POIs, ordered by count descending. Unreviwed is incorrect because, although it uses ST_Contains, the inner join excludes states with zero POIs, so it does not list "each state."	✅

Find the total length in kilometers of roads intersecting each protected area, grouped by unit_type.	SELECT pa.unit_type, SUM(ST_Length(r.geom::geography))/1000 AS total_length_km FROM ne_protected_areas pa JOIN roads r ON ST_Intersects(pa.geom, r.geom) GROUP BY pa.unit_type;	WITH inter AS (SELECT p.gid, p.unit_name, p.unit_type, ST_CollectionExtract(ST_Intersection(r.geom, p.geom), 2) AS seg FROM roads r INNER JOIN ne_protected_areas p ON ST_Intersects(r.geom, p.geom)) SELECT gid, unit_name, unit_type, SUM(ST_Length(seg::geography)) / 1000.0 AS total_km FROM inter WHERE NOT ST_IsEmpty(seg) GROUP BY gid, unit_name, unit_type ORDER BY unit_type, unit_name;	✗	System-gnerated query is incorrect because it groups by individual protected areas (gid, unit_name) rather than by unit_type as requested.	WITH intersected AS (SELECT p.unit_type, ST_CollectionExtract(ST_Intersection(r.geom, p.geom), 2) AS seg FROM roads r INNER JOIN ne_protected_areas p ON ST_Intersects(r.geom, p.geom)) SELECT unit_type, SUM(ST_Length(seg::geography)) / 1000.0 AS total_km FROM intersected WHERE NOT ST_IsEmpty(seg) GROUP BY unit_type ORDER BY unit_type;	Reviwer query is also incorrect because it groups by individual protected areas rather than by unit_type..	✗

Calculate the average area in square kilometers of counties per region, using states table.	SELECT s.region, AVG(ST_Area(c.geom::geography)/1000000) AS avg_area_sqkm FROM states s JOIN counties c ON ST_Intersects(s.geom, c.geom) GROUP BY s.region;	WITH county_areas AS (SELECT s.region, CASE WHEN ST_SRID(c.geom) IN (4326, 0) THEN ST_Area(geography(c.geom)) / 1000000.0 ELSE ST_Area(c.geom) / 1000000.0 END AS area_sq_km FROM counties c INNER JOIN states s ON substring(c.geoid::text, 1, char_length(s.geoid::text)) = s.geoid::text WHERE c.geom IS NOT NULL) SELECT region AS region, AVG(area_sq_km) AS avg_area_sq_km FROM county_areas GROUP BY region ORDER BY region;	✗	System-geneated is incorrect because its area logic is unreliable: they use ST_Area(geometry) and divide by 1e6, which yields wrong units (degrees squared or feet squared).	WITH county_areas AS (SELECT s.region, CASE WHEN ST_SRID(c.geom) IN (4326, 0) THEN ST_Area(geography(c.geom)) / 1000000.0 ELSE ST_Area(c.geom) / 1000000.0 END AS area_sq_km FROM counties c INNER JOIN states s ON substring(c.geoid::text, 1, char_length(s.geoid::text)) = s.geoid::text WHERE c.geom IS NOT NULL) SELECT region AS region, AVG(area_sq_km) AS avg_area_sq_km FROM county_areas GROUP BY region ORDER BY region;	Benchmark correctly computes county areas in square kilometers using geography (ST_Area(c.geom::geography)/1e6) and assigns counties to regions via the states table, then averages by region. The spatial join is appropriate for mapping counties to their states to access region. System-geneateds are incorrect because their area logic is unreliable: they use ST_Area(geometry) and divide by 1e6, which yields wrong units (degrees squared or feet squared).	✗

Count the number of GHCN stations in each time zone and list those with more than 5 stations.	WITH station_count AS (SELECT tz.name, COUNT(g.station_id) AS count FROM ne_time_zones tz JOIN ghcn g ON ST_Within(tz.geom, g.geom) GROUP BY tz.name) SELECT * FROM station_count WHERE count > 5;	SELECT tz.zone AS zone, COUNT(DISTINCT g.station_id) AS station_count FROM ghcn AS g INNER JOIN ne_time_zones AS tz ON ST_Contains(tz.geom, g.geom) GROUP BY tz.zone HAVING COUNT(DISTINCT g.station_id) > 5 ORDER BY station_count DESC, zone;	✅	The system-generated query correctly groups by time zone, counts unique stations using COUNT(DISTINCT g.station_id) to avoid double-counting, applies the HAVING > 5 filter, and returns the zone and count. The spatial predicate ST_Contains(polygon, point) is appropriate for associating stations (points) with time zones (polygons).	SELECT tz.zone AS zone, COUNT(DISTINCT g.station_id) AS station_count FROM ghcn AS g INNER JOIN ne_time_zones AS tz ON ST_Contains(tz.geom, g.geom) GROUP BY tz.zone HAVING COUNT(DISTINCT g.station_id) > 5 ORDER BY station_count DESC, zone;	Reviewer didn't make any changes. System-generated queries correctly count distinct GHCN stations per time zone using a containment join and filter to zones with more than 5 stations via HAVING.	✅
Find the maximum elevation of GHCN stations within each state, grouped by state name.	SELECT s.name, MAX(g.elev) AS max_elev FROM states s JOIN ghcn g ON ST_Within(s.geom, g.geom) GROUP BY s.name;	SELECT s.name AS state_name, MAX(g.elev) AS max_elevation FROM states AS s LEFT JOIN ghcn AS g ON ST_Contains(s.geom, g.geom) GROUP BY s.name ORDER BY s.name;	✅	The System-generated query correctly computes the maximum elevation of GHCN stations that are within each state's polygon using ST_Contains, and returns one row per state (including states with no stations via LEFT JOIN).	SELECT s.name AS state_name, MAX(g.elev) AS max_elevation FROM states AS s LEFT JOIN ghcn AS g ON ST_Contains(s.geom, g.geom) GROUP BY s.name ORDER BY s.name;	Reviewer didn't make any changes. System-generated queries both are correct: they use ST_Contains to enforce "within" and LEFT JOIN to keep all states, grouping by state name and taking MAX(elev).	✅

Calculate the total area in square kilometers of tracts intersecting block groups in each county.	SELECT b.county, SUM(ST_Area(t.geom::geography)/1000000) AS total_area_sqkm FROM blockgroups b JOIN tracts t ON ST_Intersects(b.geom, t.geom) GROUP BY b.county;	WITH intersections AS (SELECT bg.county, ST_Area(ST_Intersection(t.geom, bg.geom)::geography) / 1e6 AS area_km2 FROM tracts t INNER JOIN blockgroups bg ON ST_Intersects(t.geom, bg.geom) WHERE ST_IsValid(t.geom) AND ST_IsValid(bg.geom) AND ST_Area(ST_Intersection(t.geom, bg.geom)::geography) > 0) SELECT c.name AS county_name, c.state AS state_fips, ROUND(SUM(i.area_km2)::numeric, 6) AS total_intersect_area_km2 FROM intersections i INNER JOIN counties c ON i.county = c.geoid GROUP BY c.name, c.state ORDER BY c.state, c.name;	✅	The System-generated query correctly computes the area of the intersections between tracts and block groups (ST_Intersection), converts to km², and aggregates by county. This matches the intent: sum only the tract portions that actually intersect block groups, per county.	WITH intersections AS (SELECT bg.county, ST_Area(ST_Intersection(t.geom, bg.geom)::geography) / 1e6 AS area_km2 FROM tracts t INNER JOIN blockgroups bg ON ST_Intersects(t.geom, bg.geom) WHERE ST_IsValid(t.geom) AND ST_IsValid(bg.geom) AND ST_Area(ST_Intersection(t.geom, bg.geom)::geography) > 0) SELECT c.name AS county_name, c.state AS state_fips, ROUND(SUM(i.area_km2)::numeric, 6) AS total_intersect_area_km2 FROM intersections i INNER JOIN counties c ON i.county = c.geoid GROUP BY c.name, c.state ORDER BY c.state, c.name;	System-generated queries correctly compute the area of the actual intersections between tracts and block groups, convert to square kilometers, and aggregate by county, matching the question’s intent.	✅

Find roads that intersect more than one county and count how many such roads there are.	SELECT COUNT(*) FROM (SELECT r.gid, COUNT(c.gid) FROM roads r JOIN counties c ON ST_Intersects(r.geom, c.geom) GROUP BY r.gid HAVING COUNT(c.gid) > 1) AS sub;	WITH intersections AS (
		  SELECT
		    bg.county,
		
		ST_Area(ST_Intersection(t.geom, bg.geom)::geography) / 1e6 AS area_km2
		  FROM tracts t
		  INNER JOIN blockgroups bg
		    ON ST_Intersects(t.geom, bg.geom)
		  WHERE ST_IsValid(t.geom) AND ST_IsValid(bg.geom)
		    AND
		ST_Area(ST_Intersection(t.geom, bg.geom)::geography) > 0
		)
		SELECT
		  c.name AS county_name,
		  c.state AS state_fips,
		
		ROUND(SUM(i.area_km2)::numeric, 6) AS total_intersect_area_km2
		FROM intersections i
		INNER JOIN counties c
		  ON i.county = c.geoid
		GROUP BY c.name, c.state
		ORDER BY c.state, c.name;
			✅	The System-generated query is correct because it:
				  Identifies roads intersecting counties via ST_Intersects.
					WITH
					road_county_intersections AS (
					  SELECT r.gid AS road_id, c.geoid AS county_id
					  FROM roads AS r
					  INNER JOIN counties AS c
					    ON ST_Intersects(r.geom, c.geom)
					),
					road_counts AS (
					  SELECT road_id, COUNT(DISTINCT county_id) AS county_count
					  FROM road_county_intersections
					  GROUP BY road_id
					),
					multi_county_roads AS (
					  SELECT road_id, county_count
					  FROM road_counts
					  WHERE county_count > 1
					)
					SELECT
					  road_id,
					  county_count,
					  (SELECT COUNT(*) FROM multi_county_roads) AS total_roads_multi_county
					FROM multi_county_roads
					ORDER BY road_id;
						Reviewer didn't make any changes.
						System-generated queries correctly list each road that intersects more than one county (using COUNT(DISTINCT county_id)) and also provide the total number of such roads. Benchmark only returns the total count and does not list the roads; it also doesn’t use DISTINCT, which can overcount if duplicate county IDs exist.
							✅

List the protected areas with the highest number of POIs within them, using a subquery to find the max count.	SELECT pa.name FROM ne_protected_areas pa WHERE (SELECT COUNT(p.gid) FROM poi p WHERE ST_Within(p.geom, pa.geom)) = (SELECT MAX(count) FROM (SELECT COUNT(p.gid) AS count FROM poi p JOIN ne_protected_areas pa2 ON ST_Within(p.geom, pa2.geom) GROUP BY pa2.gid) AS sub);	WITH area_poi_counts AS (SELECT npa.gid AS area_id, npa.unit_name AS unit_name, npa.name AS alt_name, COUNT(*) AS poi_count FROM ne_protected_areas AS npa INNER JOIN poi AS p ON ST_Contains(npa.geom, p.geom) GROUP BY npa.gid, npa.unit_name, npa.name), max_count AS (SELECT MAX(poi_count) AS max_poi_count FROM area_poi_counts) SELECT apc.area_id, apc.unit_name, apc.alt_name, apc.poi_count FROM area_poi_counts AS apc CROSS JOIN max_count AS mc WHERE apc.poi_count = mc.max_poi_count;	✅	The System-generated query correctly counts POIs strictly within each protected area using ST_Contains, computes the maximum via a subquery (CTE), and returns all areas whose count equals that maximum.	WITH area_poi_counts AS (SELECT npa.gid AS area_id, npa.unit_name AS unit_name, npa.name AS alt_name, COUNT(*) AS poi_count FROM ne_protected_areas AS npa INNER JOIN poi AS p ON ST_Contains(npa.geom, p.geom) GROUP BY npa.gid, npa.unit_name, npa.name), max_count AS (SELECT MAX(poi_count) AS max_poi_count FROM area_poi_counts) SELECT apc.area_id, apc.unit_name, apc.alt_name, apc.poi_count FROM area_poi_counts AS apc CROSS JOIN max_count AS mc WHERE apc.poi_count = mc.max_poi_count;	System-generated queries are correct: they count POIs strictly within each protected area using ST_Contains, use a subquery (CTE) to get the max poi_count, and return all areas with that maximum.	✅

Calculate the average maxspeed of roads within each nps_region of protected areas.	SELECT pa.nps_region, AVG(r.maxspeed) AS avg_maxspeed FROM ne_protected_areas pa JOIN roads r ON ST_Within(pa.geom, r.geom) GROUP BY pa.nps_region;	SELECT pa.nps_region AS nps_region, AVG(r.maxspeed)::numeric AS avg_maxspeed FROM roads r JOIN ne_protected_areas pa ON ST_Intersects(r.geom, pa.geom) WHERE r.maxspeed IS NOT NULL GROUP BY pa.nps_region ORDER BY pa.nps_region;	❌	Unreviwed is incorrect because a road intersecting multiple protected polygons in the same nps_region will be counted multiple times, biasing the average..	WITH region_geoms AS (SELECT nps_region, ST_UnaryUnion(ST_Collect(geom)) AS geom FROM ne_protected_areas GROUP BY nps_region), region_roads AS (SELECT DISTINCT rg.nps_region, r.gid, r.maxspeed FROM roads r JOIN region_geoms rg ON ST_Wtihin(r.geom, rg.geom) WHERE r.maxspeed IS NOT NULL) SELECT nps_region, AVG(maxspeed)::numeric AS avg_maxspeed FROM region_roads GROUP BY nps_region ORDER BY nps_region;	Reviwer is correct because it dissolves protected areas by nps_region, joins roads to the region-level geometry, and deduplicates each road per region (DISTINCT by road id), preventing double-counting when a road intersects multiple protected polygons within the same region. It also excludes NULL maxspeed values before averaging.	✅

Count the number of block groups per state and list states with the average area greater than 100 square kilometers.	WITH bg_stats AS (SELECT s.name, COUNT(b.gid) AS count, AVG(ST_Area(b.geom::geography)/1000000) AS avg_area FROM states s JOIN blockgroups b ON ST_Within(s.geom, b.geom) GROUP BY s.name) SELECT * FROM bg_stats WHERE avg_area > 100;	WITH blockgroups_with_area AS (SELECT state, CASE WHEN geom IS NOT NULL THEN ST_Area(ST_Transform(geom, 6933)) / 1000000.0 ELSE NULL END AS area_km2 FROM blockgroups), state_aggregates AS (SELECT state, COUNT(*) AS blockgroup_count, AVG(area_km2) AS avg_area_km2 FROM blockgroups_with_area WHERE state IS NOT NULL AND area_km2 IS NOT NULL GROUP BY state) SELECT state AS state, blockgroup_count AS blockgroup_count FROM state_aggregates WHERE avg_area_km2 > 100 ORDER BY state;	✗	The System-generated query is incorrect because it transforms all geometries to EPSG:6933, which is an Alaska-specific equal-area CRS. Using it nationwide (e.g., for the contiguous U.S. or Hawaii) will produce incorrect areas, leading to wrong averages and filtering. Additionally, it excludes block groups with NULL geometries from both the average and the count, potentially undercounting.	WITH blockgroups_with_area AS (SELECT state, CASE WHEN geom IS NOT NULL THEN ST_Area(ST_Transform(geom, 6933)) / 1000000.0 ELSE NULL END AS area_km2 FROM blockgroups), state_aggregates AS (SELECT state, COUNT(*) AS blockgroup_count, AVG(area_km2) AS avg_area_km2 FROM blockgroups_with_area WHERE state IS NOT NULL AND area_km2 IS NOT NULL GROUP BY state) SELECT state AS state, blockgroup_count AS blockgroup_count FROM state_aggregates WHERE avg_area_km2 > 100 ORDER BY state;	Reviwer query has the same problem as unreviewed.	✗

Find the total length in meters of roads with maxspeed over 50 that intersect tracts.	SELECT SUM(ST_Length(r.geom::geography)) AS total_length_m FROM roads r JOIN tracts t ON ST_Intersects(r.geom, t.geom) WHERE r.maxspeed > 50;	WITH filtered_roads AS (SELECT r.geom FROM roads r WHERE r.maxspeed > 50), intersections AS (SELECT ST_Intersection(ST_Transform(fr.geom, 3857), ST_Transform(t.geom, 3857)) AS geom FROM filtered_roads fr INNER JOIN tracts t ON ST_Intersects(fr.geom, t.geom)) SELECT COALESCE(SUM(ST_Length(geom)), 0) AS total_length_meters FROM intersections WHERE NOT ST_IsEmpty(geom);	✅	The system-generated query correctly clips roads to the portions that actually intersect tracts (using ST_Intersection) and then sums the length of those intersecting segments, which matches the intent of measuring only the road length within/intersecting tracts. It also computes length in a projected CRS (EPSG:3857), yielding meters. The Benchmark-Proposed Query sums the full length of any road that merely intersects a tract, not just the intersecting portion, thereby overcounting lengths outside tracts. While it returns meters via geography, it does not restrict to the intersection geometry.	WITH filtered_roads AS (SELECT r.geom FROM roads r WHERE r.maxspeed > 50), intersections AS (SELECT ST_Intersection(ST_Transform(fr.geom, 3857), ST_Transform(t.geom, 3857)) AS geom FROM filtered_roads fr INNER JOIN tracts t ON ST_Intersects(fr.geom, t.geom)) SELECT COALESCE(SUM(ST_Length(geom)), 0) AS total_length_meters FROM intersections WHERE NOT ST_IsEmpty(geom);	system-generated queries both correctly compute the length of the portions of roads (maxspeed > 50) that actually intersect tracts by clipping roads to tracts (ST_Intersection) and summing their lengths in a meter-based projection.	✅

Group POIs by fclass and count those within counties of state '42'.	SELECT p.fclass, COUNT(p.gid) FROM poi p JOIN counties c ON ST_Within(p.geom, c.geom) WHERE c.state = '42' GROUP BY p.fclass;	WITH target_state AS (
		  SELECT s.geom
		  FROM states AS s
		  WHERE s.statefp = '42'
		),
		state_counties AS (
		  SELECT c.geom
		  FROM counties AS c
		  INNER JOIN target_state AS ts
		    ON ST_Intersects(c.geom, ts.geom)
		)
		SELECT p.fclass AS fclass,
		       COUNT(*) AS poi_count
		FROM poi AS p
		INNER JOIN state_counties AS sc
		  ON ST_Within(p.geom, sc.geom)
		WHERE p.fclass IS NOT NULL
		GROUP BY p.fclass
		ORDER BY poi_count DESC;
			✅	The query is correct because it restricts counties to state '42' via an attribute filter (c.state = '42') and counts POIs that spatially intersect those county polygons, grouping by fclass. For point-in-polygon, ST_Intersects is appropriate and will count POIs located in those counties.	WITH target_state AS (
					  SELECT s.geom
					  FROM states AS s
					  WHERE s.statefp = '42'
					),
					state_counties AS (
					  SELECT c.geom
					  FROM counties AS c
					  INNER JOIN target_state AS ts
					    ON ST_Intersects(c.geom, ts.geom)
					)
					SELECT p.fclass AS fclass,
					       COUNT(*) AS poi_count
					FROM poi AS p
					INNER JOIN state_counties AS sc
					  ON ST_Within(p.geom, sc.geom)
					WHERE p.fclass IS NOT NULL
					GROUP BY p.fclass
					ORDER BY poi_count DESC;
						All three queries correctly count POIs grouped by fclass within counties belonging to state '42'.	✅

Use a CTE to find the largest state by land area and count the GHCN stations within it.	WITH largest_state AS (SELECT geom FROM states ORDER BY aland DESC LIMIT 1) SELECT COUNT(g.station_id) FROM ghcn g JOIN largest_state ls ON ST_Within(g.geom, ls.geom);	WITH largest_state AS (SELECT name, geom FROM states WHERE aland = (SELECT MAX(aland) FROM states) ORDER BY aland DESC LIMIT 1) SELECT ls.name AS state_name, COUNT(DISTINCT g.station_id) AS station_count FROM largest_state ls JOIN ghcn g ON ST_Contains(ls.geom, g.geom) GROUP BY ls.name;	✅	The System-generated query correctly identifies the largest state by land area using a CTE and returns both the state's name and the count of GHCN stations within it. It uses a point-in-polygon relationship (ST_Contains) appropriate for “within,” and counts distinct station_ids to avoid duplicates. The Benchmark-Proposed Query only returns the count and omits the state identification, which the question implies should be included. It also uses ST_Intersects, which can include stations exactly on the boundary rather than strictly “within,” making it less aligned with the intent.	WITH largest_state AS (SELECT name, geom FROM states WHERE aland = (SELECT MAX(aland) FROM states) ORDER BY aland DESC LIMIT 1) SELECT ls.name AS state_name, COUNT(DISTINCT g.station_id) AS station_count FROM largest_state ls JOIN ghcn g ON ST_Contains(ls.geom, g.geom) GROUP BY ls.name;	Both reviwed and unreviwed correctly use a CTE to select the largest state by land area and then count DISTINCT GHCN stations strictly within it via ST_Contains, returning the state name and the count.	✅
Calculate the sum of water areas from states that contain protected areas of type 'National Park'.	SELECT SUM(s.awater) FROM states s JOIN ne_protected_areas pa ON ST_Contains(s.geom, pa.geom) WHERE pa.unit_type = 'National Park' GROUP BY pa.unit_type;	SELECT SUM(s.awater) AS total_awater FROM states AS s WHERE EXISTS (SELECT 1 FROM ne_protected_areas AS p WHERE p.unit_type = 'National Park' AND ST_Contains(s.geom, p.geom));	✅	The System-generated query is correct because it sums each state's awater once if the state intersects at least one protected area of type 'National Park' (using EXISTS), matching the intent to aggregate by unique states that contain such parks.	SELECT SUM(s.awater) AS total_awater FROM states AS s WHERE EXISTS (SELECT 1 FROM ne_protected_areas AS p WHERE p.unit_type = 'National Park' AND ST_Contains(s.geom, p.geom));	Bothe sytem-generateds correctly sum each state's awater once if it intersects at least one National Park, using EXISTS to avoid double counting.	✅

Find the average temperature (TMAX value) from GHCN stations in each protected area, grouped by unit_name.	SELECT pa.unit_name, AVG(g.value) AS avg_tmax FROM ne_protected_areas pa JOIN ghcn g ON ST_Intersects(pa.geom, g.geom) WHERE g.element = 'TMAX' GROUP BY pa.unit_name;	WITH tmax_clean AS (
		  SELECT
		    g.geom,
		    g.value
		  FROM ghcn AS g
		  WHERE g.element = 'TMAX'
		    AND (g.qflag IS NULL OR g.qflag = 'None')
		),
		joined AS (
		  SELECT
		    p.unit_name,
		    c.value
		  FROM ne_protected_areas AS p
		  JOIN tmax_clean AS c
		    ON ST_Intersects(c.geom, p.geom)
		)
		SELECT
		  unit_name,
		  AVG(value) AS avg_tmax
		FROM joined
		GROUP BY unit_name
		ORDER BY unit_name;
			❌	Unreviwed is incorrect because it filters on qflag = 'None', which is not a standard representation in GHCN; valid/unflagged records are typically qflag IS NULL (and sometimes empty string). This condition may exclude valid rows or include invalid ones, leading to incorrect results.	WITH tmax_clean AS (
					  SELECT
					    g.geom,
					    g.value
					  FROM ghcn AS g
					  WHERE g.element = 'TMAX'
					    AND (g.qflag IS NULL OR g.qflag = '')
					),
					joined AS (
					  SELECT
					    p.unit_name,
					    c.value
					  FROM ne_protected_areas AS p
					  JOIN tmax_clean AS c
					    ON ST_Intersects(c.geom, p.geom)
					)
					SELECT
					  unit_name,
					  AVG(value) AS avg_tmax
					FROM joined
					GROUP BY unit_name
					ORDER BY unit_name;
						Reviwed is correct because it:
						  Selects only TMAX measurements.
						  Joins GHCN points to protected areas via spatial intersection.
						  Excludes suspect measurements by filtering out records with a non-null/non-empty qflag.
						  Groups by unit_name and computes the average value as requested.
							✅

Count the number of roads per time zone and list those with total length over 100 kilometers.	WITH road_stats AS (SELECT tz.name, COUNT(r.gid) AS count, SUM(ST_Length(r.geom::geography))/1000 AS total_km FROM ne_time_zones tz JOIN roads r ON ST_Within(tz.geom, r.geom) GROUP BY tz.name) SELECT * FROM road_stats WHERE total_km > 100;	WITH clipped AS (SELECT tz.zone AS time_zone, r.gid, ST_Intersection(r.geom, tz.geom) AS geom_clip FROM roads r INNER JOIN ne_time_zones tz ON r.geom IS NOT NULL AND tz.geom IS NOT NULL AND ST_Intersects(r.geom, tz.geom)), lengths AS (SELECT time_zone, gid, ST_Length(geom_clip::geography) AS length_m FROM clipped WHERE geom_clip IS NOT NULL) SELECT time_zone, COUNT(DISTINCT gid) AS road_count, SUM(length_m) / 1000.0 AS total_length_km FROM lengths GROUP BY time_zone HAVING SUM(length_m) / 1000.0 > 100 ORDER BY total_length_km DESC;	✅	The System-generated query correctly clips each road to the time zone geometry before measuring length, sums only the in-zone lengths, and counts distinct roads per time zone. It then filters to those with total in-zone length > 100 km, matching the intent. The Benchmark-Proposed Query sums the full length of each road that intersects a time zone without clipping to the zone, overestimating total length.	WITH clipped AS (SELECT tz.zone AS time_zone, r.gid, ST_Intersection(r.geom, tz.geom) AS geom_clip FROM roads r INNER JOIN ne_time_zones tz ON r.geom IS NOT NULL AND tz.geom IS NOT NULL AND ST_Intersects(r.geom, tz.geom)), lengths AS (SELECT time_zone, gid, ST_Length(geom_clip::geography) AS length_m FROM clipped WHERE geom_clip IS NOT NULL) SELECT time_zone, COUNT(DISTINCT gid) AS road_count, SUM(length_m) / 1000.0 AS total_length_km FROM lengths GROUP BY time_zone HAVING SUM(length_m) / 1000.0 > 100 ORDER BY total_length_km DESC;	Reviwed and unreviwed queries are the same. Reviwer is correct because it clips roads to each time zone (ST_Intersection), sums only the in-zone lengths (geography meters -> km), counts distinct roads per zone, and filters zones with total length > 100 km.	✅

Use subquery to find POIs within the county with the largest area.	SELECT p.* FROM poi p WHERE ST_Within(p.geom, (SELECT geom FROM counties ORDER BY ST_Area(geom::geography) DESC LIMIT 1));	SELECT p.name AS poi_name, p.fclass AS poi_class, p.geom AS poi_geom FROM poi AS p WHERE ST_Contains((SELECT c.geom FROM counties AS c WHERE c.geom IS NOT NULL ORDER BY ST_Area(c.geom) DESC LIMIT 1), p.geom);	❌	The System-generated query is correct because it selects the county with the largest area via a subquery and uses ST_Contains to return only POIs strictly within that county polygon, matching the “within” requirement.	SELECT p.name AS poi_name, p.fclass AS poi_class, p.geom AS poi_geom FROM poi AS p WHERE ST_Contains((SELECT c.geom FROM counties AS c WHERE c.geom IS NOT NULL ORDER BY ST_Area(c.geom) DESC LIMIT 1), p.geom);	Benchamrk correctly uses a subquery to select the county with the largest area using ST_Area(geom::geography), which yields a proper area comparison, and then returns POIs that intersect that county. For point POIs, ST_Intersects effectively captures points within (including boundary). System-gnerateds may select the wrong county because it orders by ST_Area(c.geom) without casting to geography; if the data are in a geographic CRS (e.g., EPSG:4326), this compares degrees², not true area. This can misidentify the largest county.	❌

Calculate the total area in square meters of block groups intersecting multiple tracts.	SELECT SUM(ST_Area(b.geom::geography)) FROM blockgroups b WHERE (SELECT COUNT(t.gid) FROM tracts t WHERE ST_Intersects(b.geom, t.geom)) > 1;	WITH bg_tract_counts AS (
		  SELECT
		    bg.geoid AS blockgroup_geoid,
		    bg.geom,
		    COUNT(DISTINCT t.geoid) AS tract_count
		  FROM blockgroups AS bg
		  JOIN tracts AS t
		    ON ST_Intersects(bg.geom, t.geom)
		  GROUP BY bg.geoid, bg.geom
		)
		SELECT
		  blockgroup_geoid,
		
		ST_Area(bg_tract_counts.geom::geography) AS total_area_sqm
		FROM bg_tract_counts
		WHERE tract_count > 1;
			❌	The Benchmark-Proposed Query correctly returns a single scalar: the sum of areas (in square meters) of all block groups that intersect more than one tract, matching “total area.”
				The System-generated query returns one row per qualifying block group with its area, not the aggregated total. It does not compute the overall total area requested.
					WITH bg_tract_counts AS (
					  SELECT
					    bg.geoid AS blockgroup_geoid,
					    bg.geom,
					    COUNT(DISTINCT t.geoid) AS tract_count
					  FROM blockgroups AS bg
					  JOIN tracts AS t
					    ON ST_Intersects(bg.geom, t.geom)
					  GROUP BY bg.geoid, bg.geom
					)
					SELECT
					  blockgroup_geoid,
					
					ST_Area(bg_tract_counts.geom::geography) AS total_area_sqm
					FROM bg_tract_counts
					WHERE tract_count > 1;
						Benchmark correctly returns a single total area (in square meters) by summing the areas of all block groups that intersect more than one tract, matching the request for the total area.
						reviwed and unreviwed queries (which are identical) return per-block-group areas for those intersecting multiple tracts, not the total sum. They therefore do not answer the “total area” requirement.
							❌

Group counties by state and calculate the average perimeter in kilometers.	SELECT c.state, AVG(ST_Perimeter(c.geom::geography)/1000) AS avg_perim_km FROM counties c GROUP BY c.state;	SELECT
		  state AS state,
		
		AVG(ST_Perimeter(geom::geography) / 1000.0) AS avg_perimeter_km
		FROM counties
		WHERE geom IS NOT NULL
		GROUP BY state;
			✅	Both queries group counties by state and compute the average perimeter in kilometers using ST_Perimeter on geography (meters) divided by 1000, which matches the intent.	SELECT
					  state AS state,
					
					AVG(ST_Perimeter(geom::geography) / 1000.0) AS avg_perimeter_km
					FROM counties
					WHERE geom IS NOT NULL
					GROUP BY state;
						All three queries group by state and compute AVG(ST_Perimeter(geom::geography)/1000) in kilometers, which answers the question.	✅

Find the number of GHCN stations per county, using joins with states for filtering states with aland > 100000000000.	SELECT c.name, COUNT(g.station_id) FROM counties c JOIN states s ON ST_Intersects(c.geom, s.geom) JOIN ghcn g ON ST_Intersects(c.geom, g.geom) WHERE s.aland > 100000000000 GROUP BY c.name;	WITH counties_in_big_states AS (SELECT c.geoid, c.geom FROM counties c JOIN states s ON c.state = s.statefp WHERE s.aland > 100000000000 AND c.geom IS NOT NULL), stations_in_counties AS (SELECT c.geoid AS county_geoid, g.station_id FROM ghcn g JOIN counties_in_big_states c ON g.geom IS NOT NULL AND ST_Within(g.geom, c.geom)) SELECT county_geoid, COUNT(station_id) AS station_count FROM stations_in_counties GROUP BY county_geoid ORDER BY county_geoid;	✅	The System-generated query correctly filters counties to those in states with aland > 100000000000 via an attribute join (c.state = s.statefp), then counts GHCN stations that are spatially within each county polygon. It groups by a unique county identifier (geoid), avoiding duplication and name collisions.	WITH counties_in_big_states AS (SELECT c.geoid, c.geom FROM counties c JOIN states s ON c.state = s.statefp WHERE s.aland > 100000000000 AND c.geom IS NOT NULL), stations_in_counties AS (SELECT c.geoid AS county_geoid, g.station_id FROM ghcn g JOIN counties_in_big_states c ON g.geom IS NOT NULL AND ST_Within(g.geom, c.geom)) SELECT county_geoid, COUNT(station_id) AS station_count FROM stations_in_counties GROUP BY county_geoid ORDER BY county_geoid;	Unreviwed (and identical Reviwed) correctly filters counties by joining to states on state FIPS and applying s.aland > 100000000000, then counts stations strictly within each county, grouping by a unique county identifier (geoid). Reviewer didn't make any changes.	✅

Use CTE to buffer protected areas by 10km and count POIs within the buffers.	WITH buffered_pa AS (SELECT ST_Buffer(pa.geom::geography, 10000)::geometry AS buff_geom FROM ne_protected_areas pa) SELECT COUNT(p.gid) FROM poi p JOIN buffered_pa bp ON ST_Within(p.geom, bp.buff_geom);	WITH buffered_areas AS (SELECT gid, unit_name, ST_Buffer(geom::geography, 10000)::geometry AS geom_buffer FROM ne_protected_areas) SELECT b.unit_name AS area_name, COUNT(p.gid) AS poi_count FROM buffered_areas b LEFT JOIN poi p ON ST_Witin(b.geom_buffer, p.geom) GROUP BY b.unit_name ORDER BY b.unit_name;	✅	The system-generated query correctly buffers each protected area by 10 km using a CTE and counts POIs that intersect those buffers. It returns counts per protected area (by unit_name) and includes areas with zero POIs via LEFT JOIN.	WITH buffered_areas AS (SELECT gid, unit_name, ST_Buffer(geom::geography, 10000)::geometry AS geom_buffer FROM ne_protected_areas) SELECT b.unit_name AS area_name, COUNT(p.gid) AS poi_count FROM buffered_areas b LEFT JOIN poi p ON ST_Within(b.geom_buffer, p.geom) GROUP BY b.unit_name ORDER BY b.unit_name;	Query system-generated queries correctly use a CTE to buffer each protected area by 10 km and return POI counts per area (grouped by unit_name). The geography->geometry buffer ensures a 10,000-meter buffer, and the LEFT JOIN preserves areas with zero POIs.	✅
Calculate the min and max latitude of points in each time zone.	SELECT tz.name, MIN(ST_Y(g.geom)) AS min_lat, MAX(ST_Y(g.geom)) AS max_lat FROM ne_time_zones tz JOIN ghcn g ON ST_Intersects(tz.geom, g.geom) GROUP BY tz.name;	SELECT tz.zone AS zone, MIN(g.lat) AS min_lat, MAX(g.lat) AS max_lat FROM ne_time_zones AS tz JOIN ghcn AS g ON ST_Intersects(g.geom, tz.geom) GROUP BY tz.zone;	✅	The System-generated query correctly groups by the time zone identifier (tz.zone) and computes min/max latitude using the latitude attribute (g.lat), which directly reflects the points' latitude values. It uses an appropriate spatial join (ST_Intersects) between points and time zone geometries.	SELECT tz.zone AS zone, MIN(g.lat) AS min_lat, MAX(g.lat) AS max_lat FROM ne_time_zones AS tz JOIN ghcn AS g ON ST_Intersects(g.geom, tz.geom) GROUP BY tz.zone;	Unreviwed and reviwed are identical and correctly compute min/max latitude per time zone by spatially joining points to time zone polygons and aggregating the latitude column (g.lat) grouped by tz.zone.	✅

Find tracts with area larger than the average tract area, and count them per geoid prefix (first 5 chars).	WITH avg_area AS (SELECT AVG(ST_Area(geom::geography)) AS avg FROM tracts) SELECT SUBSTRING(t.geoid, 1, 5), COUNT(t.gid) FROM tracts t, avg_area a WHERE ST_Area(t.geom::geography) > a.avg GROUP BY SUBSTRING(t.geoid, 1, 5);	WITH avg_area AS (SELECT AVG(ST_Area(geom)) AS avg_geom_area FROM tracts WHERE geom IS NOT NULL), filtered AS (SELECT SUBSTRING(geoid, 1, 5) AS geoid_prefix FROM tracts, avg_area WHERE geom IS NOT NULL AND ST_Area(geom) > avg_area.avg_geom_area) SELECT geoid_prefix, COUNT(*) AS tract_count FROM filtered GROUP BY geoid_prefix ORDER BY geoid_prefix;	✅	The System-generated query correctly computes the average area over non-null geometries, filters tracts with area greater than that average using the same units/function, extracts the first 5 characters of geoid, and counts with COUNT(*). It does not assume the presence of any additional columns.	WITH avg_area AS (SELECT AVG(ST_Area(geom)) AS avg_geom_area FROM tracts WHERE geom IS NOT NULL), filtered AS (SELECT SUBSTRING(geoid, 1, 5) AS geoid_prefix FROM tracts, avg_area WHERE geom IS NOT NULL AND ST_Area(geom) > avg_area.avg_geom_area) SELECT geoid_prefix, COUNT(*) AS tract_count FROM filtered GROUP BY geoid_prefix ORDER BY geoid_prefix;	All three compute a global average area and then count tracts whose area exceeds that average, grouped by SUBSTRING(geoid, 1, 5). Benchamrk uses ST_Area(geom::geography); system queries use ST_Area(geom). Both approaches are internally consistent with their chosen area metric and produce the requested counts per geoid prefix.	✅

Group roads by fclass and sum lengths in kilometers for those intersecting states in region '3'.	SELECT r.fclass, SUM(ST_Length(r.geom::geography))/1000 AS total_km FROM roads r JOIN states s ON ST_Intersects(r.geom, s.geom) WHERE s.region = '3' GROUP BY r.fclass;	WITH target_states AS (SELECT s.geom FROM states AS s WHERE s.region = '3' AND s.geom IS NOT NULL), intersected AS (SELECT r.fclass, ST_Intersection(r.geom, ts.geom) AS geom_clip FROM roads AS r INNER JOIN target_states AS ts ON ST_Intersects(r.geom, ts.geom) WHERE r.geom IS NOT NULL) SELECT i.fclass, SUM(ST_Length(i.geom_clip::geography)) / 1000.0 AS total_km FROM intersected AS i WHERE NOT ST_IsEmpty(i.geom_clip) GROUP BY i.fclass ORDER BY i.fclass;	✅	The System-generated query correctly clips roads to the states in region '3' using ST_Intersection and sums the lengths of the clipped segments, producing total kilometers of road within those states grouped by fclass. It also avoids counting non-overlapping portions outside the states and guards against empty geometries.	WITH target_states AS (SELECT s.geom FROM states AS s WHERE s.region = '3' AND s.geom IS NOT NULL), intersected AS (SELECT r.fclass, ST_Intersection(r.geom, ts.geom) AS geom_clip FROM roads AS r INNER JOIN target_states AS ts ON ST_Intersects(r.geom, ts.geom) WHERE r.geom IS NOT NULL) SELECT i.fclass, SUM(ST_Length(i.geom_clip::geography)) / 1000.0 AS total_km FROM intersected AS i WHERE NOT ST_IsEmpty(i.geom_clip) GROUP BY i.fclass ORDER BY i.fclass;	Reviewer didn't make any changes. System-generated queries: They clip roads to the states in region '3' (ST_Intersection), then sum the lengths of the clipped segments in kilometers, grouped by fclass. They also exclude empty geometries and use geography for accurate length.	✅

Use subquery to find the state with most POIs and list its counties with POI counts.	WITH max_poi_state AS (SELECT s.name FROM states s JOIN poi p ON ST_Intersects(s.geom, p.geom) GROUP BY s.name ORDER BY COUNT(p.gid) DESC LIMIT 1) SELECT c.name, COUNT(p.gid) FROM counties c JOIN poi p ON ST_Intersects(c.geom, p.geom) JOIN max_poi_state m ON ST_Intersects(c.geom, (SELECT geom FROM states WHERE name = m.name)) GROUP BY c.name;	WITH top_state AS (SELECT s.name AS state_name, s.geom AS state_geom FROM states s JOIN poi p ON ST_Within(p.geom, s.geom) GROUP BY s.name, s.geom ORDER BY COUNT(*) DESC, s.name ASC LIMIT 1) SELECT ts.state_name, c.geoid AS county_geoid, c.name AS county_name, COUNT(*) AS poi_count FROM top_state ts JOIN counties c ON ST_Intersects(c.geom, ts.state_geom) JOIN poi p ON ST_Within(p.geom, c.geom) GROUP BY ts.state_name, c.geoid, c.name ORDER BY poi_count DESC, county_name ASC;	✗	The System-generated query is incorrect because: It also selects counties via ST_Intersects with the state, so it can include counties from adjacent states that share a boundary line. Consequently, it may count POIs from outside the intended state.	WITH top_state AS (SELECT s.name AS state_name, s.geom AS state_geom FROM states s JOIN poi p ON ST_Within(p.geom, s.geom) GROUP BY s.name, s.geom ORDER BY COUNT(*) DESC, s.name ASC LIMIT 1) SELECT ts.state_name, c.geoid AS county_geoid, c.name AS county_name, COUNT(*) AS poi_count FROM top_state ts JOIN counties c ON ST_Intersects(c.geom, ts.state_geom) JOIN poi p ON ST_Within(p.geom, c.geom) GROUP BY ts.state_name, c.geoid, c.name ORDER BY poi_count DESC, county_name ASC;	Reviwer query is incorrect because it uses ST_Intersects to join counties to the top state.	✗

Calculate the total number of bridges (bridge='T') on roads within each protected area type.	SELECT pa.unit_type, COUNT(r.gid) FROM ne_protected_areas pa JOIN roads r ON ST_Within(pa.geom, r.geom) WHERE r.bridge = 'T' GROUP BY pa.unit_type;	WITH bridge_roads AS (SELECT r.geom FROM roads AS r WHERE r.bridge = 'T'), joined AS (SELECT n.unit_type FROM bridge_roads br INNER JOIN ne_protected_areas n ON ST_Intersects(br.geom, n.geom)) SELECT j.unit_type AS protected_area_type, COUNT(*) AS bridge_count FROM joined j GROUP BY j.unit_type ORDER BY j.unit_type;	❌	The unreviwed query is incorrect since it uses ST_Intersects and lacks DISTINCT, so it can overcount and does not enforce “within.”	SELECT n.unit_type AS protected_area_type, COUNT(DISTINCT r.gid) AS bridge_count FROM roads r JOIN ne_protected_areas n ON ST_Within(r.geom, n.geom) WHERE r.bridge = 'T' GROUP BY n.unit_type ORDER BY n.unit_type;	Reviwer is correct because it counts bridge road segments within protected areas (ST_Within), groups by protected area type, and uses COUNT(DISTINCT r.gid) to avoid double-counting when a segment lies within multiple polygons of the same type.	✅

Find the average value for SNOW element in GHCN per state, for dates in 2024.	SELECT s.name, AVG(g.value) FROM states s JOIN ghcn g ON ST_Intersects(s.geom, g.geom) WHERE g.element = 'SNOW' AND g.date >= 20240101 AND g.date < 20250101 GROUP BY s.name;	WITH filtered_ghcn AS (
		  SELECT
		    g.geom,
		    g.value
		  FROM ghcn AS g
		  WHERE g.element = 'SNOW'
		    AND g.date BETWEEN 20240101 AND 20241231
		),
		joined AS (
		  SELECT
		    s.name AS state_name,
		    f.value
		  FROM filtered_ghcn AS f
		  INNER JOIN states AS s
		    ON ST_Within(f.geom, s.geom)
		)
		SELECT
		  state_name,
		  AVG(value)::float AS avg_snow
		FROM joined
		GROUP BY state_name
		ORDER BY state_name;
			✅	Both queries correctly filter GHCN records to element = 'SNOW' and dates in 2024, spatially join station points to states, and compute the per-state average of values.	WITH filtered_ghcn AS (
					  SELECT
					    g.geom,
					    g.value
					  FROM ghcn AS g
					  WHERE g.element = 'SNOW'
					    AND g.date BETWEEN 20240101 AND 20241231
					),
					joined AS (
					  SELECT
					    s.name AS state_name,
					    f.value
					  FROM filtered_ghcn AS f
					  INNER JOIN states AS s
					    ON ST_Within(f.geom, s.geom)
					)
					SELECT
					  state_name,
					  AVG(value)::float AS avg_snow
					FROM joined
					GROUP BY state_name
					ORDER BY state_name;
						All three correctly filter SNOW observations to 2024, spatially join station points to states, and compute the per-state average.	✅

Use CTE to find time zones intersecting protected areas and count roads within them.	WITH tz_pa AS (SELECT tz.geom FROM ne_time_zones tz JOIN ne_protected_areas pa ON ST_Intersects(tz.geom, pa.geom)) SELECT COUNT(r.gid) FROM roads r JOIN tz_pa tp ON ST_Intersects(r.geom, tp.geom);	WITH tz_pa AS (SELECT tz.zone, ST_Intersection(tz.geom, pa.geom) AS geom FROM ne_time_zones AS tz INNER JOIN ne_protected_areas AS pa ON ST_Intersects(tz.geom, pa.geom)), tz_pa_union AS (SELECT zone, ST_UnaryUnion(ST_Collect(geom)) AS geom FROM tz_pa GROUP BY zone) SELECT u.zone AS time_zone, COUNT(r.geom) AS roads_count FROM tz_pa_union AS u LEFT JOIN roads AS r ON ST_Intersects(r.geom, u.geom) GROUP BY u.zone ORDER BY u.zone;	✅	The System-generated query correctly: Uses a CTE to identify the intersection geometry between time zones and protected areas.	WITH tz_pa AS (SELECT tz.zone, ST_Intersection(tz.geom, pa.geom) AS geom FROM ne_time_zones AS tz INNER JOIN ne_protected_areas AS pa ON ST_Intersects(tz.geom, pa.geom)), tz_pa_union AS (SELECT zone, ST_UnaryUnion(ST_Collect(geom)) AS geom FROM tz_pa GROUP BY zone) SELECT u.zone AS time_zone, COUNT(r.geom) AS roads_count FROM tz_pa_union AS u LEFT JOIN roads AS r ON ST_Intersects(r.geom, u.geom) GROUP BY u.zone ORDER BY u.zone;	Reviewer didn't make any changes so it is corrct like unreviwed query	✅

Group blockgroups by county and calculate the variance of their areas in square kilometers.	SELECT b.county, VARIANCE(ST_Area(b.geom::geography)/1000000) AS area_variance FROM blockgroups b GROUP BY b.county;	WITH blockgroups_in_counties AS (SELECT c.geoid AS county_geoid, c.name AS county_name, ST_Area(ST_Transform(b.geom, 5070)) / 1000000.0 AS area_sq_km FROM blockgroups b INNER JOIN counties c ON ST_Within(b.geom, c.geom)) SELECT county_geoid, county_name, VAR_SAMP(area_sq_km) AS area_variance_sq_km FROM blockgroups_in_counties GROUP BY county_geoid, county_name;	✅	The system-generated query correctly groups blockgroups by county via a spatial join, computes each blockgroup's area in square kilometers using an appropriate equal-area projection (EPSG:5070), and returns the sample variance per county.	WITH blockgroups_in_counties AS (SELECT c.geoid AS county_geoid, c.name AS county_name, ST_Area(ST_Transform(b.geom, 5070)) / 1000000.0 AS area_sq_km FROM blockgroups b INNER JOIN counties c ON ST_Within(b.geom, c.geom)) SELECT county_geoid, county_name, VAR_SAMP(area_sq_km) AS area_variance_sq_km FROM blockgroups_in_counties GROUP BY county_geoid, county_name;	Reviewer didn't make any changes. Reviwed and unreviwed queris correctly group blockgroups by county via a spatial join, compute areas in square kilometers using an equal-area projection (EPSG:5070), and use VAR_SAMP for variance by county.	✅

Find POIs within 10km of GHCN stations with TMAX > 80, and count per POI fclass.	SELECT p.fclass, COUNT(p.gid) FROM poi p JOIN (SELECT geom FROM ghcn WHERE element = 'TMAX' AND value > 80) g ON ST_DWithin(p.geom::geography, g.geom::geography, 10000) GROUP BY p.fclass;	SELECT
		  p.fclass,
		  COUNT(DISTINCT p.geom) AS poi_count
		FROM poi AS p
		JOIN ghcn AS g
		  ON ST_DWithin(g.geom::geography, p.geom::geography, 10000)
		WHERE g.element = 'TMAX'
		  AND g.value > 80
		GROUP BY p.fclass
		ORDER BY p.fclass;
			✅	The system-generated query is correct because it restricts to GHCN rows with TMAX > 80, finds POIs within 10 km, and counts unique POIs per fclass while avoiding duplicate counts from multiple nearby stations by using DISTINCT. (A small improvement would be COUNT(DISTINCT p.gid) instead of geometry.)	SELECT
					  p.fclass,
					  COUNT(DISTINCT p.geom) AS poi_count
					FROM poi AS p
					JOIN ghcn AS g
					  ON ST_DWithin(g.geom::geography, p.geom::geography, 10000)
					WHERE g.element = 'TMAX'
					  AND g.value > 80
					GROUP BY p.fclass
					ORDER BY p.fclass;
						Reviwed and unreviwed correctly:
						  Filter GHCN rows to TMAX > 80.
						  Use ST_DWithin with 10,000 meters to find POIs within 10 km.
						  Count unique POIs per fclass via COUNT(DISTINCT p.geom), preventing double-counting when a POI is near multiple qualifying stations/observations.
							✅

Appendix IV

Evaluation Results of Non-spatial Queries (KaggleDBQA)

WorldSoccer Database

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation	Annotator Comment
What are the Pinnacle odds for Arsenal winning matches?	SELECT PSH FROM football_data WHERE HomeTeam LIKE "%Arsenal%" AND FTR = "H" UNION SELECT PSA FROM football_data WHERE AwayTeam LIKE "%Arsenal%" AND FTR = "A"	SELECT CASE WHEN hometeam = 'Arsenal' THEN 'home' ELSE 'away' END AS team_side, CASE WHEN hometeam = 'Arsenal' THEN awayteam ELSE hometeam END AS opponent, CASE WHEN hometeam = 'Arsenal' THEN psh ELSE psa END AS pinnacle_win_odds, league, season FROM football_data WHERE (hometeam = 'Arsenal' AND psh IS NOT NULL) OR (awayteam = 'Arsenal' AND psa IS NOT NULL);	✅	Unreviewed query correctly returns Pinnacle odds for the “Arsenal to win” outcome in all Arsenal matches (PSH when home, PSA when away) and excludes null odds. The Benchmark-Proposed Query limits to matches Arsenal actually won (via FTR), which is not asked; it also uses LIKE (risking spurious matches), UNION (deduplicating odds), and doesn’t exclude null odds.	SELECT CASE WHEN hometeam = 'Arsenal' THEN 'home' ELSE 'away' END AS team_side, CASE WHEN hometeam = 'Arsenal' THEN awayteam ELSE hometeam END AS opponent, CASE WHEN hometeam = 'Arsenal' THEN psh ELSE psa END AS pinnacle_win_odds, league, season FROM football_data WHERE (hometeam = 'Arsenal' AND psh IS NOT NULL) OR (awayteam = 'Arsenal' AND psa IS NOT NULL);	Unreviewed query and The Reviewed Query are identical and correctly return the Pinnacle odds for an Arsenal win in each Arsenal match (PSH when Arsenal is home, PSA when away), regardless of the actual result.	✅	It is an example of unclear question and incorrect question in the benchmark

Which matches in Spain did the away team win?	SELECT * FROM football_data WHERE Country = "Spain" AND FTR = "A"	SELECT fd.hometeam AS home_team, fd.awayteam AS away_team FROM football_data AS fd WHERE fd.ftr = 'A' AND FALSE;	✗	The Benchmark-Proposed Query correctly filters matches where Country = 'Spain' and FTR = 'A' (away win), returning the Spanish matches the away team won. Unreviewed query uses WHERE ftr = 'A' AND FALSE, which always yields no rows, and it also lacks a filter for Spain.	SELECT * FROM football_data WHERE Country = 'Spain' OR FTR = 'A';	The Reviewed Query is incorrect: This query is incorrect because it uses OR instead of AND.	✗	

Which Premier League matches ended in a draw in 2016?	SELECT * FROM football_data WHERE Season = "2016" AND FTR = "D" AND League = "Premier League"	SELECT
		  hometeam AS home_team,
		  awayteam AS away_team,
		  season
		FROM football_data
		WHERE league = 'Premier League'
		  AND ftr = 'D'
		  AND (
		    season = '2016'
		    OR season LIKE '2015/2016'
		    OR season LIKE '2016/2017'
		  );
			✅	Unreviewed query correctly targets Premier League draws and includes both seasons that span the calendar year 2016 (2015/2016 and 2016/2017), also accounting for a possible '2016' season label, thus encompassing matches that occurred in 2016.
				The Benchmark-Proposed Query filters only Season = "2016", which typically does not match season labels like '2015/2016' or '2016/2017', leading to no or incomplete results; it would miss draws from early 2016 that belong to the 2015/2016 season.
					SELECT
					  datetime,
					  hometeam AS home_team,
					  awayteam AS away_team
					FROM football_data
					WHERE country = 'England'
					  AND league = 'Premier League'
					  AND ftr = 'D'
					  AND datetime >= '2016-01-01'
					  AND datetime < '2017-01-01'
					ORDER BY datetime;
						The Reviewed Query correctly interprets "in 2016" as the calendar year, filters Premier League draws (FTR = 'D') within 2016-01-01 to 2017-01-01, and optionally restricts to England, which aligns with the Premier League.	✅	

Which matches had draw odds from Bet365 higher than 3?	SELECT * FROM football_data WHERE B365D > 3	SELECT
		  hometeam AS home_team,
		  awayteam AS away_team,
		  league,
		  b365d AS bet365_draw_odds
		FROM football_data
		WHERE b365d > 3;
			✅	Both filter on B365D > 3, matching “draw odds from Bet365 higher than 3.” Unreviewed query also selects home/away teams (and league) plus the odds, directly identifying the matches.	SELECT
					  hometeam AS home_team,
					  awayteam AS away_team,
					  league,
					  b365d AS
					bet365_draw_odds
					FROM football_data
					WHERE b365d > 3;
						All three correctly filter matches where Bet365 draw odds (B365D/b365d) are > 3.	✅	

What were the closing odds for a draw in matches with VfB Stuttgart?	SELECT DRAW_CLOSING FROM betfront WHERE MATCH LIKE "%VfB Stuttgart%"	SELECT
		  b.draw_closing,
		  b.match
		FROM betfront AS b
		WHERE b.match ILIKE '%VfB Stuttgart%';
			✅	Unreviewed query correctly filters matches containing "VfB Stuttgart" (case-insensitive via ILIKE) and returns the draw_closing values, which are the closing odds for a draw.	SELECT
					  b.draw_closing,
					  b.match
					FROM betfront AS b
					WHERE b.match ILIKE '%VfB Stuttgart%';
						Reviewer didn't make any changes. They are identical and correctly filter matches containing “VfB Stuttgart” (case-insensitive via ILIKE) and return the draw_closing values (with match included, which is acceptable).	✅	

Which games had no goals scored at full time?	SELECT * FROM football_data WHERE (FTHG + FTAG) = 0	SELECT
		  hometeam AS home_team,
		  awayteam AS away_team,
		  '0-0' AS score
		FROM football_data
		WHERE fthg = 0
		  AND ftag = 0;
			✅	Unreviewed query is correct because it explicitly filters matches where both FTHG = 0 and FTAG = 0, i.e., 0–0 at full time, and returns the teams with the score.	SELECT
					  hometeam AS home_team,
					  awayteam AS away_team,
					  '0-0' AS score
					FROM football_data
					WHERE fthg = 0
					  AND ftag = 0;
						Reviewer didn't make any changes.
						All three correctly identify 0-0 full-time matches: Unreviewed query by filtering rows where FTHG + FTAG = 0; Queries B and C by explicitly requiring fthg = 0 and ftag = 0 and listing the teams.
							✅	

What is the away team against Omiya Ardija in 2018?	SELECT AwayTeam FROM football_data WHERE HomeTeam = "Omiya Ardija" AND Season LIKE "%2018%"	WITH filtered_seasons AS (
		  SELECT
		    awayteam
		  FROM football_data
		  WHERE hometeam = 'Omiya Ardija'
		    AND (
		      season = '2018'
		      OR season LIKE '2018/%'
		      OR season LIKE '%/2018'
		    )
		)
		SELECT DISTINCT awayteam AS away_team
		FROM filtered_seasons;
			✅	Unreviewed query correctly filters matches where Omiya Ardija is the home team and restricts Season to 2018 or seasons spanning 2018 (e.g., 2017/2018, 2018/2019). It also returns a clean distinct list and uses proper single-quoted string literals.
				The Benchmark-Proposed Query uses double-quoted string literals (invalid in many SQL dialects), may overmatch Season with LIKE "%2018%" and can return duplicates, so it's less accurate and potentially syntactically incorrect.
					WITH filtered_seasons AS (
					  SELECT
					    awayteam
					  FROM football_data
					  WHERE hometeam = 'Omiya Ardija'
					    AND (
					      season = '2018'
					      OR season LIKE '2018/%'
					      OR season LIKE '%/2018'
					    )
					)
					SELECT DISTINCT awayteam AS away_team
					FROM filtered_seasons;
						Reviewer didn't make any changes. All three queries correctly fetch the away team(s) when Omiya Ardija is the home team in seasons that include 2018.	✅	

How many matches in Spain in 2010?	SELECT count(*) FROM football_data WHERE Season LIKE "%2010%" AND Country = "Spain"	SELECT COUNT(*) AS match_count FROM betfront WHERE LOWER(TRIM(country)) = 'spain' AND year = 2010;	✅	Unreviewed query directly filters by country = Spain (case/whitespace normalized) and by calendar year = 2010, which aligns with “matches in Spain in 2010.” The Benchmark-Proposed Query uses Season LIKE "%2010%", which will match both “2009/2010” and “2010/2011” seasons and include matches from 2009 and 2011, overcounting. It also relies on exact casing of Country and ignores potential whitespace inconsistencies.	SELECT COUNT(*) AS match_count FROM betfront WHERE LOWER(TRIM(country)) = 'spain' AND year = 2010;	Reviewer didn't make any changes. They correctly count matches in Spain during calendar year 2010 by filtering betfront on country = 'spain' (case/space-normalized) and year = 2010.	✅	

Which matches has the highest draw opening so far?	SELECT MATCH FROM betfront ORDER BY DRAW_OPENING DESC LIMIT 1	WITH max_draw AS (
		  SELECT MAX(draw_opening) AS max_draw_opening
		  FROM betfront
		)
		SELECT
		  b.match AS match,
		  b.draw_opening AS draw_opening
		FROM betfront b
		JOIN max_draw m
		  ON b.draw_opening = m.max_draw_opening;
			✅	Unreviewed query correctly identifies the maximum draw_opening and returns all matches that have that maximum, covering ties and ignoring NULLs.
				The Benchmark-Proposed Query returns only one match due to LIMIT 1 (missing ties) and can be incorrect in some DBs where ORDER BY ... DESC puts NULLs first; it may also require quoting MATCH if it's a reserved word.
					WITH max_draw AS (
					  SELECT MAX(draw_opening) AS max_draw_opening
					  FROM betfront
					)
					SELECT
					  b.match AS match,
					  b.draw_opening AS draw_opening
					FROM betfront b
					JOIN max_draw m
					  ON b.draw_opening = m.max_draw_opening;
						Reviewer didn't make any changes.
						Unreviewed query and The Reviewed Query correctly return all matches with the maximum draw_opening (handling ties), matching "Which matches has the highest draw opening".
							✅	

Which year has most matches?	SELECT YEAR FROM betfront GROUP BY YEAR ORDER BY count(*) DESC LIMIT 1	SELECT year, COUNT(*) AS match_count FROM betfront GROUP BY year ORDER BY match_count DESC, year ASC LIMIT 1;	✅	Both queries group by the year column, order by the number of matches descending, and limit to one row, returning the year with the most matches. Unreviewed query also returns the count and adds a deterministic tie-breaker (smallest year on ties)	SELECT year, COUNT(*) AS match_count FROM betfront GROUP BY year ORDER BY match_count DESC, year ASC LIMIT 1;	The Reviewed Query is correct same as unreviwed.	✅	

How many matches did Pinnacle have betting records?	SELECT count(*) FROM football_data WHERE PSH != "" AND PSD != "" AND PSA != ""	WITH bf AS (
		  SELECT row_number() OVER (ORDER BY (SELECT 1)) AS rn, match
		  FROM betfront
		  WHERE match IS NOT NULL
		),
		fd AS (
		  SELECT row_number() OVER (ORDER BY (SELECT 1)) AS rn, psh, psd, psa
		  FROM football_data
		)
		SELECT COUNT(*) AS count_matches_with_pinnacle
		FROM bf
		LEFT JOIN fd USING (rn)
		WHERE fd.psh IS NOT NULL OR fd.psd IS NOT NULL OR fd.psa IS NOT NULL;
			✗	The Benchmark-Proposed Query directly counts matches in football_data where all three Pinnacle odds fields (PSH, PSD, PSA) are populated, which aligns with “have betting records” (i.e., a complete Pinnacle odds set per match).
				Unreviewed query is incorrect because it fabricates a join between betfront and football_data using row_number() rather than a real match key, making the count dependent on arbitrary row order. It also counts rows when only one of PSH/PSD/PSA is present (OR), which doesn’t reliably represent a full Pinnacle record.
					WITH bf AS (
					  SELECT row_number() OVER (ORDER BY (SELECT 1)) AS rn, match
					  FROM betfront
					  WHERE match IS NOT NULL
					),
					fd AS (
					  SELECT row_number() OVER (ORDER BY (SELECT 1)) AS rn, psh, psd, psa
					  FROM football_data
					)
					SELECT COUNT(*) AS count_matches_with_pinnacle
					FROM bf
					LEFT JOIN fd USING (rn)
					WHERE fd.psh IS NOT NULL OR fd.psd IS NOT NULL OR fd.psa IS NOT NULL;
						The Reviewed Query is the same as unreviwed query. So, it's not correct.	✗	

How many matches did Bet365 gives higher home win odds than Pinnacle?	SELECT count(*) FROM football_data WHERE B365H > PSH	SELECT COUNT(*) AS count_matches FROM football_data WHERE b365h IS NOT NULL AND psh IS NOT NULL AND b365h > psh;	✅	Unreviewed query correctly counts matches where Bet365's home odds (B365H) are greater than Pinnacle's (PSH), explicitly excluding NULLs.	SELECT COUNT(*) AS count_matches FROM football_data WHERE b365h IS NOT NULL AND psh IS NOT NULL AND b365h > psh;	Reviewer didn't make any changes. All three correctly count matches where Bet365 home odds (B365H) are greater than Pinnacle home odds (PSH). Queries B and C add explicit NULL checks, which are redundant because the comparison excludes NULLs anyway. None of the queries are incorrect.	✅	
How many games that the total number of goals exceed 5?	SELECT count(*) FROM football_data WHERE FTHG + FTAG > 5	SELECT COUNT(*) AS games_over_5_goals FROM football_data WHERE (fthg + ftag) > 5;	✅	Both queries count matches where total goals (FTHG + FTAG) exceed 5, matching the intent of "exceed 5" (> 5). Unreviewed query's alias and lowercase identifiers don't affect correctness.	SELECT COUNT(*) AS games_over_5_goals FROM football_data WHERE (FTHG + FTAG) > 5;	Reviewer didn't make any changes. All three correctly count games where total goals (FTHG + FTAG) > 5; differences are only aliasing/case. None are incorrect.	✅	

What is the highest home losing odds in Bet365 ever?	SELECT max(B365A) FROM football_data	SELECT MAX(b365h) AS highest_home_losing_odds FROM football_data WHERE ftr = 'A' AND b365h IS NOT NULL;	❌	"Home losing" corresponds to the away win market; Bet365's away-odds column is B365A. The Benchmark-Proposed Query returns MAX(B365A), i.e., the highest home-losing (away win) odds ever offered. Unreviewed query takes MAX(B365H) on matches where the home actually lost (FTR = 'A'), which is the home-win market, not the home-losing market. Even with the filter, it targets the wrong column.	SELECT MAX(b365a) AS highest_home_losing_odds FROM football_data;	Reviwer take the maximum of Bet365's away-win odds (b365a), which represent the odds of the home team losing; "ever" implies across the whole dataset. Unreviewed query uses b365h (home-win odds) and restricts to matches the home team lost (ftr = 'A'), which answers "highest home-win odds in games the home team lost," not "highest home losing odds."	✅	

How many number of games ended in a 0-0 tie?	SELECT count(*) FROM football_data WHERE FTHG = 0 AND FTAG = 0	SELECT COUNT(*) AS zero_zero_ties
		FROM football_data
		WHERE fthg = 0
		  AND ftag = 0;
			✅	Benchamrk and unreviwed queries count rows in football_data where the full-time home and away goals are 0 (FTHG = 0 AND FTAG = 0), which exactly returns the number of 0-0 ties. The only differences are casing and an alias in Unreviewed query; these do not change the result.	SELECT COUNT(*) AS zero_zero_ties
					FROM football_data
					WHERE fthg = 0
					  AND ftag = 0;
						All three queries correctly count matches where both FTHG and FTAG are 0, which is exactly the number of 0-0 ties.
						Alias/case differences don’t affect correctness.
						None are incorrect; Unreviewed query and The Reviewed Query are identical. Reviewer didn't make any changes.
							✅		

How many league division does football_data database has?	SELECT count(Div) FROM football_data	SELECT COUNT(DISTINCT div) AS division_count FROM football_data;	✅	Unreviewed query correctly counts the number of unique league divisions using COUNT(DISTINCT div), which matches the intent of "how many league divisions." The Benchmark-Proposed Query counts the number of rows with a non-null Div value, not the number of distinct divisions, so it overcounts when multiple rows exist per division.	SELECT COUNT(DISTINCT div) AS division_count FROM football_data;	Reviewer didn't make any changes They correctly return the number of unique league divisions using COUNT(DISTINCT div).	✅	

Do other leagues have referee name records outside of Scotland and England?	SELECT count(League) FROM football_data WHERE Country != "Scotland" and Country != "England" and Referee != ""	with referee_rows as (select league, referee from football_data where referee is not null and trim(referee) <> ''), non_uk_leagues as (select distinct league from referee_rows where league is not null and league not ilike any (array['%England%', '%English%', '%Premier League%', '%Championship%', '%League 1%', '%League One%', '%League Two%', '%National League%', '%FA%', '%EFL%', '%Scottish%', '%Scotland%', '%SPFL%', '%Scottish Premiership%', '%Scottish Championship%', '%Scottish League One%', '%Scottish League Two%'])), answer as (select case when exists (select 1 from non_uk_leagues) then 'Yes' else 'No' end as has_ref_outside_scot_eng) select a.has_ref_outside_scot_eng, l.league as league_name from answer a left join non_uk_leagues l on a.has_ref_outside_scot_eng = 'Yes' order by l.league nulls last;	❌	The Benchmark-Proposed Query is correct because it directly checks for records where Country is neither Scotland nor England and a referee name is present (non-empty), which answers the yes/no existence question. Unreviewed query is incorrect because it infers country from league-name patterns rather than using the Country field. Its filters (e.g., '%Premier League%', '%FA%') will wrongly exclude many non-UK leagues (e.g., Russian Premier League) and can misclassify leagues, leading to false negatives.	select case when exists (select 1 from football_data where referee is not null and trim(referee) <> '' and country is not null and lower(country) not in ('england', 'scotland')) then 'Yes' else 'No' end as has_ref_outside_scot_eng;	The Reviewed Query is correct because it directly answers the yes/no question by checking for the existence of any rows with non-null, non-blank referee names where country is not England or Scotland, and returns a clear Yes/No result.	✅	

What's the odds for draw on Bet365 for the game Swindon v.s. Millwall for 2016/2017 season?	SELECT B365D FROM football_data WHERE HomeTeam = "Swindon" and AwayTeam = "Millwall" and Season = "2016/2017"	SELECT DISTINCT b365d AS bet365_draw_odds FROM football_data WHERE hometeam = 'Swindon' AND awayteam = 'Millwall' AND season = '2016/2017' AND b365d IS NOT NULL;	✅	It selects the Bet365 draw odds (B365D) for the exact Swindon (home) vs Millwall (away) fixture in 2016/2017, uses proper string literals, and filters out NULLs; DISTINCT avoids duplicate rows.	SELECT DISTINCT b365d AS bet365_draw_odds FROM football_data WHERE hometeam = 'Swindon' AND awayteam = 'Millwall' AND season = '2016/2017' AND b365d IS NOT NULL;	Reviewer didn't make any changes. They correctly select the Bet365 draw odds (b365d) for the specific match (Swindon home vs Millwall away) in the 2016/2017 season, use proper string literals, and exclude NULLs; DISTINCT is harmless.	✅	

Pesticide Database

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation	Annotator Comment
Provide the list of products with pesticide residues detected.	SELECT DISTINCT commod FROM resultsdata15 WHERE concen != ""	WITH detected AS (SELECT commtype, commod FROM sampledata15 WHERE concen IS NOT NULL AND ((lod IS NOT NULL AND concen >= lod) OR (lod IS NULL AND quantitate IS NOT NULL AND concen >= NULLIF(quantitate::double precision, 'NaN'::double precision)))) SELECT DISTINCT commtype AS product_type, commod AS product_name FROM detected WHERE commtype IS NOT NULL OR commod IS NOT NULL ORDER BY product_type NULLS LAST, product_name NULLS LAST;	✅	Unreviewed query is incorrect because it pulls from sampledata15, which is not the per-analyte results table needed to determine detections; the detection logic belongs on resultsdata15.	WITH detected AS (SELECT commtype, commod FROM resultsdata15 WHERE concen IS NOT NULL AND ((lod IS NOT NULL AND concen >= lod) OR (lod IS NULL AND quantitate IS NOT NULL AND concen >= NULLIF(quantitate::double precision, 'NaN'::double precision)))) SELECT DISTINCT commtype AS product_type, commod AS product_name FROM detected WHERE commtype IS NOT NULL OR commod IS NOT NULL ORDER BY product_type NULLS LAST, product_name NULLS LAST;	Reviewed query correctly identifies detections by requiring a non-null concentration that meets or exceeds LOD (or LOQ when LOD is missing), and returns a distinct list of products from the appropriate results table (resultsdata15).	✅	

What is the most effective extraction method in terms of pesticide residues detection?	SELECT extract FROM resultsdata15 GROUP BY extract ORDER BY count(*) DESC LIMIT 1	WITH base AS (SELECT r.extract AS extraction_method, r.concen, r.lod, r.pestcode, r.commod, CASE WHEN r.concen IS NOT NULL AND r.lod IS NOT NULL AND r.concen >= r.lod THEN 1 ELSE 0 END AS detect_flag FROM resultsdata15 r WHERE r.lod IS NOT NULL), agg AS (SELECT extraction_method, COUNT(*) AS total_tests, SUM(detect_flag) AS detects, CASE WHEN COUNT(*) > 0 THEN SUM(detect_flag)::decimal / COUNT(*) ELSE NULL END AS detection_rate, PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY lod) AS median_lod, PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY concen) FILTER (WHERE detect_flag = 1) AS median_detect_concen, COUNT(DISTINCT pestcode) FILTER (WHERE detect_flag = 1) AS unique_pesticides_detected, COUNT(DISTINCT commod) AS commodities_covered FROM base GROUP BY extraction_method), ranked AS (SELECT *, ROW_NUMBER() OVER (ORDER BY detection_rate DESC NULLS LAST, median_lod ASC NULLS LAST, unique_pesticides_detected DESC NULLS LAST) AS rn FROM agg) SELECT extraction_method, total_tests, detects, detection_rate, median_lod, median_detect_concen, unique_pesticides_detected,	✅	Unreviewed query operationalizes “effectiveness in terms of detection” by computing per-method detection rate (concentration ≥ LOD), and sensibly breaks ties using sensitivity (lower median LOD) and breadth (unique pesticides detected). It returns the extraction method with the highest detection performance. Benchamrk query ranks methods by number of tests only, ignoring whether pesticides were detected or the sensitivity of the method. It answers “most used,” not “most effective for detection.”	WITH base AS (SELECT r.extract AS extraction_method, r.concen, r.lod, r.pestcode, r.commod, CASE WHEN r.concen IS NOT NULL AND r.lod IS NOT NULL AND r.concen >= r.lod THEN 1 ELSE 0 END AS detect_flag FROM resultsdata15 r WHERE r.lod IS NOT NULL), agg AS (SELECT extraction_method, COUNT(*) AS total_tests, SUM(detect_flag) AS detects, CASE WHEN COUNT(*) > 0 THEN SUM(detect_flag)::decimal / COUNT(*) ELSE NULL END AS detection_rate, PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY lod) AS median_lod, PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY concen) FILTER (WHERE detect_flag = 1) AS median_detect_concen, COUNT(DISTINCT pestcode) FILTER (WHERE detect_flag = 1) AS unique_pesticides_detected, COUNT(DISTINCT commod) AS commodities_covered FROM base GROUP BY extraction_method), ranked AS (SELECT *, ROW_NUMBER() OVER (	Reviewer didn't make any changes. They correctly measure detection effectiveness by computing detection_rate per extraction method (based on concen >= lod), then ranking by highest detection_rate with tie-breakers (lower median LOD, more unique pesticides detected) to select the top method.	✅	

		commodities_covered FROM ranked WHERE rn = 1;			ORDER BY detection_rate DESC NULLS LAST, median_lod ASC NULLS LAST, unique_pesticides_detected DESC NULLS LAST) AS rn FROM agg) SELECT extraction_method, total_tests, detects, detection_rate, median_lod, median_detect_concen, unique_pesticides_detected, commodities_covered FROM ranked WHERE rn = 1;				

What is the most effective determinative method in terms of pesticide residues detection?	SELECT determin FROM resultsdata15 GROUP BY determin ORDER BY count(*) DESC LIMIT 1	WITH valid_rows AS (SELECT determin, concen, lod, CASE WHEN concen >= lod THEN 1 ELSE 0 END AS detect_flag FROM resultsdata15 WHERE concen IS NOT NULL AND lod IS NOT NULL AND determin IS NOT NULL AND trim(determin) <> ''), method_stats AS (SELECT determin, COUNT(*) AS total_tests, SUM(detect_flag) AS detections, CASE WHEN COUNT(*) > 0 THEN SUM(detect_flag)::decimal / COUNT(*) ELSE 0 END AS detection_rate FROM valid_rows GROUP BY determin), ranked AS (SELECT determin AS most_effective_method, total_tests, detections, detection_rate, ROW_NUMBER() OVER (ORDER BY detection_rate DESC, detections DESC, total_tests DESC, determin) AS rn FROM method_stats) SELECT most_effective_method FROM ranked WHERE rn = 1;	✅	Unreviewed query correctly operationalizes “effectiveness in terms of detection” by computing, per determinative method, the detection rate based on concen >= lod, then selecting the highest rate (with sensible tie-breakers). This directly measures how often each method detects residues. Benchamrk query is incorrect because it returns the most frequently used method (highest row count), not the most effective at detecting residues. It ignores concentrations and LOD entirely.	WITH valid_rows AS (SELECT determin, concen, lod, CASE WHEN concen >= lod THEN 1 ELSE 0 END AS detect_flag FROM resultsdata15 WHERE concen IS NOT NULL AND lod IS NOT NULL AND determin IS NOT NULL AND trim(determin) <> ''), method_stats AS (SELECT determin, COUNT(*) AS total_tests, SUM(detect_flag) AS detections, CASE WHEN COUNT(*) > 0 THEN SUM(detect_flag)::decimal / COUNT(*) ELSE 0 END AS detection_rate FROM valid_rows GROUP BY determin), ranked AS (SELECT determin AS most_effective_method, total_tests, detections, detection_rate, ROW_NUMBER() OVER (ORDER BY detection_rate DESC, detections DESC, total_tests DESC, determin) AS rn FROM method_stats) SELECT most_effective_method FROM ranked WHERE rn = 1;	Unreviewed query and reviewed query correctly compute each method’s detection rate (concen >= lod), rank by detection effectiveness, and return the top method while excluding invalid rows. Reviewer didn't make any changes.	✅	

What are the top 5 analyzing labs where pesticide residues detection was conducted?	SELECT lab
	FROM resultsdata15
	GROUP BY lab
	ORDER BY count(*) DESC
	LIMIT 5
		WITH detections AS (
		  SELECT
		    lab
		  FROM resultsdata15
		  WHERE concen IS NOT NULL
		    AND lod IS NOT NULL
		    AND concen >= lod
		),
		counts AS (
		  SELECT
		    lab AS analyzing_lab,
		    COUNT(*) AS detections_count
		  FROM detections
		  GROUP BY lab
		),
		ranked AS (
		  SELECT
		    analyzing_lab,
		    detections_count,
		    ROW_NUMBER() OVER (ORDER BY detections_count DESC, analyzing_lab NULLS LAST) AS rn
		  FROM counts
		)
		SELECT
		  analyzing_lab,
		  detections_count
		FROM ranked
		WHERE rn <= 5
		ORDER BY detections_count DESC, analyzing_lab NULLS LAST;
			✅	Unreviewed query filters to actual detections (concen >= lod), counts detections per analyzing lab, and returns the top 5 by detection count—matching the intent of "labs where pesticide residues detection was conducted" (i.e., detected).
				Benchamrk query ranks labs by total number of results (analyses) irrespective of whether any detection occurred, so it answers “labs with most tests,” not “labs with most detections.”
					WITH detections AS (
					  SELECT
					    lab
					  FROM resultsdata15
					  WHERE concen IS NOT NULL
					    AND lod IS NOT NULL
					    AND concen >= lod
					),
					counts AS (
					  SELECT
					    lab AS analyzing_lab,
					    COUNT(*) AS detections_count
					  FROM detections
					  GROUP BY lab
					),
					ranked AS (
					  SELECT
					    analyzing_lab,
					    detections_count,
					    ROW_NUMBER() OVER (ORDER BY detections_count DESC, analyzing_lab NULLS LAST) AS rn
					  FROM counts
					)
					SELECT
					  analyzing_lab,
					  detections_count
					FROM ranked
					WHERE rn <= 5
					ORDER BY detections_count DESC, analyzing_lab NULLS LAST;
						Reviewer didn't make any changes.
						They correctly count detection events per lab (concen >= lod), then return the top 5 labs by number of detections.
							✅		

Wha commodity type of apple has the highest pesticide residues to consume?	SELECT commtype FROM resultsdata15 WHERE commod = "AP" GROUP BY commtype ORDER BY sum(concen) DESC LIMIT 1	WITH apple_results AS (SELECT r.commtype, r.mean::numeric AS mean_value FROM resultsdata15 r LEFT JOIN sampledata15 s ON r.sample_pk = s.sample_pk WHERE r.mean IS NOT NULL AND (r.commod ILIKE '%apple%' OR s.variety ILIKE '%apple%') AND (r.testclass ILIKE '%pesticide%' OR r.testclass ILIKE '%residue%')), agg AS (SELECT commtype, MAX(mean_value) AS max_mean_residue, COUNT(*) AS pest_count FROM apple_results GROUP BY commtype) SELECT commtype AS commodity_type, max_mean_residue, pest_count FROM agg WHERE commtype IS NOT NULL ORDER BY max_mean_residue DESC LIMIT 1;	✅	Unreviewed query correctly isolates apple-related records (via commod/variety), restricts to pesticide/residue tests, and identifies the commodity type whose highest measured pesticide level (max of mean) is greatest, matching the intent of “highest pesticide residues.”	WITH apple_results AS (SELECT COALESCE(r.commtype, s.commtype) AS commtype, CAST(r.mean AS NUMERIC) AS mean_value FROM resultsdata15 r LEFT JOIN sampledata15 s ON r.sample_pk = s.sample_pk WHERE r.mean IS NOT NULL AND (r.commod ILIKE '%apple%' OR s.variety ILIKE '%apple%' OR s.commod ILIKE '%apple%') AND (r.testclass ILIKE '%pesticide%' OR r.testclass ILIKE '%residue%')), agg AS (SELECT commtype, SUM(mean_value) AS total_mean_residue, COUNT(*) AS pest_count FROM apple_results WHERE commtype IS NOT NULL GROUP BY commtype) SELECT commtype AS commodity_type, total_mean_residue, pest_count FROM agg ORDER BY total_mean_residue DESC LIMIT 1;	It targets apple-related records, filters to pesticide/residue tests, and aggregates across all pesticide measurements by commodity type using SUM(mean), which aligns with “highest pesticide residues” (plural) as an overall burden. It also safeguards commtype with COALESCE and returns the top commodity type.	✅	

Rank the country of product origins in terms of pesticide residues detection.	SELECT T1.country FROM sampledata15 as T1 JOIN resultsdata15 as T2 ON T1.sample_pk = T2.sample_pk GROUP BY T1.country ORDER BY T2.concen	WITH joined AS (
		  SELECT r.sample_pk, s.country, r.concen, r.lod
		  FROM resultsdata15 r
		  INNER JOIN sampledata15 s ON r.sample_pk = s.sample_pk
		),
		sample_detection AS (
		  SELECT
		    sample_pk,
		    country,
		    CASE WHEN BOOL_OR(concen IS NOT NULL AND lod IS NOT NULL AND concen >= lod) THEN 1 ELSE 0 END AS sample_detected
		  FROM joined
		  GROUP BY sample_pk, country
		),
		country_stats AS (
		  SELECT
		    country,
		    COUNT(DISTINCT sample_pk) AS total_samples,
		    SUM(sample_detected) AS detected_count,
		    CASE WHEN COUNT(DISTINCT sample_pk) > 0 THEN SUM(sample_detected)::decimal / COUNT(DISTINCT sample_pk) ELSE NULL END AS detection_rate
		  FROM sample_detection
		  GROUP BY country
		)
		SELECT
		  country,
		  detected_count,
		  total_samples,
		  detection_rate,
		  RANK() OVER (ORDER BY detection_rate DESC NULLS LAST, detected_count DESC) AS rank
		FROM country_stats
		ORDER BY rank, country NULLS LAST;
			✅	Unreviewed query correctly joins samples to results, defines “detection” as concen >= lod at the sample level, aggregates to country, computes detection rate (and counts), and ranks countries accordingly—matching the intent to rank origins by pesticide residue detection. Benchamrk query neither applies a detection criterion nor aggregates detections; it groups by country but orders by a non-aggregated T2.concen (invalid in most SQL dialects), and provides no rank or meaningful detection metric.	WITH joined AS (
					  SELECT r.sample_pk, s.country, r.concen, r.lod
					  FROM resultsdata15 r
					  INNER JOIN sampledata15 s ON r.sample_pk = s.sample_pk
					),
					sample_detection AS (
					  SELECT
					    sample_pk,
					    country,
					    CASE WHEN BOOL_OR(concen IS NOT NULL AND lod IS NOT NULL AND concen >= lod) THEN 1 ELSE 0 END AS sample_detected
					  FROM joined
					  GROUP BY sample_pk, country
					),
					country_stats AS (
					  SELECT
					    country,
					    COUNT(DISTINCT sample_pk) AS total_samples,
					    SUM(sample_detected) AS detected_count,
					    CASE WHEN COUNT(DISTINCT sample_pk) > 0 THEN SUM(sample_detected)::decimal / COUNT(DISTINCT sample_pk) ELSE NULL END AS detection_rate
					  FROM sample_detection
					  GROUP BY country
					)
					SELECT
					  country,
					  detected_count,
					  total_samples,
					  detection_rate,
					  RANK() OVER (ORDER BY detection_rate DESC NULLS LAST, detected_count DESC) AS rank
					FROM country_stats
					ORDER BY rank, country NULLS LAST;
						Reviewer didn't make any changes.
						Unreviewed query and reviewed query correctly: join samples to results, define detection at the sample level (any analyte concen >= LOD), aggregate to country-level totals, compute detection rate, and rank countries by detection rate (ties by detected count). This directly ranks countries “in terms of pesticide residues detection.”
							✅	

What type of collection facility results in the most risky consumption in terms of pesticide residues detection?	SELECT T1.disttype FROM sampledata15 as T1 JOIN resultsdata15 as T2 ON T1.sample_pk = T2.sample_pk GROUP BY T1.disttype ORDER BY sum(T2.concen) DESC LIMIT 1	WITH joined AS (SELECT s.disttype, r.sample_pk, r.concen, r.lod FROM resultsdata15 r INNER JOIN sampledata15 s ON r.sample_pk = s.sample_pk), clean AS (SELECT disttype, concen, lod FROM joined WHERE disttype IS NOT NULL AND concen IS NOT NULL AND lod IS NOT NULL), agg AS (SELECT disttype AS facility_type, COUNT(*) AS total_tests, COUNT(*) FILTER (WHERE concen >= lod) AS detections, CASE WHEN COUNT(*) > 0 THEN COUNT(*) FILTER (WHERE concen >= lod)::decimal / COUNT(*) ELSE 0 END AS detection_rate FROM clean GROUP BY disttype) SELECT facility_type, detections, total_tests, detection_rate FROM agg ORDER BY detection_rate DESC, detections DESC, facility_type LIMIT 1;	✅	Unreviewed query is correct because it measures detection risk by computing, per facility type (disttype), the proportion of tests where pesticide concentration meets/exceeds the limit of detection (concen >= lod). It then selects the facility type with the highest detection rate, which aligns with “most risky consumption in terms of pesticide residues detection.”	WITH sample_detection AS (SELECT r.sample_pk, MAX(CASE WHEN r.concen IS NOT NULL AND r.lod IS NOT NULL AND r.concen >= r.lod THEN 1 ELSE 0 END) AS detected FROM resultsdata15 r GROUP BY r.sample_pk), joined AS (SELECT s.disttype, sd.detected FROM sampledata15 s INNER JOIN sample_detection sd ON sd.sample_pk = s.sample_pk WHERE s.disttype IS NOT NULL) SELECT disttype AS facility_type, SUM(detected) AS samples_with_detection, COUNT(*) AS total_samples, CASE WHEN COUNT(*) > 0 THEN SUM(detected)::decimal / COUNT(*) ELSE 0 END AS detection_rate FROM joined GROUP BY disttype ORDER BY detection_rate DESC, samples_with_detection DESC, facility_type LIMIT 1;	reviewed query is correct because it measures consumer risk at the sample level: a sample is “risky” if any analyte is detected (concen >= lod). It then compares detection rates across facility types and returns the one with the highest probability a consumer encounters a detected residue.	✅	

What are the country of product origins where pesticide residues were not detected?	SELECT T1.country FROM sampledata15 as T1 JOIN resultsdata15 as T2 ON T1.sample_pk = T2.sample_pk WHERE T2.concen = "0" GROUP BY T1.country	WITH non_detects AS (
		  SELECT
		    r.sample_pk,
		    s.country
		  FROM resultsdata15 AS r
		  INNER JOIN sampledata15 AS s
		    ON r.sample_pk = s.sample_pk
		  WHERE r.concen < r.lod
		)
		SELECT DISTINCT country
		FROM non_detects
		WHERE country IS NOT NULL;
			✗	Unreviewed query is incorrect because it defines “not detected” as concentrations below the limit of detection (concen < lod), which aligns with standard analytical practice, joins to get the country, and returns distinct non-null countries.	WITH sample_status AS (
					  SELECT r.sample_pk, r.country,
					         SUM(CASE WHEN r.concen >= r.lod THEN 1 ELSE 0 END) AS detects
					  FROM resultsdata15 r
					  GROUP BY r.sample_pk, r.country
					)
					SELECT DISTINCT country
					FROM sample_status
					WHERE detects = 0 AND country IS NOT NULL;
						Reviewed query is incorrect because it references r.country from resultsdata15 (likely not in that table; should join to sampledata15) and answers a stricter question (countries with samples having zero detects across all analytes), which is not what was asked.	✗	

how many entries are of each specific food?	SELECT commod, count(*) FROM sampledata15 GROUP BY commod	SELECT
		  r.commod AS food,
		  COUNT(*) AS entry_count
		FROM resultsdata15 AS r
		GROUP BY r.commod
		ORDER BY entry_count DESC, food NULLS LAST;
			✗	Unreviewed query groups by the food field (commod) in the resultsdata15 table and counts rows per value, directly answering “how many entries are of each specific food?”. Ordering and aliasing don’t affect correctness.	SELECT
					  r.commod AS food,
					  COUNT(*) AS entry_count
					FROM resultsdata15 AS r
					ORDER BY entry_count DESC, food NULLS LAST;
						Unreviewed queryquery is incorrect because it does not include a GROUP BY clause.	✗	

which pesticides are most used?	SELECT pestcode FROM resultsdata15 GROUP BY pestcode ORDER BY count(*) DESC LIMIT 1	WITH detected AS (
		  SELECT
		    r.sample_pk,
		    r.pestcode
		  FROM resultsdata15 r
		  INNER JOIN sampledata15 s ON s.sample_pk = r.sample_pk
		  WHERE
		    (
		      r.determin IS NOT NULL
		      AND TRIM(UPPER(r.determin)) IN ('DETECT', 'DETECTED', 'DETECTION', 'QUANT', 'QUANTIFIED', 'YES', 'PRESENT')
		    )
		    OR (
		      r.concen IS NOT NULL
		      AND r.lod IS NOT NULL
		      AND r.concen >= r.lod
		    )
		),
		agg AS (
		  SELECT
		    d.pestcode AS pesticide,
		    COUNT(*) AS detections,
		    COUNT(DISTINCT d.sample_pk) AS sample_count
		  FROM detected d
		  WHERE d.pestcode IS NOT NULL AND d.pestcode <> ''
		  GROUP BY d.pestcode
		)
		SELECT
		  pesticide,
		  detections,
		  sample_count,
		  DENSE_RANK() OVER (ORDER BY detections DESC, sample_count DESC, pesticide) AS rank_order
		FROM agg
		ORDER BY detections DESC, sample_count DESC, pesticide
		LIMIT 50;
			✅	Unreviewed query is correct because it measures “most used” as most frequently detected: it filters to true detections (via determin flags or concentration ≥ LOD), excludes null/blank pestcodes, aggregates detections and distinct samples, and ranks pesticides by detection frequency across samples.	WITH detected AS (
					  SELECT DISTINCT
					    r.sample_pk,
					    r.pestcode
					  FROM resultsdata15 r
					  WHERE
					    r.pestcode IS NOT NULL
					    AND r.pestcode <> ''
					    AND (
					      (r.determin IS NOT NULL AND TRIM(UPPER(r.determin)) IN ('DETECT', 'DETECTED', 'DETECTION', 'QUANT', 'QUANTIFIED', 'YES', 'PRESENT'))
					      OR (r.concen IS NOT NULL AND r.lod IS NOT NULL AND r.concen >= r.lod)
					    )
					),
					agg AS (
					  SELECT
					    d.pestcode AS pesticide,
					    COUNT(*) AS positive_samples
					  FROM detected d
					  GROUP BY d.pestcode
					)
					SELECT
					  pesticide,
					  positive_samples,
					  DENSE_RANK() OVER (ORDER BY positive_samples DESC, pesticide) AS rank_order
					FROM agg
					ORDER BY positive_samples DESC, pesticide
					LIMIT 50;
						reviewed query correctly measures “most used” as the pesticides detected in the largest number of unique samples. It filters to true positives (by determin or concen ≥ lod), deduplicates per sample-pesticide, aggregates counts, and ranks accordingly—avoiding inflation from multiple results on the same sample.	✅		

which pesticides have been used the least?	SELECT pestcode FROM resultsdata15 GROUP BY pestcode ORDER BY count(*) LIMIT 1	WITH usage_counts AS (
		  SELECT
		    r.pestcode AS pesticide_code,
		    COUNT(*) AS usage_count
		  FROM resultsdata15 r
		  WHERE r.pestcode IS NOT NULL
		  GROUP BY r.pestcode
		),
		min_usage AS (
		  SELECT MIN(usage_count) AS min_count
		  FROM usage_counts
		)
		SELECT u.pesticide_code, u.usage_count
		FROM usage_counts u
		JOIN min_usage m ON u.usage_count = m.min_count
		ORDER BY u.pesticide_code;
			✅	Unreviewed query correctly computes usage counts per pesticide, excludes NULL pestcodes, finds the global minimum, and returns all pesticides tied for the least usage (matching the plural “which pesticides” and handling ties).
				Benchamrk query returns only a single pesticide due to LIMIT 1 (fails to return all least-used pesticides in case of ties) and may include NULL pestcodes, which are not meaningful identifiers. It also arbitrarily picks one in a tie without a deterministic tie-breaker.
					WITH usage_counts AS (
					  SELECT
					    r.pestcode AS pesticide_code,
					    COUNT(*) AS usage_count
					  FROM resultsdata15 r
					  WHERE r.pestcode IS NOT NULL
					  GROUP BY r.pestcode
					),
					min_usage AS (
					  SELECT MIN(usage_count) AS min_count
					  FROM usage_counts
					)
					SELECT u.pesticide_code, u.usage_count
					FROM usage_counts u
					JOIN min_usage m ON u.usage_count = m.min_count
					ORDER BY u.pesticide_code;
						Reviewer didn't make any changes.
						They compute usage per pesticide (excluding NULLs), find the global minimum, and return all pesticides tied for least usage—matching “which pesticides have been used the least?”
						Benchmark query returns only one pesticide due to LIMIT 1, failing to return all ties and may include NULL as a “pesticide,” so it does not fully answer the question.
							✅	In this example question asks for "pestesides" but query limits results to 1 output

how was a specific sample tested?	SELECT sample_pk, testclass FROM resultsdata15	SELECT
		  r.sample_pk,
		  r.testclass AS test_class,
		  r.extract AS extraction_method,
		  r.confmethod AS confirm_method_primary,
		  r.confmethod2 AS confirm_method_secondary,
		  r.lab AS laboratory,
		  r.lod AS limit_of_detection,
		  r.annotate AS method_notes,
		  r.pestcode AS analyte,
		  s.year,
		  s.month,
		  s.day,
		  s.state,
		  s.site,
		  s.distst
		FROM resultsdata15 r
		INNER JOIN sampledata15 s
		  ON r.sample_pk = s.sample_pk
		WHERE r.sample_pk = $1;
			✅	Unreviewed query correctly targets a specific sample via r.sample_pk = $1 and returns the testing methodology (test class, extraction method, confirmation methods, lab, LOD, method notes) and analyte context, which collectively answer “how was it tested?” Benchamrk query is incorrect because it isn’t filtered to a specific sample and only returns testclass, omitting key method details (extraction, confirmation methods, lab, LOD, notes).	SELECT
					  r.sample_pk,
					  r.testclass AS test_class,
					  r.extract AS extraction_method,
					  r.confmethod AS confirm_method_primary,
					  r.confmethod2 AS confirm_method_secondary,
					  r.lab AS laboratory,
					  r.lod AS limit_of_detection,
					  r.annotate AS method_notes,
					  r.pestcode AS analyte,
					  s.year,
					  s.month,
					  s.day,
					  s.state,
					  s.site,
					  s.distst
					FROM resultsdata15 r
					INNER JOIN sampledata15 s
					  ON r.sample_pk = s.sample_pk
					WHERE r.sample_pk = $1;
						Reviewer didn't make any changes.
						Unreviewed query and reviewed query correctly filter to the specific sample (r.sample_pk = $1) and return the method-related fields (testclass, extract, confmethod, confmethod2, lab, lod, annotate), which answer how the sample was tested.
						Unreviewed query is incorrect: it lacks a filter for the specific sample and only returns sample_pk and testclass, omitting key method details.
							✅	

in which state was a specific sample grown?	SELECT sample_pk, growst FROM sampledata15	WITH target_sample AS (SELECT rd.sample_pk FROM resultsdata15 rd WHERE rd.sample_pk = :target_sample_pk) SELECT sd.growst AS growing_state, sd.sample_pk AS sample_pk FROM sampledata15 sd INNER JOIN target_sample ts ON sd.sample_pk = ts.sample_pk WHERE sd.country = 'USA';	✅	Unreviewed query correctly targets a specific sample via :target_sample_pk and returns its growing state (sd.growst). The extra join to resultsdata15 and the USA filter are unnecessary and could exclude some cases, but it still answers the specific-sample question.	SELECT sd.growst AS growing_state, sd.sample_pk FROM sampledata15 sd WHERE sd.sample_pk = :target_sample_pk;	reviewed query correctly filters sampledata15 by the given sample_pk and returns the growst (state) for that specific sample.	✅	
in which state was a specific sample packed?	SELECT sample_pk, packst FROM sampledata15	SELECT s.packst AS packing_state FROM resultsdata15 r INNER JOIN sampledata15 s ON r.sample_pk = s.sample_pk WHERE r.sample_pk = :target_sample_pk LIMIT 1;	✅	Unreviewed query is correct because it selects the packing state (s.packst) for the specified sample using a parameterized filter on sample_pk. While the INNER JOIN to resultsdata15 is unnecessary, it does not change the returned state for samples present in both tables and ensures a single targeted result. Benchamrk query is incorrect because it returns packst for all samples without filtering to the specific sample; it does not answer “for a specific sample.”	SELECT packst AS packing_state FROM sampledata15 WHERE sample_pk = :target_sample_pk;	reviewed query correctly filters by the specific sample (sample_pk) and returns its packing state (packst) from the authoritative sample table.	✅	

in which state was a specific sample distributed?	SELECT sample_pk, distst FROM sampledata15	WITH target_sample AS (
		  SELECT r.sample_pk
		  FROM resultsdata15 r
		  WHERE r.sample_pk = $1
		)
		SELECT s.distst AS distribution_state
		FROM target_sample t
		INNER JOIN sampledata15 s
		  ON s.sample_pk = t.sample_pk
		LIMIT 1;
			✅	Unreviewed query filters to the specific sample via the parameter ($1) and returns its distribution state (distst) from sampledata15, which directly answers the question. The join to resultsdata15 is unnecessary and could exclude samples not present there, but for a given sample_id present in resultsdata15 it yields the correct state. Benchamrk query does not filter by a specific sample and returns all samples and their states, so it does not answer the specific-sample question.	SELECT distst AS distribution_state
					FROM sampledata15
					WHERE sample_pk = $1
					LIMIT 1;
						reviewed query directly returns the distribution state (distst) for the specified sample_pk ($1), which answers the question precisely.	✅	

<table>
<tr><td>show all imported samples?</td><td>SELECT sample_pk FROM sampledata15 WHERE origin = "2"</td><td>WITH joined AS (
  SELECT
    sd.sample_pk AS sample_id,
    sd.origin,
    sd.country,
    sd.year,
    sd.month,
    sd.day,
    sd.variety,
    sd.state,
    sd.site,
    sd.distst
  FROM sampledata15 sd
  INNER JOIN resultsdata15 rd
    ON rd.sample_pk = sd.sample_pk
  WHERE sd.sample_pk IS NOT NULL
    AND (
      COALESCE(NULLIF(TRIM(sd.origin), ''), '') ILIKE 'import%'
      OR
COALESCE(NULLIF(TRIM(sd.country), ''), '') NOT ILIKE 'united states'
)
)
SELECT
  sample_id,
  origin,
  country,
  year,
  month,
  day,
  variety,
  state,
  site,
  distst
FROM joined
ORDER BY year NULLS LAST, month NULLS LAST, day NULLS LAST, sample_id;</td><td>✅</td><td>Unreviewed query correctly identifies “imported” samples using human-readable fields (origin starting with “import” or a non–United States country) and returns the relevant sample attributes with valid PostgreSQL syntax.
Benchamrk query is incorrect: it assumes a magic code (origin = 2) without evidence that origin is coded that way, and uses double quotes for a string literal, which in PostgreSQL refer to identifiers (likely causing an error like column "2" does not exist). It would also miss text values like “Imported.”</td><td>SELECT
  sd.sample_pk AS sample_id,
  sd.origin,
  sd.country,
  sd.year,
  sd.month,
  sd.day,
  sd.variety,
  sd.state,
  sd.site,
  sd.distst
FROM sampledata15 sd
WHERE

(NULLIF(BTRIM(sd.origin), '') IS NOT NULL AND sd.origin ILIKE 'import%')
  OR (

NULLIF(BTRIM(sd.country), '') IS NOT NULL
    AND sd.country !~* '^\s*(united states(of america)?|u\.?s\.?a?\.?|usa|us)\s*$'
)
ORDER BY sd.year NULLS LAST, sd.month NULLS LAST, sd.day NULLS LAST, sd.sample_pk;</td><td>reviewed query correctly identifies imported samples without restricting to only those with results, and robustly classifies imports as either origin starting with “import” or any non-empty country that is not the US (handling common US variants via regex). It returns full sample details and avoids duplicates.</td><td>✅</td><td></td></tr>
<tr><td>which foods are captured in the data set?</td><td>SELECT DISTINCT commod FROM sampledata15</td><td>SELECT DISTINCT r.commod AS food
FROM resultsdata15 AS r
WHERE r.commod IS NOT NULL
ORDER BY food;</td><td>✅</td><td>Unreviewed query correctly returns the distinct, non-null list of foods (commod) present in the results dataset, which answers “which foods are captured in the data set?”</td><td>SELECT DISTINCT r.commod AS food
FROM resultsdata15 AS r
WHERE r.commod IS NOT NULL
ORDER BY food;</td><td>Reviewer query correctly list distinct, non-null foods (commod) from the results dataset (resultsdata15), aligning with the question and avoiding NULLs; ORDER BY is fine.</td><td>✅</td><td></td></tr>
</table>

how many samples have unknown countries of origin?	SELECT count(*) FROM sampledata15 WHERE origin = "3"	WITH joined_samples AS (
		 SELECT
		  r.sample_pk,
		  s.country
		 FROM resultsdata15 AS r
		 INNER JOIN sampledata15 AS s
		  ON r.sample_pk = s.sample_pk
		),
		unknown_country_samples AS (
		 SELECT DISTINCT sample_pk
		 FROM joined_samples
		 WHERE country IS NULL
		   OR country = "
		   OR country ILIKE 'unknown'
		   OR country ILIKE 'unk%'
		)
		SELECT COUNT(*) AS unknown_country_sample_count
		FROM unknown_country_samples;
			✅	Unreviewed query identifies samples whose country field is unknown (NULL, empty, or begins with “unk”), and counts distinct sample_pk to avoid double-counting due to multiple result rows. This directly answers “how many samples have unknown countries of origin?” Benchamrk query is incorrect: it assumes an undocumented code (origin = "3") represents “unknown,” uses double quotes for a string literal (invalid in many SQL dialects), and ignores NULL/empty/“unknown” textual values. It also targets a different column than “country,” which the question implies.	SELECT COUNT(DISTINCT sample_pk) AS unknown_country_sample_count
					FROM sampledata15
					WHERE country IS NULL
					  OR country = "
					  OR country ILIKE 'unknown'
					  OR country ILIKE 'unk%';
						reviewed query correctly counts unique samples in sampledata15 whose country is missing or labeled as unknown (NULL, empty string, or strings starting with “unk”, case-insensitive). It operates directly on the samples table and returns a sample-level count.	✅	

Which are the top 10 commodities that have the highest residue during 2015?	SELECT T2.commod FROM resultsdata15 as T2 JOIN sampledata15 as T1 ON T1.sample_pk = T2.sample_pk WHERE T1.year = 15 GROUP BY T2.commod ORDER BY sum(T2.concen) DESC LIMIT 10	WITH joined AS (
		  SELECT
		    r.commod AS commodity,
		    r.concen
		  FROM resultsdata15 r
		  INNER JOIN sampledata15 s
		    ON r.sample_pk = s.sample_pk
		  WHERE s.year = '2015'
		),
		aggregated AS (
		  SELECT
		    commodity,
		    SUM(concen) AS total_residue
		  FROM joined
		  GROUP BY commodity
		)
		SELECT
		  commodity,
		  total_residue
		FROM aggregated
		ORDER BY total_residue DESC NULLS LAST
		LIMIT 10;
			✅	Unreviewed query correctly joins results to samples, filters for year 2015, aggregates residue (SUM(concen)) by commodity, orders descending, and limits to the top 10.	SELECT
					  r.commod AS commodity,
					  MAX(r.concen) AS max_residue
					FROM resultsdata15 r
					JOIN sampledata15 s ON r.sample_pk = s.sample_pk
					WHERE s.year = '2015'
					GROUP BY r.commod
					ORDER BY max_residue DESC NULLS LAST
					LIMIT 10;
						Reviewed query correctly interprets “highest residue” as the maximum residue per commodity in 2015, filters year properly, groups by commodity, and orders by the maximum concentration descending to return the top 10.	✅	

Name some imported commodities that are not found in the US.	SELECT commod FROM sampledata15 WHERE origin = 2 AND commod not in (SELECT commod FROM sampledata15 WHERE origin = 1)	SELECT DISTINCT r.commtype AS commodity FROM resultsdata15 AS r WHERE r.commtype IS NOT NULL;	❌	Benchmark-proposed query correctly returns commodities that appear with origin = 2 (imports) and do not appear with origin = 1 (US), matching “imported commodities that are not found in the US.” Unreviewed query merely lists distinct commodity types from resultsdata15 without filtering for imports or excluding US, so it does not answer the question.	WITH us_comm AS (SELECT DISTINCT commtype FROM sampledata15 WHERE commtype IS NOT NULL AND country IS NOT NULL AND UPPER(country) IN ('UNITED STATES', 'UNITED STATES OF AMERICA', 'USA', 'US')) SELECT DISTINCT s.commtype FROM sampledata15 s LEFT JOIN us_comm u ON u.commtype = s.commtype WHERE s.commtype IS NOT NULL AND s.country IS NOT NULL AND UPPER(s.country) NOT IN ('UNITED STATES', 'UNITED STATES OF AMERICA', 'USA', 'US') AND u.commtype IS NULL;	Reviewed query correctly returns commodities that appear only in non-US rows (treated as imported) and excludes any commodity present in US rows via an anti-join, handles NULLs, and deduplicates.	✅	

Tell me the average pesticide residue for each state in the US where food is grown.	SELECT T1.growst, avg(T2.concen) FROM sampledata15 as T1 JOIN resultsdata15 as T2 ON T1.sample_pk = T2.sample_pk GROUP BY T1.growst	WITH joined AS (SELECT s.state, r.concen FROM resultsdata15 AS r INNER JOIN sampledata15 AS s ON r.sample_pk = s.sample_pk WHERE COALESCE(s.country, 'United States') = 'United States' AND r.concen IS NOT NULL) SELECT state, AVG(concen) AS avg_residue FROM joined GROUP BY state ORDER BY state NULLS LAST;	✗	Benchmark-Proposed query groups by growst (the state where the commodity was grown) and averages concen from the joined results, matching "average pesticide residue for each state in the US where food is grown." Unreviewed query groups by s.state (sampling/collection state), not the growing state, and filters on s.country (treating NULL as US), which can include imported products sampled in the US and misclassify unknown origins.	SELECT s.origin AS state, AVG(r.concen) AS avg_residue FROM resultsdata15 AS r INNER JOIN sampledata15 AS s ON r.sample_pk = s.sample_pk WHERE s.commtype = 'FOOD' AND s.origin IS NOT NULL AND r.concen IS NOT NULL GROUP BY s.origin ORDER BY s.origin;	Reviewed query is incorrect because it filters only on concen > 0 but does not restrict to samples grown in the United States. As a result, it can include foreign-grown food, which violates the question's requirement for "each state in the US where food is grown."	✗	

Which varieties of apple typically have higher pesticide levels?	SELECT T1.variety FROM resultsdata15 as T2 JOIN sampledata15 as T1 ON T1.sample_pk = T2.sample_pk WHERE T2.commod = "AP" GROUP BY T1.variety ORDER BY sum(T2.concen) DESC LIMIT 1	WITH apple_results AS (SELECT r.sample_pk, r.pestcode, r.conunit, COALESCE(r.concen, 0.0) AS working_conc FROM resultsdata15 r WHERE r.commod = 'Apple' AND r.conunit IS NOT NULL), harmonized AS (SELECT ar.sample_pk, ar.pestcode, ar.working_conc FROM apple_results ar WHERE ar.conunit = (SELECT ar2.conunit FROM apple_results ar2 GROUP BY ar2.conunit ORDER BY COUNT(*) DESC LIMIT 1)), joined AS (SELECT s.variety, h.sample_pk, h.pestcode, h.working_conc FROM harmonized h INNER JOIN sampledata15 s ON s.sample_pk = h.sample_pk WHERE COALESCE(s.claim, '') NOT ILIKE '%organic%' AND s.variety IS NOT NULL), by_variety AS (SELECT j.variety, AVG(j.working_conc) AS avg_conc, SUM(CASE WHEN j.working_conc > 0 THEN 1 ELSE 0 END)::float / NULLIF(COUNT(*),0) AS pct_detections, COUNT(DISTINCT j.sample_pk) AS n_samples, COUNT(DISTINCT j.pestcode) AS n_pesticides FROM joined j GROUP BY j.variety) SELECT variety, avg_conc, pct_detections, n_samples, n_pesticides	✗	Unreviewed query is incorrect since it filters to apples, harmonizes units (keeps the modal conunit to avoid mixing units)	WITH sample_level AS (SELECT s.variety, r.sample_pk, COUNT(DISTINCT CASE WHEN COALESCE(r.concen, 0) > 0 THEN r.pestcode END) AS detects FROM sampledata15 s JOIN resultsdata15 r ON r.sample_pk = s.sample_pk WHERE s.commod = 'Apple' AND s.variety IS NOT NULL GROUP BY s.variety, r.sample_pk), variety_agg AS (SELECT variety, AVG(detects)::float AS avg_detects_per_sample, SUM(CASE WHEN detects > 0 THEN 1 ELSE 0 END)::float / COUNT(*) AS pct_samples_with_any_detect, COUNT(*) AS n_samples FROM sample_level GROUP BY variety) SELECT variety, avg_detects_per_sample, pct_samples_with_any_detect, n_samples FROM variety_agg WHERE n_samples >= 5 ORDER BY avg_detects_per_sample DESC, pct_samples_with_any_detect DESC, n_samples DESC;	Reviewed query is incorrect because: It measures counts of detected pesticides, not concentration levels, so it answers "how many pesticides" rather than "how high the levels are." It does not address unit comparability.	✗		

		FROM by_variety WHERE n_samples >= 5 ORDER BY avg_conc DESC, pct_detections DESC, n_pesticides DESC;						

Which commodities have the pesticides concentration much higher than their limit for detection?	SELECT commod FROM resultsdata15 WHERE concen > lod	WITH ratios AS (SELECT r.commod, r.concen, r.lod, r.concen / r.lod AS ratio FROM resultsdata15 r WHERE (r.testclass ILIKE 'pesticide%' OR r.testclass = 'Pesticide' OR r.testclass = 'Pesticides') AND r.concen IS NOT NULL AND r.lod IS NOT NULL AND r.lod > 0 AND (r.quantitate IS NULL OR r.quantitate NOT ILIKE 'ND')) SELECT DISTINCT commod FROM ratios WHERE ratio > 1 ORDER BY commod;	❌	Uneviewed query properly filters to pesticides but only uses ratio > 1 (just above LOD, not “much higher”), so it does not meet the intent.	WITH pesticide_results AS (SELECT r.commod AS commodity, r.conunit AS unit, r.concen AS concentration, r.lod AS limit_of_detection, CASE WHEN r.lod > 0 THEN r.concen / r.lod END AS multiple_of_lod FROM resultsdata15 r INNER JOIN sampledata15 s ON s.sample_pk = r.sample_pk WHERE (r.testclass ILIKE 'pesticide%' OR r.testclass = 'Pesticide' OR r.testclass = 'Pesticides') AND r.concen IS NOT NULL AND r.lod IS NOT NULL AND r.lod > 0 AND (r.quantitate IS NULL OR r.quantitate NOT ILIKE 'ND') AND r.conunit IS NOT NULL), high_exceedances AS (SELECT commodity, unit, concentration, limit_of_detection, multiple_of_lod FROM pesticide_results WHERE multiple_of_lod >= 10) SELECT DISTINCT commodity FROM high_exceedances ORDER BY commodity NULLS LAST;	Reviewed query is correct because it restricts to pesticide tests, computes the concentration-to-LOD ratio, and uses a high threshold (>= 10x) to operationalize “much higher,” while guarding against null/zero LOD and excluding non-detects; it returns distinct commodities as asked.	✅	

Name top 5 labs with the maximum number of testing.	SELECT lab FROM resultsdata15 GROUP BY lab ORDER BY count(*) DESC LIMIT 5	WITH lab_counts AS (SELECT lab AS lab_name, COUNT(*) AS test_count FROM resultsdata15 GROUP BY lab) SELECT lab_name, test_count FROM lab_counts ORDER BY test_count DESC LIMIT 5;	✅	Unreviewed query correctly groups by lab, counts tests, orders by count descending, and limits to 5; it also returns the counts, which directly reflect “maximum number of testing.”	WITH lab_counts AS (SELECT lab AS lab_name, COUNT(*) AS test_count FROM resultsdata15 GROUP BY lab) SELECT lab_name, test_count FROM lab_counts ORDER BY test_count DESC LIMIT 5;	All three correctly aggregate tests by lab and order by the count descending, returning the top 5 labs.	✅	

Which country is the safest in terms of the pesticide concentration found in imported foods?	SELECT T1.country FROM sampledata15 as T1 JOIN resultsdata15 as T2 ON T1.sample_pk = T2.sample_pk GROUP BY T1.country ORDER BY sum(T2.concen) LIMIT 1	WITH standardized AS (
		  SELECT
		    r.sample_pk,
		    r.concen,
		    r.lod,
		    r.conunit,
		    CASE
		      WHEN r.conunit IS NULL THEN r.concen
		      ELSE r.concen
		    END AS std_concen
		  FROM resultsdata15 r
		),
		handled AS (
		  SELECT
		    s.sample_pk,
		    CASE
		      WHEN std_concen IS NULL AND lod IS NOT NULL THEN lod / 2.0
		      WHEN lod IS NOT NULL AND std_concen < lod THEN lod / 2.0
		      ELSE std_concen
		    END AS adj_concen,
		    CASE
		      WHEN std_concen IS NULL AND lod IS NOT NULL THEN 1
		      WHEN lod IS NOT NULL AND std_concen < lod THEN 1
		      ELSE 0
		    END AS nd_flag
		  FROM standardized s
		  WHERE s.std_concen IS NULL OR s.std_concen >= 0
		),
		joined AS (
		  SELECT
		    sd.country,
		    h.adj_concen,
		    h.nd_flag
		  FROM handled h
		  INNER JOIN sampledata15 sd
		    ON h.sample_pk = sd.sample_pk
		  WHERE sd.country IS NOT NULL
		)
		SELECT
		  country,
		  AVG(adj_concen) AS safety_metric,
		  AVG(nd_flag::double precision) AS nd_rate,
		  COUNT(*) AS n_results
		FROM joined
		GROUP BY country
		ORDER BY safety_metric ASC, nd_rate DESC, n_results DESC
		LIMIT 1;
			✅	Unreviewed query is correct because it measures safety by the average pesticide concentration per country, handles non-detects via LOD/2, excludes invalid negatives, and selects the country with the lowest mean concentration—aligning with “safest in terms of pesticide concentration.”	WITH standardized AS (
					  SELECT
					    r.sample_pk,
					    r.concen,
					    r.lod,
					    r.conunit,
					    CASE
					      WHEN r.concen IS NULL THEN NULL
					      WHEN LOWER(COALESCE(r.conunit, '')) IN ('ppm','mg/kg') THEN r.concen
					      WHEN LOWER(COALESCE(r.conunit, '')) IN ('ppb','ug/kg','µg/kg','mcg/kg') THEN r.concen / 1000.0
					      WHEN LOWER(COALESCE(r.conunit, '')) = 'ng/g' THEN r.concen / 1000.0
					      ELSE r.concen
					    END AS std_concen_mgkg
					  FROM resultsdata15 r
					),
					handled AS (
					  SELECT
					    s.sample_pk,
					    CASE
					      WHEN s.std_concen_mgkg IS NULL AND s.lod IS NOT NULL THEN s.lod / 2.0
					      WHEN s.lod IS NOT NULL AND s.std_concen_mgkg < s.lod THEN s.lod / 2.0
					      ELSE s.std_concen_mgkg
					    END AS adj_concen_mgkg
					  FROM standardized s
					  WHERE s.std_concen_mgkg IS NULL OR s.std_concen_mgkg >= 0
					),
					joined AS (
					  SELECT
					    sd.country,
					    h.adj_concen_mgkg
					  FROM handled h
					  JOIN sampledata15 sd ON sd.sample_pk = h.sample_pk
					  WHERE sd.country IS
						reviewed query is correct because it:
						 Restricts to imported foods by excluding United States.
						 Normalizes concentrations to a common unit (mg/kg), making cross-country averages meaningful.
						 Accounts for non-detects by substituting LOD/2 and removes negative values.
						 Uses average concentration per country (a proper safety metric) and returns the country with the lowest average.
							✅	

					NOT NULL AND
					sd.country <> 'United States'
					)
					SELECT country
					FROM (
					  SELECT country,
					AVG(adj_concen_mgkg)
					AS avg_conc, COUNT(*)
					AS n_results
					  FROM joined
					  GROUP BY country
					) x
					ORDER BY avg_conc ASC,
					n_results DESC
					LIMIT 1;				

Which is the most popular state in the US in terms of commodities distribution?	SELECT distst FROM sampledata15 GROUP BY distst ORDER BY count(*) DESC LIMIT 1	WITH cleaned AS (SELECT state, NULLIF(TRIM(distst), '')::numeric AS distst_num FROM sampledata15 WHERE COALESCE(UPPER(TRIM(country)), '') IN ('US', 'USA', 'UNITED STATES', 'UNITED STATES OF AMERICA') AND state IS NOT NULL AND TRIM(state) <> ''), agg AS (SELECT state, SUM(distst_num) AS total_distst FROM cleaned WHERE distst_num IS NOT NULL GROUP BY state), ranked AS (SELECT state, total_distst, RANK() OVER (ORDER BY total_distst DESC) AS rnk FROM agg) SELECT state, total_distst FROM ranked WHERE rnk = 1 ORDER BY state;	✅	Unreviewed query correctly filters to US records, groups by state, aggregates the distribution metric (distst) to quantify “commodities distribution,” and returns the state(s) with the highest total. This matches the intent: identify the top US state by distribution amount.	WITH us AS (SELECT TRIM(state) AS state, commod FROM sampledata15 WHERE COALESCE(UPPER(TRIM(country)), '') IN ('US', 'USA', 'UNITED STATES', 'UNITED STATES OF AMERICA') AND TRIM(COALESCE(state, '')) <> '' AND commod IS NOT NULL), agg AS (SELECT state, COUNT(DISTINCT commod) AS commodity_count FROM us GROUP BY state), ranked AS (SELECT state, commodity_count, RANK() OVER (ORDER BY commodity_count DESC) AS rnk FROM agg) SELECT state, commodity_count FROM ranked WHERE rnk = 1 ORDER BY state;	reviewed query filters to US, uses the state field, and ties “commodities distribution” to commodities by counting distinct commod per state, then returns the top state(s). This directly addresses “most popular state in the US in terms of commodities distribution.”	✅	

What is the maximum pesticide concentration ever found in a commodity?	SELECT max(concen) FROM resultsdata15	WITH ranked AS (
		  SELECT
		    r.commod AS commodity,
		    r.concen AS concentration,
		    r.conunit AS concentration_unit,
		    ROW_NUMBER() OVER (ORDER BY r.concen DESC NULLS LAST) AS rn
		  FROM resultsdata15 r
		)
		SELECT
		  commodity,
		  concentration AS max_concentration,
		  concentration_unit
		FROM ranked
		WHERE rn = 1;
			✅	Benchamrk query directly returns the maximum concentration value across all records (MAX ignores NULLs), which exactly answers the question.	WITH ranked AS (
					  SELECT
					    r.commod AS commodity,
					    r.concen AS concentration,
					    r.conunit AS concentration_unit,
					    ROW_NUMBER() OVER (ORDER BY r.concen DESC NULLS LAST) AS rn
					  FROM resultsdata15 r
					)
					SELECT
					  commodity,
					  concentration AS max_concentration,
					  concentration_unit
					FROM ranked
					WHERE rn = 1;
						Reviewer didn't make any changes.
						Reviewed query correctly returns the global maximum concentration (max(concen)) across all records, directly answering the question.
							✅	

When is sample 3763 collected?	SELECT year, month, day FROM sampledata15 WHERE sample_pk = 3763	WITH target_sample AS (
		  SELECT r.sample_pk
		  FROM resultsdata15 r
		  WHERE r.sample_pk = 3763
		)
		SELECT
		  s.year AS collection_year,
		  s.month AS collection_month,
		  s.day AS collection_day
		FROM target_sample t
		INNER JOIN sampledata15 s
		  ON s.distst::INTEGER = t.sample_pk;
			❌	Benchmark-proposed query is correct because it directly selects the collection date (year, month, day) from sampledata15 where sample_pk = 3763, which matches the question.
				Unreviewed query is incorrect because it joins sampledata15 on s.distst::INTEGER = t.sample_pk instead of joining on s.sample_pk. This mismatched join can return the wrong row or no row and unnecessarily depends on resultsdata15, which is irrelevant to the collection date.
					SELECT
					  year AS collection_year,
					  month AS collection_month,
					  day AS collection_day
					FROM sampledata15
					WHERE sample_pk = 3763;
						Reviewed query directly retrieve the collection year, month, and day for sample_pk = 3763 from sampledata15 (reviewed query just renames the columns).	✅	

Which state has the most apple collected?	SELECT distst FROM sampledata15 WHERE commod = "AP" GROUP BY distst ORDER BY count(*) DESC LIMIT 1	WITH apple_samples AS (SELECT r.sample_pk, s.state FROM resultsdata15 r INNER JOIN sampledata15 s ON r.sample_pk = s.sample_pk WHERE r.commtype = 'apple'), state_counts AS (SELECT state, COUNT(*) AS apple_count FROM apple_samples GROUP BY state) SELECT state, apple_count FROM state_counts ORDER BY apple_count DESC NULLS LAST LIMIT 1;	❌	Why Benchamrk query is correct: It counts samples at the sample level (sampledata15), filters apples via the commodity code (commod = 'AP'), groups by the state field (distst), and returns the state with the highest sample count. This directly answers “Which state has the most apple collected?”	SELECT state, COUNT(*) AS apple_count FROM sampledata15 WHERE commtype = 'apple' GROUP BY state ORDER BY apple_count DESC NULLS LAST LIMIT 1;	reviewed query correctly filters apple samples at the sample level (commtype = 'apple'), groups by state, counts samples per state, and returns the top state—matching “Which state has the most apple collected?”	✅	

Which state has grow the most apples?	SELECT growst FROM sampledata15 WHERE commod = "AP" GROUP BY growst ORDER BY count(*) DESC LIMIT 1	WITH apple_samples AS (
		  SELECT
		    rd.sample_pk,
		    sd.growst
		  FROM resultsdata15 rd
		  INNER JOIN sampledata15 sd
		    ON rd.sample_pk = sd.sample_pk
		  WHERE rd.commod = 'Apple'
		    AND sd.country = 'US'
		),
		state_counts AS (
		  SELECT
		    growst AS state,
		    COUNT(DISTINCT sample_pk) AS apple_sample_count
		  FROM apple_samples
		  GROUP BY growst
		)
		SELECT state, apple_sample_count
		FROM state_counts
		ORDER BY apple_sample_count DESC NULLS LAST, state ASC
		LIMIT 1;
			✗	Benchamrk query is correct because it counts apple samples per growing state directly from the sample metadata (sampledata15) using the commodity code for apples (commod = 'AP'), groups by growst, and returns the state with the highest count. This avoids duplication from result rows and aligns with the intent to find the state with the most apple samples. Note: use single quotes for the literal ('AP') in most SQL dialects. Unreviewed query is incorrect because it filters resultsdata15 with rd.commod = 'Apple', whereas commodity values are typically coded (e.g., 'AP'), so it likely matches nothing. The join to resultsdata15 is unnecessary for counting samples by state and, despite using DISTINCT, adds complexity without benefit.	SELECT
					  state,
					  SUM(quantity) AS total_quantity
					FROM sampledata15
					WHERE commod = 'Apple'
					  AND country = 'US'
					GROUP BY state
					ORDER BY total_quantity DESC NULLS LAST, state ASC
					LIMIT 1;
						reviewed query is incorrect because it references non-matching columns/values for this schema (state vs growst, commod = 'Apple' vs 'AP') and assumes a quantity field to sum; even if present, sample “quantity” would not represent total production.	✗	

How many class of products are there?	SELECT count(DISTINCT variety) FROM sampledata15	SELECT COUNT(DISTINCT testclass) AS count_distinct_classes FROM resultsdata15 WHERE testclass IS NOT NULL;	✅	Unreviewed query counts the number of distinct values in the testclass column, which directly corresponds to “class of products.” The WHERE clause excluding NULLs is redundant but harmless since COUNT(DISTINCT) ignores NULLs. Benchamrk query counts distinct variety values in a different table (sampledata15). “Variety” is not the same as “class,” so it does not answer the question about product classes.	SELECT COUNT(DISTINCT testclass) AS count_distinct_classes FROM resultsdata15 WHERE testclass IS NOT NULL;	Unreviewed query and reviewed query correctly count the number of distinct product classes via COUNT(DISTINCT testclass) on resultsdata15 while excluding NULLs. Reviewer didn't make any changes.	✅	
what's the 4 digit collection code of sample 3763?	SELECT site FROM sampledata15 WHERE sample_pk = 3763	SELECT collection_code FROM samples WHERE sample_id = 3763 AND length(collection_code) = 4 LIMIT 1;	✅	Unreviewed query selects the collection_code for sample_id = 3763 and constrains it to 4 characters, matching the request for a 4-digit collection code. Benchamrk query returns the site field from a different table and does not retrieve the collection_code nor enforce a 4-digit constraint, so it does not answer the question.	Reviewer Faild	reviewed query is not a valid SQL query.	❌	This example shows sometime LLMs doesn't generate response in form of prompted request. The system after several tries return "Faild" response

If sample 6480 is imported, which country is it originally from?	SELECT country FROM sampledata15 WHERE sample_pk = 6480 AND origin = 2	WITH target_sample AS (
		  SELECT r.sample_pk
		  FROM resultsdata15 r
		  WHERE r.sample_pk = 6480
		)
		SELECT s.country AS country
		FROM target_sample t
		INNER JOIN sampledata15 s
		  ON t.sample_pk = s.distst;
			❌	Benchamrk query correctly returns the country for sample_pk = 6480 only if the sample is imported (origin = 2), which aligns with the question's condition about being imported. Unreviewed query joins on s.distst = r.sample_pk (likely a mismatched key; it should be s.sample_pk), and it does not filter for imported status, so it can return incorrect or unrelated rows.	SELECT country
					FROM sampledata15
					WHERE sample_pk = 6480;
						Unreviewed query correctly applies the conditional ("if imported") by filtering on origin = 2 and returns the country for sample_pk = 6480.	✅	

How many number of units are there in sample 9628?	SELECT quantity FROM sampledata15 WHERE sample_pk = 9628	SELECT 'Please provide the table name(s) that contain "sample" records and "units", along with full column lists and brief sample rows if available.' AS message;	❌	Benchamrk query directly retrieves the number of units by selecting the quantity for sample_pk = 9628, which aligns with the question. Unreviewed query does not query any table or return a unit count; it only returns a static message, so it doesn't answer the question.	SELECT quantity FROM sampledata15 WHERE sample_pk = 9628;	Reviewed query correctly selects the number of units (quantity) for sample_pk = 9628 from sampledata15. Reviewer didn't make any changes.	✅	In this example the LLM returns a static message instead of sql query.
What's the code for test for sample 7498?	SELECT testclass FROM resultsdata15 WHERE sample_pk = 7498	SELECT DISTINCT pestcode AS code FROM resultsdata15 WHERE sample_pk = 7498;	❌	Benchmark-proposed query is correct because it returns the test-related field (testclass) for sample 7498, which represents the test's identifier/code. Unreviewed query is incorrect because pestcode is the analyte/pesticide code, not the test code the question asks for.	SELECT DISTINCT pestcode AS code FROM resultsdata15 WHERE sample_pk = 7498;	Reviewed query didn't replaced the wrong column name so the reviwed query is still incorrect.	❌	

What's the code for confirmation for the latest sample?	SELECT confmethod FROM resultsdata15 as T2 JOIN sampledata15 as T1 ON T1.sample_pk = T2.sample_pk ORDER BY year, month, day DESC LIMIT 1	WITH joined AS (
		  SELECT
		    r.sample_pk,
		    COALESCE(NULLIF(r.confmethod, ''), NULLIF(r.confmethod2, '')) AS conf_code,
		    s.year,
		    s.month,
		    s.day
		  FROM resultsdata15 r
		  INNER JOIN sampledata15 s ON s.sample_pk = r.sample_pk
		),
		ordered AS (
		  SELECT
		    sample_pk,
		    conf_code,
		    ROW_NUMBER() OVER (
		      ORDER BY
		        NULLIF(s.year, '')::int DESC NULLS LAST,
		        NULLIF(s.month, '')::int DESC NULLS LAST,
		        NULLIF(s.day, '')::int DESC NULLS LAST,
		        sample_pk DESC NULLS LAST
		    ) AS rn
		  FROM joined s
		)
		SELECT conf_code
		FROM ordered
		WHERE rn = 1;
			✅	Unreviewed query correctly identifies the latest sample by ordering year, month, and day in descending numeric order (handling blanks as NULLs and using a tie-breaker), and returns the first non-empty confirmation code (confmethod, falling back to confmethod2). Benchamrk query orders only day in descending order while year and month default to ascending, which can return a non-latest sample; it also ignores empty-string handling and the confmethod2 fallback.	WITH joined AS (
					  SELECT
					    r.sample_pk,
					COALESCE(NULLIF(r.confmethod, ''), NULLIF(r.confmethod2, '')) AS conf_code,
					    s.year,
					    s.month,
					    s.day
					  FROM resultsdata15 r
					  INNER JOIN sampledata15 s ON s.sample_pk = r.sample_pk
					),
					ordered AS (
					  SELECT
					    sample_pk,
					    conf_code,
					    ROW_NUMBER() OVER (
					      ORDER BY
					        NULLIF(s.year, '')::int DESC NULLS LAST,
					        NULLIF(s.month, '')::int DESC NULLS LAST,
					        NULLIF(s.day, '')::int DESC NULLS LAST,
					        sample_pk DESC NULLS LAST
					    ) AS rn
					  FROM joined s
					)
					SELECT conf_code
					FROM ordered
					WHERE rn = 1;
						Unreviewed query and reviewed query correctly pick the most recent sample by ordering year, month, and day in descending order (handling empty strings and NULLs), and return a usable confirmation code via COALESCE of confmethod/confmethod2.	✅		

Which lab has analyzed the most sample?	SELECT lab FROM resultsdata15 GROUP BY lab ORDER BY count(*) DESC LIMIT 1	WITH lab_counts AS (
		  SELECT
		    lab,
		    COUNT(DISTINCT sample_pk) AS sample_count
		  FROM
		    resultsdata15
		  WHERE
		    lab IS NOT NULL
		  GROUP BY
		    lab
		),
		max_count AS (
		  SELECT MAX(sample_count) AS max_sample_count FROM lab_counts
		)
		SELECT
		  lc.lab,
		  lc.sample_count
		FROM
		  lab_counts lc
		JOIN
		  max_count mc
		ON
		  lc.sample_count = mc.max_sample_count
		ORDER BY
		  lc.sample_count DESC, lc.lab ASC
		LIMIT 1;
			✅	Unreviewed query correctly counts distinct samples per lab (COUNT(DISTINCT sample_pk)), excludes NULL labs, and returns the lab with the maximum sample count—matching the intent of “most samples analyzed.”
				Benchamrk query counts rows (COUNT(*)) rather than distinct samples, which can overcount when multiple rows exist per sample, and it doesn’t exclude NULL labs, potentially returning NULL or a lab with more rows but not more unique samples.
					WITH lab_counts AS (
					  SELECT
					    lab,
					    COUNT(DISTINCT sample_pk) AS sample_count
					  FROM
					    resultsdata15
					  WHERE
					    lab IS NOT NULL
					  GROUP BY
					    lab
					),
					max_count AS (
					  SELECT MAX(sample_count) AS max_sample_count FROM lab_counts
					)
					SELECT
					  lc.lab,
					  lc.sample_count
					FROM
					  lab_counts lc
					JOIN
					  max_count mc
					ON
					  lc.sample_count = mc.max_sample_count
					ORDER BY
					  lc.sample_count DESC, lc.lab ASC
					LIMIT 1;
						Reviewer didn't make any changes
						They correctly count distinct samples per lab (COUNT(DISTINCT sample_pk)), exclude NULL labs, and select the lab with the maximum sample count matching “Which lab has analyzed the most sample?”
							✅	

What's the majority of test classification result?	SELECT max(testclass) FROM resultsdata15	WITH counts AS (
		  SELECT
		    testclass AS majority_testclass,
		    COUNT(*) AS majority_count
		  FROM resultsdata15
		  WHERE testclass IS NOT NULL AND testclass <> ''
		  GROUP BY testclass
		),
		ranked AS (
		  SELECT
		    majority_testclass,
		    majority_count,
		    RANK() OVER (ORDER BY majority_count DESC, majority_testclass ASC) AS rnk
		  FROM counts
		)
		SELECT
		  majority_testclass,
		  majority_count
		FROM ranked
		WHERE rnk = 1
		LIMIT 1;
			✅	Unreviewed query correctly computes the mode (most frequent) testclass by counting occurrences, ranking by count (with a deterministic tie-break), and returning the top value; it also ignores NULL/empty strings, which are typically non-results.
				Benchamrk query returns the lexicographically maximum testclass, ignoring frequency, so it does not answer which class is the majority.
					WITH counts AS (
					  SELECT
					    testclass AS majority_testclass,
					    COUNT(*) AS majority_count
					  FROM resultsdata15
					  WHERE testclass IS NOT NULL AND testclass <> ''
					  GROUP BY testclass
					),
					ranked AS (
					  SELECT
					    majority_testclass,
					    majority_count,
					    RANK() OVER (ORDER BY majority_count DESC, majority_testclass ASC) AS rnk
					  FROM counts
					)
					SELECT
					  majority_testclass,
					  majority_count
					FROM ranked
					WHERE rnk = 1
					LIMIT 1;
						Reviewer didn't make any changes.
						Unreviewed query and reviewed query correctly compute the most frequent testclass (mode) by counting occurrences, ranking by count descending, and returning the top class (with a deterministic tie-breaker).
							✅		

What's the unit of measure for sample 3879?	SELECT conunit FROM resultsdata15 WHERE sample_pk = 3879	SELECT
		  resultsdata15.conunit AS unit_of_measure
		FROM
		  resultsdata15;
			❌	Benchamrk query is correct because it filters by the specific sample (sample_pk = 3879) and returns the unit of measure (conunit) for that sample. Unreviewed query is incorrect because it lacks a WHERE clause and returns the unit of measure for all samples, not specifically for sample 3879.	SELECT unit_of_measure
					FROM resultsdata15
					WHERE sample_pk = 3879;
						Reviewed queryis basesd on a non-exist column name.	❌	

What's the unit of measure used for the product commodity code PO?	SELECT conunit FROM resultsdata15 WHERE commod = "PO"	WITH filtered AS (
		  SELECT commod, conunit
		  FROM resultsdata15
		  WHERE commod = 'PO'
		)
		SELECT DISTINCT
		  commod AS commodity_code,
		  conunit AS unit_of_measure
		FROM filtered;
			✅	Unreviewed query correctly filters commod = 'PO' using standard string literals and returns the distinct unit_of_measure (eliminating duplicates), which directly answers the question.	WITH filtered AS (
					  SELECT commod, conunit
					  FROM resultsdata15
					  WHERE commod = 'PO'
					)
					SELECT DISTINCT
					  commod AS commodity_code,
					  conunit AS unit_of_measure
					FROM filtered;
						Reviewer didn't make any changes	✅	

What's the mean result finding for product AP?	SELECT mean FROM resultsdata15 WHERE commod = "AP"	WITH ap_results AS (
		  SELECT
		    concen,
		    conunit
		  FROM resultsdata15
		  WHERE commod = 'AP'
		    AND concen IS NOT NULL
		)
		SELECT
		  AVG(concen) AS mean_result,
		  conunit
		FROM ap_results
		GROUP BY conunit
		ORDER BY conunit NULLS LAST;
			✗	Unreviewed query correctly filters to commod = 'AP', excludes NULL results, and computes the mean (AVG) of the result values (concen), returning means by unit (conunit) to avoid mixing units. Benchmark-proposed query is incorrect because it does not compute an average, likely references a non-existent column named mean, and uses double quotes for a string literal ("AP"), which is invalid in many SQL dialects.	SELECT AVG(CAST(mean AS NUMERIC)) AS mean_result
					FROM resultsdata15
					WHERE commod = 'AP'
					AND mean IS NOT NULL;
						Reviewed query averages the mean column (a mean-of-means) and ignores units, which can yield a misleading result if units differ or if mean is a derived per-row summary rather than the raw result value.	✗	

What's the most common extraction method?	SELECT max(extract) FROM resultsdata15	WITH counts AS (
		  SELECT
		    extract AS extraction_method,
		    COUNT(*) AS method_count
		  FROM resultsdata15
		  WHERE extract IS NOT NULL AND extract <> ''
		  GROUP BY extract
		)
		SELECT extraction_method, method_count
		FROM counts
		ORDER BY method_count DESC, extraction_method ASC
		FETCH FIRST 1 ROW ONLY;
			✅	Unreviewed query correctly finds the mode by grouping on extract, excluding null/empty values, counting occurrences, and returning the highest count (with a deterministic tie-breaker).
				Benchamrk query uses MAX(extract), which returns the lexicographically largest value, not the most frequent one, so it does not answer the question.
					WITH counts AS (
					  SELECT
					    extract AS extraction_method,
					    COUNT(*) AS method_count
					  FROM resultsdata15
					  WHERE extract IS NOT NULL AND extract <> ''
					  GROUP BY extract
					)
					SELECT extraction_method, method_count
					FROM counts
					ORDER BY method_count DESC, extraction_method ASC
					FETCH FIRST 1 ROW ONLY;
						System-generated queries correctly compute the most common extraction method by counting non-null/non-empty values, ordering by count descending (with a deterministic tie-breaker), and returning the top result. Reviewer didn't make any changes.	✅	

Which lab conduct of most number of tests?	SELECT lab FROM resultsdata15 GROUP BY lab ORDER BY count(*) DESC LIMIT 1	WITH lab_counts AS (SELECT lab AS top_lab, COUNT(*) AS test_count FROM resultsdata15 GROUP BY lab), max_count AS (SELECT MAX(test_count) AS max_test_count FROM lab_counts) SELECT lc.top_lab, lc.test_count FROM lab_counts lc JOIN max_count mc ON lc.test_count = mc.max_test_count ORDER BY lc.top_lab NULLS LAST LIMIT 1;	✅	Unreviewed query correctly computes counts per lab, finds the maximum, and returns a lab with that maximum (also returning the count). In ties, it deterministically picks one by lab name.	WITH lab_counts AS (SELECT lab, COUNT(*) AS test_count FROM resultsdata15 GROUP BY lab), max_count AS (SELECT MAX(test_count) AS max_test_count FROM lab_counts) SELECT lc.lab FROM lab_counts lc JOIN max_count mc ON lc.test_count = mc.max_test_count ORDER BY lc.lab;	reviewed query correctly computes the maximum test count per lab and returns all lab(s) that achieved that maximum, which directly answers “Which lab conducted the most number of tests?” and handles ties appropriately.	✅	
Which lab is used for testing for prodict AP?	SELECT lab FROM resultsdata15 WHERE commod = "AP"	SELECT DISTINCT lab AS lab FROM resultsdata15 WHERE commod = 'prodict AP' AND lab IS NOT NULL;	✅	Unreviewed query correctly filters for the specified commodity ('prodict AP'), excludes NULL labs, and returns distinct lab names—matching the intent to identify which lab(s) are used.	SELECT DISTINCT lab AS lab FROM resultsdata15 WHERE commod = 'product AP' AND lab IS NOT NULL;	Reviewed query correctly targets the intended commodity name (“product AP”), returns unique labs, and excludes NULLs, matching the question’s intent despite the typo.	✅	

How many samples come from other countries?	SELECT count(*) FROM sampledata15 WHERE origin = "2"	SELECT COUNT(*) AS other_country_sample_count FROM sampledata15 WHERE country IS NOT NULL AND TRIM(BOTH FROM country) <> 'US';	✅	Unreviewed query directly counts samples whose country is specified and not US, which aligns with "from other countries." Benchamrk query assumes a coded value (origin = 2) means "other countries" without any given codebook	SELECT COUNT(*) AS other_country_sample_count FROM sampledata15 WHERE country IS NOT NULL AND LENGTH(TRIM(country)) > 0 AND LOWER(TRIM(country)) NOT IN ('us', 'usa', 'u.s.', 'u.s.a.', 'united states', 'united states of america');	Reviewed query is correct because it explicitly counts rows with a non-empty country value that is not any common variant of the United States (case-insensitive), aligning with "other countries."	✅	
What kind of food has been tested 'detect' most?	SELECT commod FROM resultsdata15 WHERE mean = "A" GROUP BY commod ORDER BY count(*) DESC LIMIT 1	WITH detects AS (SELECT commod, COUNT(*) AS detect_count FROM resultsdata15 WHERE LOWER(COALESCE(determin, '')) = 'detect' GROUP BY commod) SELECT commod AS food, detect_count FROM detects ORDER BY detect_count DESC NULLS LAST, food NULLS LAST LIMIT 1;	✅	Unreviewed query correctly filters for rows where the test result equals 'detect', groups by food (commod), counts detections, and returns the food with the highest number of detections—directly answering which food has been tested 'detect' most. Benchamrk query does not filter on 'detect' at all; it filters on mean = "A" (irrelevant to detection status and using nonstandard string quoting), so it does not answer the question.	WITH detects AS (SELECT commod, COUNT(*) AS detect_count FROM resultsdata15 WHERE LOWER(COALESCE(determin, '')) = 'detect' GROUP BY commod) SELECT commod AS food, detect_count FROM detects ORDER BY detect_count DESC NULLS LAST, food NULLS LAST LIMIT 1;	Reviewer didn't make any changes. They correctly filter rows where determin = 'detect' (case-insensitive), count detects per commod (food), and return the food with the highest detect count.	✅	

Which state produce the most organic food?	SELECT state FROM sampledata15 WHERE claim = "PO" GROUP BY state ORDER BY count(*) DESC LIMIT 1	WITH organic_rows AS (
		  SELECT
		    state
		  FROM sampledata15
		  WHERE claim IS NOT NULL
		    AND trim(lower(claim)) IN ('organic', 'org', 'organic certified', 'certified organic')
		)
		SELECT
		  state,
		  COUNT(*) AS organic_count
		FROM organic_rows
		GROUP BY state
		ORDER BY organic_count DESC, state ASC
		LIMIT 1;
			✅	Unreviewed query correctly filters rows where the claim denotes organic (case/whitespace-normalized), counts them by state, and returns the state with the highest count, which aligns with “Which state produce the most organic food?” Benchamrk query filters for claim = "PO", which does not clearly correspond to “organic” and likely excludes relevant records. It’s also case-sensitive and uses double quotes for a string literal, which may be invalid in some SQL dialects.	SELECT
					  state,
					  SUM(quantity) AS organic_quantity
					FROM sampledata15
					WHERE claim IS NOT NULL
					  AND LOWER(claim) LIKE '%organic%'
					GROUP BY state
					ORDER BY organic_quantity DESC, state ASC
					LIMIT 1;
						Reviewed query correctly measures “produce the most” by summing quantity of entries labeled organic per state and returns the top state. Unreviewed query counts rows instead of production volume and uses a narrow set of exact labels, so it can misidentify the top producer.	✅	

Where do the US import the most food ?	SELECT max(country) FROM sampledata15	WITH us_imports AS (
		  SELECT
		    country,
		    SUM(quantity) AS total_quantity
		  FROM sampledata15
		  WHERE distst = 'US'
		  GROUP BY country
		)
		SELECT
		  country AS source_country,
		  total_quantity
		FROM us_imports
		ORDER BY total_quantity DESC NULLS LAST
		LIMIT 1;
			✗	Unreviewed query filters to flows destined for the US (distst = 'US'), aggregates quantities by source country, and returns the country with the highest total quantity, which matches “from where does the US import the most food.” (Assumes the table contains food records.)
				Benchamrk query simply returns the lexicographically greatest country value across all rows, without filtering to US imports or considering quantities, so it does not answer the question.
					SELECT
					  country AS source_country,
					  SUM(quantity) AS total_quantity
					FROM sampledata15
					WHERE distst IS NOT NULL
					  AND commtype IS NOT NULL
					  AND country IS NOT NULL
					GROUP BY country
					ORDER BY total_quantity DESC
					LIMIT 1;
						Reviewed query does not restrict to US imports and only filters out NULLs, so it can include other destinations and doesn’t answer the question.	✗		

What kind of food has been test the most?	SELECT max(commod) FROM resultsdata15	WITH tests_by_food AS (
		  SELECT
		    commod AS food,
		    COUNT(DISTINCT sample_pk) AS test_count
		  FROM resultsdata15
		  WHERE commod IS NOT NULL
		    AND sample_pk IS NOT NULL
		  GROUP BY commod
		)
		SELECT food, test_count
		FROM tests_by_food
		ORDER BY test_count DESC
		FETCH FIRST 1 ROW WITH TIES;
			✅	Unreviewed query groups by food (commod), counts distinct tests via sample_pk, filters nulls, and returns the food(s) with the highest test count (including ties). This directly answers “which kind of food has been tested the most.” Benchamrk query returns the lexicographically maximum food name (MAX(commod)), which has nothing to do with how often it was tested.	WITH tests_by_food AS (
					  SELECT
					    commod AS food,
					    COUNT(DISTINCT sample_pk) AS test_count
					  FROM resultsdata15
					  WHERE commod IS NOT NULL
					    AND sample_pk IS NOT NULL
					  GROUP BY commod
					)
					SELECT food, test_count
					FROM tests_by_food
					ORDER BY test_count DESC
					FETCH FIRST 1 ROW WITH TIES;
						Reviewer didn't make any changes. They correctly count distinct samples per food (commod) and return the food(s) with the highest test count, handling ties appropriately.	✅		

Which kind of pesticide is the easiest to be tested?	SELECT max(pestcode) FROM resultsdata15	WITH valid_results AS (SELECT pestcode, lod FROM resultsdata15 WHERE lod IS NOT NULL AND lod > 0 AND COALESCE(TRIM(determin), '') <> ''), pest_lod AS (SELECT pestcode, MIN(lod) AS min_lod FROM valid_results GROUP BY pestcode), global_min AS (SELECT MIN(min_lod) AS global_min_lod FROM pest_lod) SELECT p.pestcode AS pesticide_code, p.min_lod AS ease_metric_lod FROM pest_lod p JOIN global_min g ON p.min_lod = g.global_min_lod ORDER BY p.pestcode ASC;	✅	Unreviewed query uses LOD as a proxy for “ease of testing,” finding the pesticide(s) with the lowest detection limit (minimum LOD per pestcode, then the global minimum), which directly answers the question and properly handles ties and invalid LODs. Benchamrk query simply returns the maximum pestcode value and ignores any testing sensitivity metric, so it does not address the question.	WITH valid_results AS (SELECT testclass, lod FROM resultsdata15 WHERE lod IS NOT NULL AND lod > 0 AND COALESCE(TRIM(testclass), '') <> ''), class_lod AS (SELECT testclass, MIN(lod) AS min_lod FROM valid_results GROUP BY testclass), global_min AS (SELECT MIN(min_lod) AS global_min_lod FROM class_lod) SELECT c.testclass, c.min_lod FROM class_lod c JOIN global_min g ON c.min_lod = g.global_min_lod ORDER BY c.testclass ASC;	Reviewed query is correct because it identifies the pesticide kind (testclass) with the lowest limit of detection (LOD), a valid proxy for “easiest to be tested,” using only valid, positive LODs and returning the class(es) with the global minimum.	✅		

USWildFires Database

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation	Annotator Comment
Which year has the most wildfires?	SELECT FIRE_YEAR FROM Fires GROUP BY FIRE_YEAR ORDER BY count(*) DESC LIMIT 1	WITH parsed_dates AS (SELECT CAST(date_trunc('year', to_timestamp((discovery_date - 2440587.5) * 86400)) AS date) AS year_start FROM fires WHERE discovery_date IS NOT NULL), counts AS (SELECT EXTRACT(YEAR FROM year_start)::int AS year, COUNT(*) AS fire_count FROM parsed_dates GROUP BY EXTRACT(YEAR FROM year_start)) SELECT year, fire_count FROM counts ORDER BY fire_count DESC, year ASC LIMIT 1;	✅	Unreviewed query correctly converts discovery_date (Julian day) to a timestamp, extracts the year, counts fires per year, and returns the year with the highest count (with a deterministic tie-break on year).	WITH parsed_dates AS (SELECT CAST(date_trunc('year', to_timestamp((discovery_date - 2440587.5) * 86400)) AS date) AS year_start FROM fires WHERE discovery_date IS NOT NULL), counts AS (SELECT EXTRACT(YEAR FROM year_start)::int AS year, COUNT(*) AS fire_count FROM parsed_dates GROUP BY EXTRACT(YEAR FROM year_start)) SELECT year, fire_count FROM counts ORDER BY fire_count DESC, year ASC LIMIT 1;	Reviewer didn't make any changes. All three queries correctly identify the year with the highest number of fires: Unreviewed query groups by FIRE_YEAR and orders by count descending to return the top year.	✅	

Which time of day has the most wildfires?	SELECT DISCOVERY_TIME FROM Fires GROUP BY DISCOVERY_TIME ORDER BY count(*) DESC LIMIT 1	WITH parsed_times AS (SELECT discovery_time, CASE WHEN discovery_time ~ '^\s*\d{1,4}\s*$' AND (lpad(trim(discovery_time), 4, '0'))::int BETWEEN 0 AND 2359 AND substring(lpad(trim(discovery_time), 4, '0') FROM 3 FOR 2)::int < 60 THEN lpad(trim(discovery_time), 4, '0') ELSE NULL END AS hhmm FROM fires), hour_buckets AS (SELECT (substring(hhmm FROM 1 FOR 2))::int AS hour_of_day FROM parsed_times WHERE hhmm IS NOT NULL), counts AS (SELECT hour_of_day AS time_of_day, COUNT(*) AS fire_count FROM hour_buckets GROUP BY hour_of_day) SELECT time_of_day, fire_count FROM counts ORDER BY fire_count DESC, time_of_day LIMIT 1;	✅	Unreviewed query correctly normalizes HHMM strings, validates time ranges, buckets records by hour-of-day (0–23), and returns the hour with the highest wildfire count. This aligns with "time of day" as an hour-based aggregation and excludes invalid/missing times. The Benchmark-Proposed Query groups by the raw DISCOVERY_TIME value, returning the most frequent exact recorded value (e.g., 1400) rather than aggregating by time-of-day. It doesn't normalize formats (e.g., 800 vs 0800), can be skewed by data-entry artifacts, and may even select NULL if it's most common. Hence it does not reliably answer the intended question.	WITH parsed_times AS (SELECT discovery_time, CASE WHEN discovery_time ~ '^\s*\d{1,4}\s*$' AND (lpad(trim(discovery_time), 4, '0'))::int BETWEEN 0 AND 2359 AND substring(lpad(trim(discovery_time), 4, '0') FROM 3 FOR 2)::int < 60 THEN lpad(trim(discovery_time), 4, '0') ELSE NULL END AS hhmm FROM fires), hour_buckets AS (SELECT (substring(hhmm FROM 1 FOR 2))::int AS hour_of_day FROM parsed_times WHERE hhmm IS NOT NULL), counts AS (SELECT hour_of_day AS time_of_day, COUNT(*) AS fire_count FROM hour_buckets GROUP BY hour_of_day) SELECT time_of_day, fire_count FROM counts ORDER BY fire_count DESC, time_of_day LIMIT 1;	Reviewer didn't make any changes. Queries B and C correctly parse valid HHMM times, bucket by hour, aggregate counts, and return the hour with the most fires; they are identical. Unreviewed query groups by exact discovery_time values (including minutes and possible NULLs), not by hour, so it doesn't properly answer "which time of day" and can be skewed by specific minute defaults.	✅	It is an example that shows why the system-generated queries are more reliable in case of considering all possabilities

What are the most common causes of the wildfires?	SELECT STAT_CAUSE_DESCR FROM Fires GROUP BY STAT_CAUSE_DESCR ORDER BY count(*) DESC LIMIT 1	WITH cause_counts AS (
		  SELECT
		    stat_cause_descr AS cause,
		    COUNT(*) AS fire_count
		  FROM fires
		  WHERE stat_cause_descr IS NOT NULL AND stat_cause_descr <> ''
		  GROUP BY stat_cause_descr
		)
		SELECT
		  cause,
		  fire_count,
		  RANK() OVER (ORDER BY fire_count DESC) AS rank_order
		FROM cause_counts
		ORDER BY fire_count DESC, cause ASC;
			✅	Unreviewed query correctly computes frequencies per cause, excludes null/blank values, orders by count descending, and provides a rank so the most common causes (rank 1) are identifiable; returning the full ordered list still answers which causes are most common.
				The Benchmark-Proposed Query returns only a single cause (LIMIT 1), does not filter out NULL/empty values (which could incorrectly surface), ignores ties, and provides no counts, so it does not answer "causes" (plural).
					WITH cause_counts AS (
					  SELECT
					    stat_cause_descr AS cause,
					    COUNT(*) AS fire_count
					  FROM fires
					  WHERE stat_cause_descr IS NOT NULL AND stat_cause_descr <> ''
					  GROUP BY stat_cause_descr
					)
					SELECT
					  cause,
					  fire_count,
					  RANK() OVER (ORDER BY fire_count DESC) AS rank_order
					FROM cause_counts
					ORDER BY fire_count DESC, cause ASC;
						Reviewer didn't make any changes. System generateds correctly count and rank causes, exclude null/empty values, and list causes in descending frequency, answering which causes are most common.	✅	The example shows that LLMs often have a good understanding of the question intent

How long does it take to control wildfires on average?	SELECT avg(CONT_DATE - DISCOVERY_DATE) FROM Fires	WITH parsed AS (
		  SELECT
		    discovery_date,
		    discovery_time,
		    cont_date,
		    cont_time,
		    -- parse discovery time HHMM to minutes;
			❌	The Benchmark-Proposed Query computes the average elapsed time between discovery and control as avg(CONT_DATE DISCOVERY_DATE), which answers the question (typically in days). AVG ignores nulls, so rows without dates don’t skew the result.
				Unreviewed query is incomplete: it doesn’t finish parsing times, compute a duration, or aggregate to an average, so it cannot produce the requested answer.
					Reviewer Faild	Reviewed query is not valid SQL query and doesn't return expected resutls.	❌	

Which state has the most wildfires?	SELECT State FROM Fires GROUP BY State ORDER BY count(*) DESC LIMIT 1	WITH fire_counts AS (
		  SELECT
		    state,
		    COUNT(*) AS fire_count
		  FROM fires
		  GROUP BY state
		)
		SELECT
		  state,
		  fire_count AS count
		FROM fire_counts
		ORDER BY fire_count DESC
		LIMIT 1;
			✅	Benchmark and unreviewed queries group by state, count fires per state, order by the count descending, and limit to 1, which returns the state with the most wildfires. Unreviewed query also returns the count; The Benchmark-Proposed Query returns just the state.	WITH fire_counts AS (
					  SELECT
					    state,
					    COUNT(*) AS fire_count
					  FROM fires
					  GROUP BY state
					)
					SELECT
					  state,
					  fire_count AS count
					FROM fire_counts
					ORDER BY fire_count DESC
					LIMIT 1;
						All three queries correctly count fires per state and return the state with the highest count (ORDER BY ... DESC LIMIT 1).	✅	

Which state has the largest wildfire?	SELECT State FROM Fires ORDER BY FIRE_SIZE DESC LIMIT 1	SELECT state, fire_size AS max_fire_size FROM fires WHERE fire_size = (SELECT MAX(fire_size) FROM fires);	✅	Unreviewed query correctly returns the state(s) associated with the global maximum fire_size and handles ties, which aligns with asking which state has the largest wildfire. The Benchmark-Proposed Query arbitrarily returns only one state due to LIMIT 1, potentially missing other states if multiple fires share the maximum size.	SELECT state, fire_size AS max_fire_size FROM fires WHERE fire_size = (SELECT MAX(fire_size) FROM fires);	Reviewer didn't make any changes. Reviewer correctly return the state(s) whose fire_size equals the overall maximum, handling ties. Benchamrk query returns only one state via LIMIT 1 and can miss other states tied for the largest wildfire, making it unreliable.	✅	It is a good example of how the system consider possibalities
What enrity is reponsible for managing the land at the point of origin of the most wildfires?	SELECT OWNER_DESCR FROM Fires GROUP BY OWNER_DESCR ORDER BY count(*) DESC LIMIT 1	WITH fire_counts AS (SELECT owner_descr AS managing_entity, COUNT(*) AS fire_count FROM fires GROUP BY owner_descr) SELECT managing_entity, fire_count FROM fire_counts ORDER BY fire_count DESC LIMIT 1;	✅	Benchmark and unreviewed queries group fires by owner_descr (the land-managing entity at the point of origin), order groups by the number of fires in descending order, and return the top entity, which matches the question's intent. Unreviewed query also returns the count; The Benchmark-Proposed Query returns just the entity.	WITH fire_counts AS (SELECT owner_descr AS managing_entity, COUNT(*) AS fire_count FROM fires GROUP BY owner_descr) SELECT managing_entity, fire_count FROM fire_counts ORDER BY fire_count DESC LIMIT 1;	All three correctly group fires by owner_descr (the managing entity at the origin), count occurrences, order by the count descending, and return the top entity.	✅	

What was the most common cause of fire between 2000 and 2005?	SELECT STAT_CAUSE_DESCR FROM Fires WHERE FIRE_YEAR BETWEEN 2000 AND 2005 GROUP BY STAT_CAUSE_DESCR ORDER BY count(*) DESC LIMIT 1	WITH normalized_dates AS (
		  SELECT
		    stat_cause_descr,
		
		(to_timestamp((discovery_date - 2440587.5) * 86400)::date) AS discovery_dt
		  FROM fires
		  WHERE discovery_date IS NOT NULL
		),
		filtered AS (
		  SELECT
		    stat_cause_descr
		  FROM normalized_dates
		  WHERE EXTRACT(YEAR FROM discovery_dt) BETWEEN 2000 AND 2005
		)
		SELECT
		  stat_cause_descr AS cause,
		  COUNT(*) AS fire_count
		FROM filtered
		GROUP BY stat_cause_descr
		ORDER BY fire_count DESC
		LIMIT 1;
			✅	Unreviewed query: Converts discovery_date to a calendar date, filters years 2000–2005 inclusive, aggregates by stat_cause_descr, and returns the top count. This yields the most common cause in that window (assuming discovery_date is stored as a Julian day number).	WITH normalized_dates AS (
					  SELECT
					    stat_cause_descr,
					
					(to_timestamp((discovery_date - 2440587.5) * 86400)::date) AS discovery_dt
					  FROM fires
					  WHERE discovery_date IS NOT NULL
					),
					filtered AS (
					  SELECT
					    stat_cause_descr
					  FROM normalized_dates
					  WHERE EXTRACT(YEAR FROM discovery_dt) BETWEEN 2000 AND 2005
					)
					SELECT
					  stat_cause_descr AS cause,
					  COUNT(*) AS fire_count
					FROM filtered
					GROUP BY stat_cause_descr
					ORDER BY fire_count DESC
					LIMIT 1;
						All three correctly find the most frequent stat_cause_descr for years 2000–2005 by filtering to that range, grouping by cause, ordering by count descending, and taking the top result.
						None are incorrect.
							✅	

How many fires were more than 100 acres in 2006?	SELECT count(*) FROM Fires WHERE FIRE_YEAR = 2006 AND FIRE_SIZE > 100	SELECT COUNT(*) AS fires_over_100_acres_2006 FROM fires WHERE fire_year = 2006 AND fire_size > 100;	✅	Benchmark and unreviewed queries count fires where fire_year = 2006 and fire_size > 100, which matches “more than 100 acres in 2006.”	SELECT COUNT(*) AS fires_over_100_acres_2006 FROM fires WHERE fire_year = 2006 AND fire_size > 100;	All three queries correctly count fires in 2006 with fire_size > 100, matching “more than 100 acres in 2006.” There are no incorrect queries.	✅	
Which states had the largest number of fires in 2001?	SELECT State FROM Fires WHERE FIRE_YEAR = 2001 GROUP BY State ORDER BY count(*) DESC LIMIT 1	WITH fires_2001 AS (SELECT state, COUNT(*) AS fire_count FROM fires WHERE fire_year = 2001 GROUP BY state), max_count AS (SELECT MAX(fire_count) AS max_fire_count FROM fires_2001) SELECT f.state, f.fire_count FROM fires_2001 f JOIN max_count m ON f.fire_count = m.max_fire_count ORDER BY f.fire_count DESC, f.state ASC;	✅	Unreviewed query correctly filters to 2001, aggregates fires by state, finds the maximum count, and returns all states whose counts equal that maximum, properly handling ties. The Benchmark-Proposed Query returns only a single state due to LIMIT 1, so it fails to return all states in the event of a tie and does not fully answer the plural “states” question.	WITH fires_2001 AS (SELECT state, COUNT(*) AS fire_count FROM fires WHERE fire_year = 2001 GROUP BY state), max_count AS (SELECT MAX(fire_count) AS max_fire_count FROM fires_2001) SELECT f.state, f.fire_count FROM fires_2001 f JOIN max_count m ON f.fire_count = m.max_fire_count ORDER BY f.fire_count DESC, f.state ASC;	Unreviewed query and Reviewed query correctly compute per-state fire counts for 2001 and return all states whose counts equal the maximum, handling ties as the question implies.	✅	It is an example of incorrect benchamrk qurey

How many acres burned in fires in California each year between 2000 and 2005?	SELECT sum(FIRE_SIZE), FIRE_YEAR FROM Fires WHERE State = "CA" AND FIRE_YEAR BETWEEN 2000 AND 2005 GROUP BY FIRE_YEAR	SELECT fire_year AS year, SUM(fire_size) AS total_acres_burned FROM fires WHERE state = 'CA' AND fire_year BETWEEN 2000 AND 2005 AND fire_size IS NOT NULL AND fire_size > 0 GROUP BY fire_year ORDER BY fire_year ASC;	✅	Benchmark and unreviewed queries filter California and the years 2000–2005 (inclusive), group by year, and sum fire_size to yield total acres burned per year, answering the question.	WITH years(y) AS (SELECT 2000 UNION ALL SELECT 2001 UNION ALL SELECT 2002 UNION ALL SELECT 2003 UNION ALL SELECT 2004 UNION ALL SELECT 2005) SELECT y AS year, COALESCE(SUM(f.fire_size), 0) AS total_acres_burned FROM years LEFT JOIN fires f ON f.fire_year = y AND f.state = 'CA' GROUP BY y ORDER BY y ASC;	Reviewed query is correct because it generates every year from 2000 to 2005 and left-joins California fires, summing acres per year and using COALESCE to return 0 when no fires exist. This matches "each year between 2000 and 2005."	✅	

Which year had the largest number of fires?	SELECT FIRE_YEAR
	FROM Fires
	GROUP BY FIRE_YEAR
	ORDER BY count(*) DESC
	LIMIT 1
		WITH parsed_dates AS (
		  SELECT
		    CAST(EXTRACT(YEAR FROM (DATE '4713-11-24'::date + (discovery_date || ' days')::interval)) AS integer) AS year
		  FROM fires
		  WHERE discovery_date IS NOT NULL
		),
		year_counts AS (
		  SELECT year, COUNT(*) AS fire_count
		  FROM parsed_dates
		  GROUP BY year
		)
		SELECT year, fire_count
		FROM year_counts
		ORDER BY fire_count DESC, year ASC
		LIMIT 1;
			❌	The Benchmark-Proposed Query correctly groups by FIRE_YEAR and orders by the number of fires descending, returning the year with the highest count. This directly answers the question.
				Unreviewed query attempts to derive the year from discovery_date using a Julian-day conversion but uses the wrong epoch (DATE '4713-11-24' instead of DATE '4713-11-24 BC'), producing years offset by ~9,400 years. Even if the grouping counts align, the reported year is incorrect. It also assumes discovery_date is a Julian day number; if it's already a date, the expression would fail.
					SELECT fire_year, COUNT(*) AS fire_count
					FROM fires
					WHERE fire_year IS NOT NULL
					GROUP BY fire_year
					ORDER BY fire_count DESC, fire_year ASC
					LIMIT 1;
						Reviewer and benchamark correctly count fires per fire_year and select the year with the maximum; Reviewer also excludes NULLs and provides deterministic tie-breaking.	✅	

Show all fires caused by campfires in Texas.	SELECT *
	FROM Fires
	WHERE State = "TX" AND STAT_CAUSE_DESCR LIKE "Campfire"
		SELECT
		  county AS county,
		  discovery_date AS discovery_date,
		  fire_size AS fire_size,
		  fire_size_class AS fire_size_class,
		  latitude AS latitude,
		  longitude AS longitude
		FROM fires
		WHERE stat_cause_descr = 'Campfire'
		  AND state = 'TX';
			✅	Unreviewed query correctly filters for fires with stat_cause_descr = 'Campfire' in state = 'TX', which directly answers the question.	SELECT
					  county AS county,
					  discovery_date AS discovery_date,
					  fire_size AS fire_size,
					  fire_size_class AS fire_size_class,
					  latitude AS latitude,
					  longitude AS longitude
					FROM fires
					WHERE stat_cause_descr = 'Campfire'
					  AND state = 'TX';
						System-generateds correctly filter fires where stat_cause_descr = 'Campfire' and state = 'TX'. Returning a subset of columns still shows all qualifying fires. Reviewer didn't make any changes.	✅	

Who was responsible for the land of the biggest fire in Oregon in 2015?	SELECT OWNER_DESCR FROM FIres WHERE State = "OR" AND FIRE_YEAR = 2015 ORDER BY FIRE_SIZE DESC LIMIT 1	WITH or_2015 AS (SELECT state, fire_year, fire_size, owner_descr FROM fires WHERE state = 'OR' AND fire_year = 2015 AND fire_size IS NOT NULL), ranked AS (SELECT owner_descr AS responsible_party, fire_size AS max_fire_size, state, fire_year, RANK() OVER (ORDER BY fire_size DESC) AS rnk FROM or_2015) SELECT responsible_party FROM ranked WHERE rnk = 1 LIMIT 1;	✅	Unreviewed query correctly filters Oregon in 2015, excludes NULL fire sizes, ranks by fire_size to identify the largest fire, and returns the land-responsible party (owner_descr). This reliably answers the question even in databases where NULLs could otherwise interfere.	WITH or_2015 AS (SELECT state, fire_year, fire_size, owner_descr FROM fires WHERE state = 'OR' AND fire_year = 2015 AND fire_size IS NOT NULL), ranked AS (SELECT owner_descr AS responsible_party, fire_size AS max_fire_size, state, fire_year, RANK() OVER (ORDER BY fire_size DESC) AS rnk FROM or_2015) SELECT responsible_party FROM ranked WHERE rnk = 1 LIMIT 1;	System-generateds are correct: They restrict to Oregon in 2015, exclude NULL fire sizes, rank fires by size descending, and return the owner_descr of the largest fire (rnk = 1), which answers “who was responsible for the land” of the biggest fire.	✅	

Which counties in Washington had fires in 2012?	SELECT COUNTY FROM Fires WHERE State = "WA" AND FIRE_YEAR = 2012	SELECT DISTINCT county AS county FROM fires WHERE state = 'WA' AND fire_year = 2012 ORDER BY county;	✅	Unreviewed query is correct because it filters to Washington (WA), the year 2012, and returns a distinct list of counties, matching the intent to list which counties had fires. The Benchmark-Proposed Query is incorrect because it can return duplicate county rows (no DISTINCT)	SELECT DISTINCT county AS county FROM fires WHERE state = 'WA' AND fire_year = 2012 ORDER BY county;	System-generateds correctly return a distinct list of WA counties with fires in 2012.	✅		
How many arson-related fires were there in 2010?	SELECT count(*) FROM Fires WHERE FIRE_YEAR = 2010 AND STAT_CAUSE_DESCR LIKE "%Arson%"	SELECT COUNT(*) AS arson_fire_count_2010 FROM fires WHERE fire_year = 2010 AND LOWER(stat_cause_descr) = 'arson';	✅	Unreviewed query is correct: it counts fires in 2010 whose cause is exactly Arson, using a case-insensitive comparison, which matches the standard enumerated cause value. The Benchmark-Proposed Query is incorrect: it uses a substring match that can include unintended values (e.g., “Non-Arson”) and relies on double-quoted string literals, which are invalid in many SQL dialects; it may also be case-sensitive and miss matches depending on the DB.	SELECT COUNT(*) AS arson_fire_count_2010 FROM fires WHERE fire_year = 2010 AND LOWER(stat_cause_descr) = 'arson';	Unreviewed query and Reviewed query correctly count 2010 fires with an arson cause using a case-insensitive exact match on the standardized cause descriptor. Benchmark query uses a partial match with LIKE and double-quoted string literals, which is DB-dependent and may be case-sensitive or over/undercount compared to the exact cause category.	✅		

How many years of data are recorded in this database?	SELECT count(DISTINCT FIRE_YEAR) FROM Fires	WITH parsed_dates AS (SELECT discovery_date, NULLIF(TRIM(cont_date), '') AS cont_date_text FROM fires), bounds AS (SELECT MIN(discovery_date) AS min_disc_jd, MAX(CASE WHEN cont_date_text ~ '^[0-9]+(\.[0-9]+)?$' THEN cont_date_text::double precision ELSE NULL END) AS max_cont_jd FROM parsed_dates), year_bounds AS (SELECT EXTRACT(YEAR FROM to_timestamp((min_disc_jd - 2440587.5) * 86400))::int AS start_year, EXTRACT(YEAR FROM to_timestamp((max_cont_jd - 2440587.5) * 86400))::int AS end_year FROM bounds WHERE min_disc_jd IS NOT NULL AND max_cont_jd IS NOT NULL) SELECT GREATEST(end_year - start_year + 1, 0) AS years_of_data FROM year_bounds;	❌	The Benchmark-Proposed Query correctly counts the number of distinct years represented in the data (COUNT(DISTINCT FIRE_YEAR)), which directly answers “How many years of data are recorded?” Unreviewed query measures the inclusive span from the earliest discovery year to the latest containment year after converting Julian dates. This returns the calendar coverage, not the count of years with data. It can overstate the result if years are missing or if a fire spans New Year, and may return no result if cont_date is null/non-numeric.	SELECT COUNT(DISTINCT fire_year) AS years_of_data FROM fires;	Benchamark and reviewer correctly count the distinct FIRE_YEAR values, which directly answers how many years are represented in the data. Unreviewed query measures the inclusive span from the earliest discovery_date to the latest containment date, assuming continuous coverage and potentially counting years with no data; it answers a different question and can be inaccurate due to date parsing and span logic.	✅	

Average date of year that fire was discovered from 2000~2004?	SELECT avg(DISCOVERY_DATE) FROM Fires where FIRE_YEAR BETWEEN 2000 AND 2004	WITH fires_with_year AS (SELECT discovery_doy, EXTRACT(YEAR FROM (to_timestamp((discovery_date - 2440587.5) * 86400) AT TIME ZONE 'UTC'))::int AS yr FROM fires) SELECT AVG(discovery_doy)::float AS avg_discovery_doy FROM fires_with_year WHERE yr BETWEEN 2000 AND 2004;	✅	Unreviewed query correctly computes the average day-of-year (discovery_doy) for fires whose discovery year is between 2000 and 2004, matching the intent of "Average date of year." The Benchmark-Proposed Query averages the DISCOVERY_DATE itself, yielding an average absolute date/timestamp rather than an average day-of-year, so it does not answer the question.	WITH fires_with_year AS (SELECT discovery_doy, EXTRACT(YEAR FROM (to_timestamp((discovery_date - 2440587.5) * 86400) AT TIME ZONE 'UTC'))::int AS yr FROM fires) SELECT AVG(discovery_doy)::float AS avg_discovery_doy FROM fires_with_year WHERE yr BETWEEN 2000 AND 2004;	Reviewed query correctly computes the average of discovery_doy (day-of-year) and filters records by the discovery year (2000–2004), matching the intent: average day-of-year of discovery in that period.	✅	
What's the most common cause of the fire (code) in the database?	SELECT STAT_CAUSE_CODE FROM Fires GROUP BY STAT_CAUSE_CODE ORDER BY count(*) DESC LIMIT 1	WITH cause_counts AS (SELECT stat_cause_code AS cause_code, COUNT(*) AS cause_count FROM fires WHERE stat_cause_code IS NOT NULL GROUP BY stat_cause_code) SELECT cause_code, cause_count FROM cause_counts ORDER BY cause_count DESC, cause_code LIMIT 1;	✅	Unreviewed query is correct because it counts occurrences per cause code, excludes NULLs (which aren't valid codes), orders by frequency, and returns the top code (deterministically resolving ties). The Benchmark-Proposed Query is incorrect because it includes NULL in the grouping and can return NULL as the "most common cause code" if missing values are most frequent; it also lacks deterministic tie-breaking.	WITH cause_counts AS (SELECT stat_cause_code AS cause_code, COUNT(*) AS cause_count FROM fires WHERE stat_cause_code IS NOT NULL GROUP BY stat_cause_code) SELECT cause_code, cause_count FROM cause_counts ORDER BY cause_count DESC, cause_code LIMIT 1;	System-generated queries correctly return the most frequent non-null stat_cause_code, which matches the intent to find the most common cause code.	✅	

What’s the year that have the largest acres in the fire area?	SELECT FIRE_YEAR FROM Fires ORDER BY FIRE_SIZE DESC LIMIT 1	WITH parsed_dates AS (SELECT (to_timestamp((discovery_date - 2440587.5) * 86400) AT TIME ZONE 'UTC')::date AS discovery_dt, fire_size FROM fires WHERE discovery_date IS NOT NULL AND fire_size IS NOT NULL), yearly AS (SELECT EXTRACT(YEAR FROM discovery_dt)::int AS discovery_year, SUM(fire_size) AS total_acres FROM parsed_dates GROUP BY EXTRACT(YEAR FROM discovery_dt)) SELECT discovery_year, total_acres FROM yearly ORDER BY total_acres DESC LIMIT 1;	✅	Unreviewed query correctly aggregates fire_size (acres) by year (derived from discovery_date via proper Julian-to-timestamp conversion), sums acres per year, and returns the year with the highest total burned acres. This matches the intent: the year with the largest total acres burned. The Benchmark-Proposed Query is incorrect because it orders by FIRE_SIZE and returns the FIRE_YEAR of the single largest fire event, not the year with the largest total acres across all fires. It performs no aggregation.	WITH parsed_dates AS (SELECT (to_timestamp((discovery_date - 2440587.5) * 86400) AT TIME ZONE 'UTC')::date AS discovery_dt, fire_size FROM fires WHERE discovery_date IS NOT NULL AND fire_size IS NOT NULL), yearly AS (SELECT EXTRACT(YEAR FROM discovery_dt)::int AS discovery_year, SUM(fire_size) AS total_acres FROM parsed_dates GROUP BY EXTRACT(YEAR FROM discovery_dt)) SELECT discovery_year, total_acres FROM yearly ORDER BY total_acres DESC LIMIT 1;	Reviewer didn't make any changes so it is same as unreviewed and is correct.	✅	

How many missing / not specified name of the owner are there in the database?	SELECT count(*) FROM Fires WHERE OWNER_DESCR = "MISSING/NOT SPECIFIED"	SELECT 0 AS missing_owner_count;	❌	The Benchmark-Proposed Query correctly counts rows where the owner name is recorded as the sentinel value "MISSING/NOT SPECIFIED," matching the question's intent. Unreviewed query returns a hardcoded 0 and does not query the data, so it cannot correctly answer the question.	Reviewer Faild	Reviewed query is not a valid SQL query and does not execute.	❌	

What's the county that contains most recorded burns in 2000 according to the database?	SELECT COUNTY FROM Fires GROUP BY COUNTY ORDER BY count(*)	WITH burns_2000 AS (
		  SELECT county
		  FROM fires
		  WHERE fire_year = 2000
		),
		counts AS (
		  SELECT county, COUNT(*) AS burn_count
		  FROM burns_2000
		  GROUP BY county
		)
		SELECT county
		FROM counts
		ORDER BY burn_count DESC
		LIMIT 1;
			✅	Unreviewed query filters to 2000, counts burns per county, orders by count descending, and returns the top county (LIMIT 1), matching the question.
				The Benchmark-Proposed Query doesn't filter for 2000, orders ascending by count, and returns all counties (no LIMIT), so it doesn't answer the question.
					WITH burns_2000 AS (
					  SELECT county
					  FROM fires
					  WHERE fire_year = 2000
					),
					counts AS (
					  SELECT county, COUNT(*) AS burn_count
					  FROM burns_2000
					  GROUP BY county
					)
					SELECT county
					FROM counts
					ORDER BY burn_count DESC
					LIMIT 1;
						System generated queries correctly filter to 2000, count burns per county, sort by count descending, and return the top county.	✅	It is an example of incorrect benchamrk qurey

How many distinct cause of the fire descriptions are there in the database?	SELECT count(DISTINCT STAT_CAUSE_DESCR) FROM Fires	SELECT COUNT(DISTINCT stat_cause_descr) AS distinct_cause_count FROM fires;	✅	Benchamrak and unreviewed queries compute COUNT(DISTINCT STAT_CAUSE_DESCR) from the fires table, which returns the number of unique cause-of-fire descriptions. Differences in casing and the presence of an alias do not affect the result.	SELECT COUNT(DISTINCT stat_cause_descr) AS distinct_cause_count FROM fires;	All three correctly compute the number of distinct cause-of-fire descriptions via COUNT(DISTINCT stat_cause_descr/STAT_CAUSE_DESCR) on the fires table; casing and aliasing differences don't change the result. Reviewer didn't make any changes.	✅	

Sum of the acres burned in NC state in 2006?	SELECT sum(FIRE_SIZE) FROM Fires WHERE State = "NY" and FIRE_YEAR = "2006"	WITH jd_bounds AS (SELECT -- Julian Day Numbers for 2006-01-01 and 2007-01-01 date_part('julian', DATE '2006-01-01')::double precision AS jd_start, date_part('julian', DATE '2007-01-01')::double precision AS jd_end), filtered AS (SELECT fire_size FROM fires, jd_bounds WHERE state = 'NC' AND discovery_date >= jd_bounds.jd_start AND discovery_date < jd_bounds.jd_end) SELECT SUM(fire_size) AS total_acres_burned_2006_nc FROM filtered;	✅	Unreviewed query correctly filters NC, restricts to calendar year 2006 via discovery_date bounds, and sums FIRE_SIZE (acres). The Benchmark-Proposed Query is incorrect because it filters for NY instead of NC; it also uses double quotes for string literals, which is invalid in many SQL dialects.	WITH jd_bounds AS (SELECT -- Julian Day Numbers for 2006-01-01 and 2007-01-01 date_part('julian', DATE '2006-01-01')::double precision AS jd_start, date_part('julian', DATE '2007-01-01')::double precision AS jd_end), filtered AS (SELECT fire_size FROM fires, jd_bounds WHERE state = 'NC' AND discovery_date >= jd_bounds.jd_start AND discovery_date < jd_bounds.jd_end) SELECT SUM(fire_size) AS total_acres_burned_2006_nc FROM filtered;	Reviewer didn't make any changes	✅	It is an example of incorrect benchamrk qurey

What was the cause of the largest wildfire in Utah in the year 1997?	SELECT * FROM Fires WHERE State = "UT" AND FIRE_YEAR = 1997 ORDER BY FIRE_SIZE DESC LIMIT 1	SELECT stat_cause_descr AS cause FROM fires WHERE state = 'UT' AND fire_year = 1997 ORDER BY fire_size DESC LIMIT 1;	✅	Unreviewed query filters Utah and 1997, orders by fire_size descending to get the largest fire, and returns stat_cause_descr as the cause, which directly answers the question. The Benchmark-Proposed Query returns the entire row (*) instead of just the cause, uses double quotes for the string "UT" (invalid in standard SQL), and is presented inside a list literal, making it syntactically incorrect.	SELECT stat_cause_descr AS cause FROM fires WHERE state = 'UT' AND fire_year = 1997 ORDER BY fire_size DESC LIMIT 1;	Reviewed query filters to Utah in 1997, orders by fire_size descending to pick the largest fire, and returns the cause (stat_cause_descr), matching the question exactly.	✅	
How many wildfires in Gloucester county have been larger than 10 acres?	SELECT count(*) FROM Fires WHERE COUNTY = "Gloucester" AND FIRE_SIZE > 10	SELECT COUNT(*) AS wildfire_count FROM fires WHERE county = '147' AND fire_size > 10;	❌	Ueviewed query filters county by '147' instead of the name 'Gloucester', so it won’t answer the question unless the column stores county codes (not indicated).	SELECT COUNT(*) AS wildfire_count FROM fires WHERE county = 'Gloucester' AND fire_size > 10;	Reviewed query correctly counts rows in the fires table where county = 'Gloucester' and fire_size > 10, using proper string literal quoting.	✅	This example shows that the data is vague since int the county column there are both county names and codes.

Which state experiences the most wildfires?	SELECT State FROM Fires GROUP BY State ORDER BY count(*) DESC LIMIT 1	WITH fires_by_state AS (
		  SELECT
		    state,
		    COUNT(*) AS total_fires
		  FROM fires
		  WHERE state IS NOT NULL
		  GROUP BY state
		),
		ranked AS (
		  SELECT
		    state,
		    total_fires,
		    RANK() OVER (ORDER BY total_fires DESC, state ASC) AS rnk
		  FROM fires_by_state
		)
		SELECT
		  state,
		  total_fires
		FROM ranked
		WHERE rnk = 1
		LIMIT 1;
			✅	Unreviewed query correctly counts fires per state, excludes NULL states, and returns the state with the highest count, which answers the question.
				The Benchmark-Proposed Query does not exclude NULL states, so it can incorrectly return NULL as the “state” if many records have missing state values; it also arbitrarily picks one row on ties without guaranteeing a valid state.
					WITH fires_by_state AS (
					  SELECT
					    state,
					    COUNT(*) AS total_fires
					  FROM fires
					  WHERE state IS NOT NULL
					  GROUP BY state
					),
					ranked AS (
					  SELECT
					    state,
					    total_fires,
					    RANK() OVER (ORDER BY total_fires DESC, state ASC) AS rnk
					  FROM fires_by_state
					)
					SELECT
					  state,
					  total_fires
					FROM ranked
					WHERE rnk = 1
					LIMIT 1;
						Unreviewed query and Reviewed query correctly count fires per state, exclude NULL states, and return the top state by total fires.	✅	

What is the leading cause of wildfires?	SELECT STAT_CAUSE_DESCR FROM Fires GROUP BY STAT_CAUSE_DESCR ORDER BY count(*) DESC LIMIT 1	WITH cause_counts AS (SELECT stat_cause_descr AS cause, COUNT(*) AS fire_count FROM fires WHERE stat_cause_descr IS NOT NULL GROUP BY stat_cause_descr) SELECT cause, fire_count FROM cause_counts ORDER BY fire_count DESC, cause ASC LIMIT 1;	✅	Unreviewed query correctly finds the most frequent cause by counting per stat_cause_descr, excludes NULLs (non-causes), orders by count descending, and returns the top result. This aligns with “leading cause” by frequency and avoids returning a NULL. The Benchmark-Proposed Query does not exclude NULL cause descriptions; if NULLs are frequent, it could return NULL as the “leading cause,” which is invalid. It also lacks deterministic tie-breaking.	WITH cause_counts AS (SELECT stat_cause_descr AS cause, COUNT(*) AS fire_count FROM fires WHERE stat_cause_descr IS NOT NULL GROUP BY stat_cause_descr) SELECT cause, fire_count FROM cause_counts ORDER BY fire_count DESC, cause ASC LIMIT 1;	System-generateds correctly compute the most frequent non-null stat_cause_descr and return the top cause (with its count), matching the question’s intent.	✅	

On what type of land (public or private) do more wildfires occur?	SELECT OWNER_DESCR FROM Fires GROUP BY OWNER_DESCR ORDER BY count(*) DESC LIMIT 1	WITH mapped AS (SELECT CASE WHEN owner_descr ILIKE ANY (ARRAY['USFS','NPS','BLM','BIA','FWS','BOR','DOD','DOE','OTHER FEDERAL', 'FEDERAL','STATE','COUNTY','CITY','MUNICIPAL','MUNICIPAL/LOCAL','TRIBAL', 'STATE OR PRIVATE','STATE/COUNTY','LOCAL','PUBLIC']) THEN 'Public' WHEN owner_descr ILIKE ANY (ARRAY['PRIVATE','PRIVATE NON-INDUSTRIAL','INDUSTRIAL','RAILROAD','UTILITY', 'PRIVATE INDUSTRIAL','COMMERCIAL']) THEN 'Private' WHEN owner_descr ILIKE 'MISSING/NOT SPECIFIED' THEN 'Unknown' ELSE 'Unknown' END AS land_type FROM fires), counts AS (SELECT land_type, COUNT(*) AS fire_count FROM mapped GROUP BY land_type), ranked AS (SELECT land_type, fire_count, RANK() OVER (ORDER BY CASE WHEN land_type IN ('Public','Private') THEN fire_count ELSE NULL END DESC NULLS LAST) AS rnk FROM counts) SELECT land_type, fire_count FROM ranked WHERE land_type IN ('Public','Private') ORDER BY fire_count DESC LIMIT 1;	✅	Unreviewed query maps many owner descriptors into the broader categories Public and Private, aggregates counts by these two types, and returns which has more fires—directly answering the question. The Benchmark-Proposed Query finds only the single most frequent OWNER_DESCR value and does not aggregate multiple public owners together; it may return a specific agency (e.g., USFS) rather than “Public” vs “Private,” so it does not answer the question.	WITH mapped AS (SELECT CASE WHEN owner_descr ILIKE ANY (ARRAY['USFS','NPS','BLM','BIA','FWS','BOR','DOD','DOE','OTHER FEDERAL', 'FEDERAL','STATE','COUNTY','CITY','MUNICIPAL','MUNICIPAL/LOCAL','TRIBAL', 'STATE OR PRIVATE']) THEN 'Public' WHEN owner_descr ILIKE ANY (ARRAY['PRIVATE','PRIVATE NON-INDUSTRIAL','INDUSTRIAL','RAILROAD','UTILITY', 'PRIVATE INDUSTRIAL','COMMERCIAL']) THEN 'Private' WHEN owner_descr ILIKE 'MISSING/NOT SPECIFIED' THEN 'Unknown' ELSE 'Unknown' END AS land_type FROM fires), counts AS (SELECT land_type, COUNT(*) AS fire_count FROM mapped GROUP BY land_type), ranked AS (SELECT land_type, fire_count, RANK() OVER (ORDER BY CASE WHEN land_type IN ('Public','Private') THEN fire_count ELSE NULL END DESC NULLS LAST) AS rnk FROM counts) SELECT land_type, fire_count FROM ranked WHERE land_type IN	Reviewed query correctly maps owner descriptors into Public vs Private (covering federal, state, county, city/municipal, local, and tribal), counts fires by these two types, and returns the higher count. It directly answers which type (Public or Private) has more wildfires.	✅	

						('Public','Private') ORDER BY fire_count DESC LIMIT 1;			

In 2014, how many wildfires were the result of mismanaged campfires?	SELECT count(*) FROM Fires WHERE STAT_CAUSE_DESCR LIKE "%Campfire%" AND FIRE_YEAR = 2014	WITH fires_with_dates AS (SELECT discovery_date, stat_cause_descr, to_timestamp((discovery_date - 2440587.5) * 86400) AT TIME ZONE 'UTC' AS discovery_ts FROM fires) SELECT COUNT(*) AS wildfire_count_2014_mismanaged_campfires FROM fires_with_dates WHERE EXTRACT(YEAR FROM discovery_ts) = 2014 AND stat_cause_descr = 'Campfire';	✅	Unreviewed query correctly filters fires discovered in 2014 by converting the Julian discovery_date to a UTC timestamp and requiring stat_cause_descr = 'Campfire', then counts them.	WITH fires_with_dates AS (SELECT discovery_date, stat_cause_descr, to_timestamp((discovery_date - 2440587.5) * 86400) AT TIME ZONE 'UTC' AS discovery_ts FROM fires) SELECT COUNT(*) AS wildfire_count_2014_mismanaged_campfires FROM fires_with_dates WHERE EXTRACT(YEAR FROM discovery_ts) = 2014 AND stat_cause_descr = 'Campfire';	System-generateds correctly count fires where stat_cause_descr = 'Campfire' and the discovery year is 2014 by properly converting the Julian discovery_date to a timestamp and extracting the year.	✅	

How many total acres of land in Texas have seen a wildfire in the decade between 2000-2010?	SELECT sum(FIRE_SIZE) FROM Fires WHERE State = "TX" AND FIRE_YEAR BETWEEN 2000 AND 2010	SELECT SUM(f.fire_size) AS total_acres_burned FROM fires AS f WHERE f.state = 'TX' AND f.discovery_date BETWEEN (DATE '2000-01-01' + 1721425.5)::date - DATE '0001-01-01' AND (DATE '2010-12-31' + 1721425.5)::date - DATE '0001-01-01';	✕	The Benchmark-Proposed Query correctly sums FIRE_SIZE for Texas and restricts to the requested years using FIRE_YEAR BETWEEN 2000 AND 2010 (inclusive). Minor dialect note: use single quotes for 'TX'. Unreviewed query mis-handles the date filter by attempting a nonstandard Julian-date conversion (adding 1,721,425.5 to a DATE, then casting/subtracting), which is dialect-dependent, drops the .5 day component, and can misplace the bounds; it's not a reliable or correct way to filter DISCOVERY_DATE.	Reviewer Faild	Reviewed query is not a valid SQL query.	✕	

Which state has the most number of fires being recorded?	SELECT State FROM Fires GROUP BY State ORDER BY count(*) DESC LIMIT 1	WITH fires_by_state AS (
		  SELECT
		    state AS state,
		    COUNT(*) AS fire_count
		  FROM fires
		  GROUP BY state
		),
		ranked AS (
		  SELECT
		    state,
		    fire_count,
		    ROW_NUMBER() OVER (ORDER BY fire_count DESC) AS rn
		  FROM fires_by_state
		)
		SELECT
		  state,
		  fire_count
		FROM ranked
		WHERE rn = 1;
			✅	Benchamrk and reviewed compute fire counts per state and return the state with the highest count (A via ROW_NUMBER over aggregated counts; B via GROUP BY with ORDER BY COUNT(*) DESC LIMIT 1).	WITH fires_by_state AS (
					  SELECT
					    state AS state,
					    COUNT(*) AS fire_count
					  FROM fires
					  GROUP BY state
					),
					ranked AS (
					  SELECT
					    state,
					    fire_count,
					    ROW_NUMBER() OVER (ORDER BY fire_count DESC) AS rn
					  FROM fires_by_state
					)
					SELECT
					  state,
					  fire_count
					FROM ranked
					WHERE rn = 1;
						All three queries correctly identify the state with the highest number of fire records. Unreviewed query orders states by COUNT(*) descending and returns the top state.
						None are incorrect relative to the question
							✅	

What is the most common reason that causes a fire?	SELECT STAT_CAUSE_DESCR FROM Fires GROUP BY STAT_CAUSE_DESCR ORDER BY count(*) DESC LIMIT 1	WITH cause_counts AS (SELECT stat_cause_descr AS cause, COUNT(*) AS cnt FROM fires GROUP BY stat_cause_descr) SELECT cause FROM cause_counts ORDER BY cnt DESC LIMIT 1;	✅	Both queries aggregate by STAT_CAUSE_DESCR, rank by COUNT(*) descending, and return the single most frequent cause, matching the question.	WITH cause_counts AS (SELECT stat_cause_descr AS cause, COUNT(*) AS cnt FROM fires GROUP BY stat_cause_descr) SELECT cause FROM cause_counts ORDER BY cnt DESC LIMIT 1;	All three queries correctly find the most frequent stat_cause_descr by grouping, counting, ordering by count descending, and limiting to one result.	✅	
What is the total area that has been burned until now?	SELECT sum(FIRE_SIZE) FROM Fires	WITH current_julian AS (SELECT EXTRACT(JULIAN FROM CURRENT_DATE)::real AS today_julian), filtered_fires AS (SELECT f.fire_size FROM fires f CROSS JOIN current_julian cj WHERE f.discovery_date IS NULL OR f.discovery_date <= cj.today_julian) SELECT SUM(fire_size)::double precision AS total_burned_area FROM filtered_fires;	❌	The Benchmark-Proposed Query correctly returns the total burned area by summing FIRE_SIZE over all records, which aligns with “until now” given the dataset records past fires. Unreviewed query is incorrect: it filters on discovery_date using EXTRACT(JULIAN FROM CURRENT_DATE), which is not valid in PostgreSQL and, even conceptually, mismatches types (comparing a Julian-derived number to a date/JDN). This would either error out or exclude most valid rows, leading to an incorrect total.	WITH current_julian AS (SELECT EXTRACT(JULIAN FROM CURRENT_DATE)::real AS today_julian), filtered_fires AS (SELECT f.fire_size FROM fires f CROSS JOIN current_julian cj WHERE f.discovery_date IS NULL OR f.discovery_date <= cj.today_julian) SELECT SUM(fire_size)::double precision AS total_burned_area FROM filtered_fires;	Unreviewed query and Reviewed query are identical and incorrect: they compare discovery_date to EXTRACT(JULIAN FROM CURRENT_DATE), which yields a day-of-year or otherwise non-comparable value, causing an apples-to-oranges comparison that can exclude most valid rows or fail by type. They also include rows with NULL discovery_date, potentially inflating results.	❌	

How many acres burned in 2000s and 1990s?	SELECT sum(FIRE_SIZE) FROM Fires WHERE STATE = "TX" AND FIRE_YEAR BETWEEN 2000 AND 2010 UNION SELECT sum(FIRE_SIZE) FROM Fires WHERE STATE = "TX" AND FIRE_YEAR BETWEEN 1990 AND 2000	WITH classified AS (SELECT CASE WHEN discovery_date BETWEEN 2447893 AND 2451544 THEN '1990s' WHEN discovery_date BETWEEN 2451545 AND 2455197 THEN '2000s' ELSE NULL END AS decade, fire_size FROM fires) SELECT decade, SUM(fire_size) AS total_acres FROM classified WHERE decade IS NOT NULL GROUP BY decade ORDER BY decade;	✅	Unreviewed query correctly sums fire_size (acres) for all fires and classifies records into the 1990s and 2000s using appropriate Julian day ranges, returning totals per decade as asked. The Benchmark-Proposed Query is incorrect because it limits to STATE = "TX" (not requested) and uses overlapping/incorrect year ranges (1990–2000 and 2000–2010), double-counting 2000 and including 2010 in the 2000s.	WITH classified AS (SELECT CASE WHEN discovery_date BETWEEN 2447893 AND 2451544 THEN '1990s' WHEN discovery_date BETWEEN 2451545 AND 2455197 THEN '2000s' ELSE NULL END AS decade, fire_size FROM fires) SELECT decade, SUM(fire_size) AS total_acres FROM classified WHERE decade IS NOT NULL GROUP BY decade ORDER BY decade;	Unreviewed query and Reviewed query are identical and correct: they classify fires into 1990s and 2000s using discovery_date Julian day ranges and sum fire_size (acres), returning one total per decade.	✅	

How many fire cases occur on the land without any owner?	SELECT count(*) FROM Fires WHERE OWNER_DESCR = "MISSING/NOT SPECIFIED"	SELECT COUNT(*) AS fire_cases_without_owner FROM fires WHERE owner_descr = 'MISSING/NOT SPECIFIED';	✅	Unreviewed query correctly counts rows where owner_descr equals 'MISSING/NOT SPECIFIED' using proper SQL string quoting, matching the intent.	SELECT COUNT(*) AS fire_cases_without_owner FROM fires WHERE owner_descr IS NULL OR TRIM(owner_descr) = '' OR UPPER(TRIM(owner_descr)) = 'MISSING/NOT SPECIFIED';	Reviewed query correctly captures all ownerless cases by counting NULLs, empty strings, and the explicit “MISSING/NOT SPECIFIED” value (case/whitespace insensitive), aligning with “without any owner.”	✅	

What is the latitudinal band that is most likely to experience wildfires in the USA?	SELECT LATITUDE FROM Fires GROUP BY LATITUDE ORDER BY count(*) DESC LIMIT 1	WITH valid_states AS (SELECT unnest(ARRAY['AL','AK','AZ','AR','CA','CO','CT','DE','FL','GA', 'HI','ID','IL','IN','IA','KS','KY','LA','ME','MD', 'MA','MI','MN','MS','MO','MT','NE','NV','NH','NJ', 'NM','NY','NC','ND','OH','OK','OR','PA','RI','SC', 'SD','TN','TX','UT','VT','VA','WA','WV','WI','WY','DC']) AS state_code), fires_filtered AS (SELECT f.latitude, f.state FROM fires f JOIN valid_states s ON f.state = s.state_code WHERE f.latitude BETWEEN 18 AND 72), binned AS (SELECT CONCAT((FLOOR(f.latitude / 5) * 5)::int, '-', ((FLOOR(f.latitude / 5) * 5) + 5)::int) AS lat_band FROM fires_filtered f), band_counts AS (SELECT lat_band, COUNT(*) AS fire_count, COUNT(*)::numeric / SUM(COUNT(*)) OVER () AS pct_of_total FROM binned GROUP BY lat_band) SELECT lat_band, fire_count, pct_of_total FROM band_counts ORDER BY fire_count DESC LIMIT 1;	✅	Unreviewed query correctly filters to U.S. states, bins latitudes into bands (5-degree ranges), counts fires per band, and returns the band with the highest frequency—matching the request for the “latitudinal band … most likely” in the USA. The Benchmark-Proposed Query is incorrect because it does not restrict to the USA and groups by exact latitude values (not bands), returning a single latitude rather than a latitudinal band.	WITH valid_states AS (SELECT unnest(ARRAY['AL','AK','AZ','AR','CA','CO','CT','DE','FL','GA', 'HI','ID','IL','IN','IA','KS','KY','LA','ME','MD', 'MA','MI','MN','MS','MO','MT','NE','NV','NH','NJ', 'NM','NY','NC','ND','OH','OK','OR','PA','RI','SC', 'SD','TN','TX','UT','VT','VA','WA','WV','WI','WY','DC']) AS state_code), fires_filtered AS (SELECT f.latitude, f.state FROM fires f JOIN valid_states s ON f.state = s.state_code WHERE f.latitude BETWEEN 18 AND 72), binned AS (SELECT CONCAT((FLOOR(f.latitude / 5) * 5)::int, '-', ((FLOOR(f.latitude / 5) * 5) + 5)::int) AS lat_band FROM fires_filtered f), band_counts AS (SELECT lat_band, COUNT(*) AS fire_count, COUNT(*)::numeric / SUM(COUNT(*)) OVER () AS pct_of_total FROM binned GROUP BY lat_band) SELECT lat_band, fire_count, pct_of_total FROM band_counts ORDER BY fire_count DESC LIMIT 1;	Unreviewed query and Reviewed query correctly filter to U.S. states, bin latitudes into 5-degree bands, and return the band with the highest wildfire count, which aligns with “latitudinal band most likely to experience wildfires in the USA.”	✅	

GeoNuclearData Database

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation	Annotator Comment
What are the top 10 countries with most number of operational plants?	SELECT Country FROM nuclear_power_plants WHERE Status = "Operational" GROUP BY Country ORDER BY count(Name) DESC LIMIT 10	WITH operational_plants AS (SELECT country FROM nuclear_power_plants WHERE status = 'Operational'), country_counts AS (SELECT country, COUNT(*) AS plant_count FROM operational_plants GROUP BY country) SELECT country, plant_count FROM country_counts ORDER BY plant_count DESC, country ASC LIMIT 10;	✅	Unreviewed query correctly filters operational plants, aggregates by country using COUNT(*), orders by the count descending, and limits to 10. It uses standard SQL string literals and returns the intended ranking (and the counts).	WITH operational_plants AS (SELECT country FROM nuclear_power_plants WHERE status = 'Operational'), country_counts AS (SELECT country, COUNT(*) AS plant_count FROM operational_plants GROUP BY country) SELECT country, plant_count FROM country_counts ORDER BY plant_count DESC, country ASC LIMIT 10;	Unreviewed query and Reviewed query correctly filter operational plants, aggregate counts per country, and return the top 10 with both country and count, using standard string literals and COUNT(*).	✅	

What are the planed nuclear power plants and their located countries?	SELECT Name, Country FROM nuclear_power_plants WHERE Status = "Planned"	SELECT
		  name AS plant_name,
		  country AS country
		FROM
		  nuclear_power_plants
		WHERE
		  status = 'planned';
			❌	Unreviewed query incorrectly selects 'planned' instead of 'Planned'	SELECT
					  name AS plant_name,
					  country AS country
					FROM
					  nuclear_power_plants
					WHERE
					  status = 'Planned';
						Reviewed query is correct: is selects the plant name and country from nuclear_power_plants where status = 'Planned', matching the question.	✅	

What's the most used nuclear reactor model?	SELECT ReactorModel FROM nuclear_power_plants GROUP BY ReactorModel ORDER BY count(*) DESC LIMIT 1	WITH model_usage AS (
		  SELECT
		    reactormodel AS model_name,
		    COUNT(*) AS plant_count
		  FROM nuclear_power_plants
		  WHERE reactormodel IS NOT NULL
		  GROUP BY reactormodel
		)
		SELECT
		  model_name,
		  plant_count
		FROM model_usage
		ORDER BY plant_count DESC, model_name ASC
		LIMIT 1;
			✅	Unreviewed query correctly counts usage per reactor model, excludes NULLs (which are not valid models), orders by highest count, and returns the top model (with a deterministic tie-break by name). The Benchmark-Proposed Query includes NULLs in the grouping; if NULL appears most frequently, it could incorrectly return NULL as the "most used" model. It also has nondeterministic behavior on ties.	WITH cleaned AS (
					  SELECT TRIM(reactormodel) AS model_name
					  FROM nuclear_power_plants
					  WHERE reactormodel IS NOT NULL AND TRIM(reactormodel) <> ''
					),
					model_usage AS (
					  SELECT model_name, COUNT(*) AS plant_count
					  FROM cleaned
					  GROUP BY model_name
					)
					SELECT model_name, plant_count
					FROM model_usage
					ORDER BY plant_count DESC, model_name ASC
					LIMIT 1;
						Reviewed query correctly computes the most used model by excluding NULLs and empty/whitespace-only values, normalizing with TRIM to avoid counting the same model under slightly different whitespace, aggregating counts, and selecting the top result with a deterministic tie-breaker.	✅	

Which country is Kaiga-4 built in?	SELECT Country FROM nuclear_power_plants WHERE Name = "Kaiga-4"	SELECT country AS country FROM nuclear_power_plants WHERE name = 'Kaiga-4';	✅	Unreviewed query correctly filters by name using a proper string literal ('Kaiga-4') and returns the country column.	SELECT country AS country FROM nuclear_power_plants WHERE name = 'Kaiga-4';	System-generated queries use a proper string literal ('Kaiga-4') to filter the name and select the country, directly answering the question.	✅	
How many PHWR are there today?	SELECT count(*) FROM nuclear_power_plants WHERE ReactorType = "PHWR"	SELECT COUNT(id) AS phwr_count FROM nuclear_power_plants WHERE reactortype = 'PHWR' AND status = 'Operational';	✅	Unreviewed query filters to status = 'Operational', which aligns with “today” (currently active PHWR reactors), and counts only those. The Benchmark-Proposed Query counts all PHWR entries regardless of status (including shutdown/under construction), so it doesn’t reflect “today.” Additionally, it may use nonstandard quoting for the string literal in some SQL dialects.	SELECT COUNT(id) AS phwr_count FROM nuclear_power_plants WHERE reactortype = 'PHWR' AND status = 'Operational';	Reviewer didn't make any changes. Unreviewed query and Reviewed query correctly count PHWR reactors with status = 'Operational', matching "today." Unreviewed query counts all PHWR regardless of status, so it overcounts and does not reflect current/operational reactors.	✅	

How many kinds of nuclear reactor model in the world?	SELECT count(DISTINCT ReactorModel) FROM nuclear_power_plants	SELECT COUNT(DISTINCT reactormodel) AS reactor_model_count FROM nuclear_power_plants;	✅	Benchamrak-proposed and Unreviewed query compute the number of distinct ReactorModel values in the nuclear_power_plants table, which answers the question about how many kinds of reactor models exist globally (assuming the table covers worldwide plants).	SELECT COUNT(DISTINCT reactormodel) AS reactor_model_count FROM nuclear_power_plants;	Reviewer didn't make any changes. All three queries correctly count the number of distinct reactor models in the nuclear_power_plants table.	✅	

How many nuclear power plants were shut down now?	SELECT count(*) FROM nuclear_power_plants WHERE Status = "Shutdown"	WITH normalized AS (
		  SELECT
		    id,
		    name,
		    status,
		    NULLIF(operationalto, '')::date AS operational_to_date
		  FROM nuclear_power_plants
		),
		shutdown_plants AS (
		  SELECT
		    id,
		    name
		  FROM normalized
		  WHERE
		    (operational_to_date IS NOT NULL AND operational_to_date <= CURRENT_DATE)
		    OR status IN ('Shutdown')
		)
		SELECT COUNT(*) AS shut_down_count
		FROM shutdown_plants;
			✗	The Benchmark-Proposed Query and System-Generated Query are incorrect because they only count rows with Status exactly equal to Shutdown, ignoring other shut-down statuses and plants whose operational end date has passed.	WITH normalized AS (
					  SELECT
					    id,
					    name,
					    status,
					    NULLIF(operationalto, '')::date AS operational_to_date
					  FROM nuclear_power_plants
					),
					shutdown_plants AS (
					  SELECT
					    id,
					    name
					  FROM normalized
					  WHERE
					    (operational_to_date IS NOT NULL AND operational_to_date >= CURRENT_DATE)
					    OR status IN ('Shutdown', 'Decommissioned', 'Permanent Shutdown')
					)
					SELECT COUNT(*) AS shut_down_count
					FROM shutdown_plants;
						This query is incorrect because it checks operational_to_date >= CURRENT_DATE instead of <= CURRENT_DATE	✗	

How many countries have at least 3 nuclear power plants?	SELECT Country FROM nuclear_power_plants Group BY Country HAVING count(Name) > 3	WITH plant_counts AS (SELECT npp.country, COUNT(*) AS plant_count FROM nuclear_power_plants AS npp GROUP BY npp.country), countries_with_3plus AS (SELECT pc.country FROM plant_counts AS pc WHERE pc.plant_count >= 3) SELECT COUNT(*) AS num_countries_with_at_least_3_plants FROM countries_with_3plus;	✅	Unreviewed query groups by country, filters countries with plant_count >= 3, and returns the number of such countries answering "How many countries have at least 3 nuclear power plants?" The Benchmark-Proposed Query is incorrect because it uses > 3 (excludes countries with exactly 3), returns a list of countries instead of a count, and counts Name (which could exclude nulls).	WITH plant_counts AS (SELECT npp.country, COUNT(*) AS plant_count FROM nuclear_power_plants AS npp GROUP BY npp.country), countries_with_3plus AS (SELECT pc.country FROM plant_counts AS pc WHERE pc.plant_count >= 3) SELECT COUNT(*) AS num_countries_with_at_least_3_plants FROM countries_with_3plus;	Reviewer didn't make any changes. Unreviewed query and Reviewed query correctly count countries with at least 3 plants by aggregating per country, filtering with >= 3, then counting those countries. Unreviewed query is incorrect because it uses > 3 (not "at least 3") and returns a list of countries instead of the number of countries.	✅	Good example of benchmark incorrect query

Which country has the most nuclear power plants?	SELECT Country FROM nuclear_power_plants GROUP BY Country ORDER BY count(name) DESC LIMIT 1	WITH operational_plants AS (SELECT country FROM nuclear_power_plants WHERE status = 'Operational'), plant_counts AS (SELECT country, COUNT(*) AS plant_count FROM operational_plants GROUP BY country) SELECT country AS country, plant_count AS plant_count FROM plant_counts ORDER BY plant_count DESC LIMIT 1;	✅	Unreviewed query is correct because it interprets "has" as currently possessing and filters to status = 'Operational', then finds the country with the highest count. The Benchmark-Proposed Query is incorrect because it counts all plants regardless of status (including decommissioned/planned), which does not align with the present-tense intent. It also uses COUNT(name), which could ignore rows with null names.	WITH operational_plants AS (SELECT country FROM nuclear_power_plants WHERE status = 'Operational'), plant_counts AS (SELECT country, COUNT(*) AS plant_count FROM operational_plants GROUP BY country) SELECT country AS country, plant_count AS plant_count FROM plant_counts ORDER BY plant_count DESC LIMIT 1;	Unreviewed query (and identical Reviewed query) correctly interpret “has” as currently operational, filter status = 'Operational', aggregate by country, and return the top country.	✅	
Which country has only one nuclear power plants?	SELECT Country FROM nuclear_power_plants GROUP BY Country HAVING count(Name) = 1	SELECT country, COUNT(*) AS plant_count FROM nuclear_power_plant GROUP BY country HAVING COUNT(*) = 1;	❌	Unreviewed query references an incorrect table name (nuclear_power_plant vs nuclear_power_plants), so it wouldn’t answer the question.	SELECT country, COUNT(*) AS plant_count FROM nuclear_power_plants GROUP BY country HAVING COUNT(*) = 1;	Reviewed query correctly returns countries with exactly one row (plant) in the table.	✅	

Which country has the most capacities of nuclear power plants?	SELECT Country FROM nuclear_power_plants GROUP BY Country ORDER BY sum(Capacity) DESC LIMIT 1	WITH operational_plants AS (SELECT country, capacity::numeric AS capacity_numeric FROM nuclear_power_plants WHERE status = 'Operational'), country_totals AS (SELECT country, SUM(capacity_numeric) AS total_capacity FROM operational_plants GROUP BY country) SELECT country, total_capacity FROM country_totals ORDER BY total_capacity DESC LIMIT 1;	✅	Unreviewed query is correct because it sums capacities per country for plants that a country currently has (status = 'Operational'), casts capacity to numeric to ensure proper aggregation, orders by total capacity, and returns the top country. The Benchmark-Proposed Query is incorrect because it does not filter to operational plants, so it can include planned, under-construction, or retired capacities, which does not match the present-tense “has” in the question. It also relies on implicit typing for Capacity, which may fail if the column is not numeric.	WITH operational_plants AS (SELECT country, capacity::numeric AS capacity_numeric FROM nuclear_power_plants WHERE status = 'Operational'), country_totals AS (SELECT country, SUM(capacity_numeric) AS total_capacity FROM operational_plants GROUP BY country) SELECT country, total_capacity FROM country_totals ORDER BY total_capacity DESC LIMIT 1;	Reviewer didn't make any changes. They correctly sum capacity (casting to numeric) for operational plants by country, order by total capacity, and return the top country.	✅	

Which country has the least capacities of nuclear power plants?	SELECT Country FROM nuclear_power_plants GROUP BY Country ORDER BY sum(Capacity) LIMIT 1	WITH capacities AS (
		  SELECT
		    country,
		    COALESCE(NULLIF(TRIM(capacity), '')::numeric, 0) AS capacity_num
		  FROM nuclear_power_plants
		),
		totals AS (
		  SELECT
		    country,
		    SUM(capacity_num) AS total_capacity
		  FROM capacities
		  GROUP BY country
		),
		min_capacity AS (
		  SELECT MIN(total_capacity) AS min_total_capacity FROM totals
		)
		SELECT t.country, t.total_capacity
		FROM totals t
		JOIN min_capacity m ON t.total_capacity = m.min_total_capacity
		ORDER BY t.country;
			✅	Unreviewed query correctly sums capacities per country, safely casts/cleans values (handles blanks/NULLs as 0), then selects the country(ies) with the minimum total capacity, properly handling ties. The Benchmark-Proposed Query can fail or mis-rank results: it doesn't cast/clean capacity (errors if stored as text), may treat NULLs inconsistently (potentially picking a country with NULL sum), and arbitrarily returns one country in case of ties.	WITH capacities AS (
					  SELECT
					    country,
					    COALESCE(NULLIF(TRIM(capacity), '')::numeric, 0) AS capacity_num
					  FROM nuclear_power_plants
					),
					totals AS (
					  SELECT
					    country,
					    SUM(capacity_num) AS total_capacity
					  FROM capacities
					  GROUP BY country
					),
					min_capacity AS (
					  SELECT MIN(total_capacity) AS min_total_capacity FROM totals
					)
					SELECT t.country, t.total_capacity
					FROM totals t
					JOIN min_capacity m ON t.total_capacity = m.min_total_capacity
					ORDER BY t.country;
						Unreviewed query and Reviewed query correctly sum plant capacities by country, handle empty/NULL capacity values via safe casting, compute the minimum total, and return all countries tied for the least total capacity—fully answering the intent.	✅	

What are the top 3 countries which have the most nuclear power plants?	SELECT Country FROM nuclear_power_plants GROUP BY Country ORDER BY sum(Name) DESC LIMIT 3	SELECT country AS country, COUNT(id) AS plant_count FROM nuclear_power_plants GROUP BY country ORDER BY plant_count DESC LIMIT 3;	✅	Unreviewed query correctly counts plants per country (COUNT(id)), orders by that count descending, and limits to 3, matching “top 3 countries with the most nuclear power plants.” The Benchmark-Proposed Query orders by SUM(Name), which is inappropriate (and likely invalid) since Name is non-numeric and it doesn’t count plants, so it cannot determine which countries have the most plants.	SELECT country AS country, COUNT(id) AS plant_count FROM nuclear_power_plants GROUP BY country ORDER BY plant_count DESC LIMIT 3;	Unreviewed query and Reviewed query correctly count plants per country and return the top 3 by count.	✅	Good example of benchmark incorrect query

Which country has the most nuclear station?	SELECT Country FROM nuclear_power_plants GROUP BY Country ORDER BY count(Name) DESC LIMIT 1	WITH filtered_plants AS (
		  SELECT country
		  FROM nuclear_power_plants
		  WHERE status IN ('Operational', 'Under Construction')
		),
		country_counts AS (
		  SELECT country, COUNT(*) AS station_count
		  FROM filtered_plants
		  GROUP BY country
		),
		ranked AS (
		  SELECT country, station_count, RANK() OVER (ORDER BY station_count DESC) AS rnk
		  FROM country_counts
		)
		SELECT country, station_count
		FROM ranked
		WHERE rnk = 1;
			❌	Unreviewed query is incorrect because it includes stations “Under Construction,” which inflates counts beyond what a country currently has. It also returns multiple rows on ties and includes station_count, deviating from the request for just the country. Benchmark query is incorrect because it counts all plants regardless of status (e.g., decommissioned/shut down), which can misrepresent which country currently “has” the most stations. It also arbitrarily breaks ties with LIMIT 1.	WITH country_counts AS (
					  SELECT country, COUNT(*) AS station_count
					  FROM nuclear_power_plants
					  WHERE status = 'Operational'
					  GROUP BY country
					),
					ranked AS (
					  SELECT country, ROW_NUMBER() OVER (ORDER BY station_count DESC, country ASC) AS rn
					  FROM country_counts
					)
					SELECT country
					FROM ranked
					WHERE rn = 1;
						Reviewed query is correct because it counts only currently operational nuclear stations per country and returns the single country with the highest count, matching the likely intent of “has the most” (current presence).	✅		

Which county has the most nuclear station shut down?	SELECT Country FROM nuclear_power_plants WHERE Status = "Shutdown" GROUP BY Country ORDER BY count(Name) DESC LIMIT 1	WITH shutdown_plants AS (
		  SELECT
		    county
		  FROM nuclear_power_plants
		  WHERE status IN ('Shut down', 'Decommissioned', 'Closed')
		)
		SELECT
		  county,
		  COUNT(*) AS shutdown_count
		FROM shutdown_plants
		GROUP BY country
		ORDER BY shutdown_count DESC
		FETCH FIRST 1 ROW ONLY;
			❌	Unreviewed query uses statuses 'Shut down', 'Decommissioned', and 'Closed', which expands beyond "Shutdown" and even omits the exact "Shutdown" label, likely excluding relevant records and including others not asked for. <u>Note: Here, the question is vague since 'Decommissioned' and 'Closed' also refer to shot down status.</u>
				The Benchmark question is incorrect because mention county (not country) that there is no data about counties in the table so their intent were country probably.
					WITH shutdown_plants AS (
					  SELECT
					    county
					  FROM nuclear_power_plants
					  WHERE status IN ('Shutdown')
					)
					SELECT
					  county,
					  COUNT(*) AS shutdown_count
					FROM shutdown_plants
					GROUP BY country
					ORDER BY shutdown_count DESC
					FETCH FIRST 1 ROW ONLY;
						Reviewed query correctly filters plants with status exactly "Shutdown", groups by county, and returns the county with the highest count, matching the question. Unreviewed query groups by Country (not county), so it answers a different question.	✅	

What nuclear station has the largest nuclear power plant capacity?	SELECT Name FROM nuclear_power_plants ORDER BY Capacity DESC LIMIT 1	WITH operational_plants AS (SELECT name AS plant_name, capacity FROM nuclear_power_plants WHERE status = 'Operational' AND capacity IS NOT NULL), cast_capacity AS (SELECT plant_name, capacity::numeric AS capacity_mw FROM operational_plants) SELECT plant_name, capacity_mw FROM cast_capacity ORDER BY capacity_mw DESC FETCH FIRST 1 ROW ONLY;	❌	The Benchmark-Proposed Query orders all plants by Capacity in descending order and returns the top Name, directly answering "which station has the largest capacity" without imposing extra constraints. Unreviewed query incorrectly restricts to status = 'Operational', which the question does not specify, potentially excluding the true largest-capacity station if it is non-operational. It also returns an extra column (capacity), which is unnecessary.	SELECT name AS station_name, SUM(capacity::numeric) AS total_capacity_mw FROM nuclear_power_plants WHERE capacity IS NOT NULL GROUP BY name ORDER BY total_capacity_mw DESC FETCH FIRST 1 ROW ONLY;	Reviewed query is incorrect because it uses SUM(capacity) grouped by name.	❌	
How many operating nuclear station in France?	SELECT count(*) FROM nuclear_power_plants WHERE Country = "France" and Status = "Operational"	SELECT COUNT(*) AS operating_plants_in_france FROM nuclear_power_plants WHERE country = 'France' AND status = 'Operational';	✅	Unreviewed query correctly counts operational nuclear power plants in France using standard SQL with proper single-quoted string literals and valid syntax.	SELECT COUNT(*) AS operating_plants_in_france FROM nuclear_power_plants WHERE country = 'France' AND status = 'Operational';	Unreviewed query and Reviewed query correctly count operational nuclear plants in France using proper string literals and filters.	✅	

How many nuclear station are under construction?	SELECT count(Name) FROM nuclear_power_plants WHERE Status = "Under Construction"	SELECT COUNT(*) AS under_construction_count FROM nuclear_power_plants WHERE status = 'Under Construction';	✅	Unreviewed query is correct because it counts all rows with status = 'Under Construction' using COUNT(*), which includes all matching stations regardless of NULLs, and uses standard single-quoted string literals.	SELECT COUNT(*) AS under_construction_count FROM nuclear_power_plants WHERE status = 'Under Construction';	Reviewed query correctly counts all rows where status = 'Under Construction' using COUNT(*), which includes all matching rows regardless of NULLs in other columns.	✅	
What nuclear reactor model is the most popular?	SELECT ReactorModel FROM nuclear_power_plants GROUP BY ReactorModel ORDER BY count(*) DESC LIMIT 1	WITH model_counts AS (SELECT reactormodel AS reactor_model, COUNT(*) AS count_of_plants FROM nuclear_power_plants WHERE status = 'Operational' GROUP BY reactormodel) SELECT reactor_model, count_of_plants FROM model_counts ORDER BY count_of_plants DESC, reactor_model ASC LIMIT 1;	❌	The Benchmark-Proposed Query groups by ReactorModel and orders by COUNT(*) DESC to return the most frequent model across all records, matching the unqualified request for the “most popular” model. It also returns just the model name, which is what was asked. Unreviewed query restricts to status = 'Operational', adding an unstated assumption. It answers “most popular among operational plants,” which may differ from the overall most popular model and thus does not directly answer the question.	WITH model_counts AS (SELECT reactormodel AS reactor_model, COUNT(*) AS count_of_plants FROM nuclear_power_plants WHERE reactormodel IS NOT NULL GROUP BY reactormodel) SELECT reactor_model, count_of_plants FROM model_counts ORDER BY count_of_plants DESC, reactor_model ASC LIMIT 1;	Reviewed query is correct because it counts all non-null reactor models across the entire table, orders by frequency descending (with a deterministic tie-breaker), and returns the most frequent model, matching “most popular” without adding assumptions.	✅	

Which country is Chinon-A3 in?	SELECT Country FROM nuclear_power_plants WHERE Name = "Chinon-A3"	SELECT country AS country FROM nuclear_power_plants WHERE name = 'Chinon-A3';	✅	Unreviewed query correctly selects the country from nuclear_power_plants where the plant name equals the string 'Chinon-A3' using standard SQL string quoting. The Benchmark-Proposed Query uses double quotes around "Chinon-A3", which in standard SQL (e.g., PostgreSQL, SQL Server) denotes an identifier, not a string literal, causing an error or incorrect comparison.	SELECT country AS country FROM nuclear_power_plants WHERE name = 'Chinon-A3';	Reviewer didn't make any changes. Reviewed and unreviewed correctly filter name = 'Chinon-A3' using proper string literals and return the country.	✅	
What are operational nuclear power plants in Japan called?	SELECT Name FROM nuclear_power_plants where Status = "Operational" and Country = "Japan"	SELECT name AS plant_name FROM nuclear_power_plants WHERE country = 'Japan' AND status = 'Operational';	✅	Unreviewed query correctly filters nuclear_power_plants to country = 'Japan' and status = 'Operational' and returns the plant names, matching the question, using standard SQL string quoting.	SELECT name AS plant_name FROM nuclear_power_plants WHERE country = 'Japan' AND status = 'Operational';	System-generateds use standard SQL string literals and correctly filter Japanese plants with status 'Operational', returning their names.	✅	

Which country first started using nuclear power plant(s)?	SELECT Country FROM nuclear_power_plants ORDER BY OperationalFrom LIMIT 1	WITH operational_plants AS (SELECT country, to_date(operationalfrom, 'YYYY-MM-DD') AS operational_from_date FROM nuclear_power_plants WHERE status = 'Operational' AND operationalfrom IS NOT NULL), country_first_dates AS (SELECT country, MIN(operational_from_date) AS country_first_date FROM operational_plants GROUP BY country), earliest_country AS (SELECT country AS first_country, country_first_date AS first_operational_date, RANK() OVER (ORDER BY country_first_date ASC, country ASC) AS rnk FROM country_first_dates) SELECT first_country, first_operational_date FROM earliest_country WHERE rnk = 1 ORDER BY first_country;	✗	The Benchmark-Proposed Query correctly finds the country of the globally earliest OperationalFrom date, which directly answers which country first started using nuclear power plants. Unreviewed query is incorrect because it filters to status = 'Operational', excluding decommissioned/retired plants and thus can miss the true first adopter. It also ranks by (date, country), breaking ties arbitrarily instead of acknowledging co-earliest countries.	WITH operational_plants AS (SELECT country, to_date(operationalfrom, 'YYYY-MM-DD') AS operational_from_date FROM nuclear_power_plants WHERE status = 'Operational' AND operationalfrom IS NOT NULL), country_first_dates AS (SELECT country, MIN(operational_from_date) AS country_first_date FROM operational_plants GROUP BY country), earliest_country AS (SELECT country AS first_country, country_first_date AS first_operational_date, RANK() OVER (ORDER BY country_first_date ASC, country ASC) AS rnk FROM country_first_dates) SELECT first_country, first_operational_date FROM earliest_country WHERE rnk = 1 ORDER BY first_country;	Reviewed query is invalid for the same reason as unreviewed query.	✗	

How many nuclear power plants are in preparation to be used in Japan?	SELECT count(*) FROM nuclear_power_plants WHERE Country = "Japan" AND Status = "Under Construction"	SELECT COUNT(*) AS num_plants_in_preparation FROM nuclear_power_plants WHERE country = 'Japan' AND status IN ('Planned', 'Under Construction', 'Permitted', 'Site Prep');	✅	This question is vage. Unreviewed query interprets “in preparation” as encompassing multiple pre-operational phases (e.g., Planned, Permitted, Site Prep, Under Construction), thus counting all Japanese plants being prepared for future use. The Benchmark-Proposed Query only counts plants “Under Construction,” omitting other preparatory stages like Planned, Permitted, and Site Prep, thereby undercounting.	SELECT COUNT(*) AS num_plants_in_preparation FROM nuclear_power_plants WHERE country = 'Japan' AND status IN ('Planned', 'Under Construction', 'Permitted', 'Site Prep');	Reviewed query correctly interprets “in preparation to be used” as including pre-operational stages: Planned, Under Construction, Permitted, and Site Prep.	✅		

What is the current situation of the nuclear power plant in the United States with the maximum capacity?	SELECT Status FROM nuclear_power_plants WHERE Country = "United States" ORDER BY Capacity DESC LIMIT 1	WITH us_plants AS (
		  SELECT
		    name AS plant_name,
		    capacity,
		    status AS current_status
		  FROM nuclear_power_plants
		  WHERE country = 'United States'
		),
		ranked AS (
		  SELECT
		    plant_name,
		    capacity,
		    current_status,
		    ROW_NUMBER() OVER (
		      ORDER BY NULLIF(regexp_replace(capacity, '[^0-9\.]', '', 'g'), '')::numeric DESC NULLS LAST,
		               plant_name
		    ) AS rn
		  FROM us_plants
		)
		SELECT
		  plant_name,
		  capacity AS max_capacity,
		  current_status
		FROM ranked
		WHERE rn = 1
		LIMIT 1;
			✅	Unreviewed query correctly filters U.S. plants, determines the maximum capacity using a numeric sort even if capacity is stored as text with units, and returns the plant's current_status (the "current situation"). It is syntactically valid for Postgres (as implied by the use of regexp_replace and ::numeric).	WITH us_plants AS (
					  SELECT
					    name AS plant_name,
					    capacity,
					    status AS current_status
					  FROM nuclear_power_plants
					  WHERE country = 'United States'
					),
					ranked AS (
					  SELECT
					    plant_name,
					    capacity,
					    current_status,
					    ROW_NUMBER() OVER (
					      ORDER BY NULLIF(regexp_replace(capacity, '[^0-9]', '', 'g'), '')::numeric DESC NULLS LAST,
					               plant_name
					    ) AS rn
					  FROM us_plants
					)
					SELECT
					  plant_name,
					  capacity AS max_capacity,
					  current_status
					FROM ranked
					WHERE rn = 1
					LIMIT 1;
						Reviewed query correctly filters US plants and determines the maximum capacity by converting capacity strings to numeric while preserving decimals, then returns the current status (and related fields). It is robust to capacities stored with units/commas/decimals.
						Reviewed query strips all non-digits, removing decimal points and potentially altering values (e.g., 999.9 → 9999)
							✅		

What is the maximum capacity of existing pressurized water reactor?	SELECT max(Capacity) FROM nuclear_power_plants WHERE ReactorType = "PWR" and Status = "Operational"	SELECT MAX(capacity::numeric) AS max_capacity_pwr_existing FROM nuclear_power_plants WHERE reactortype = 'PWR' AND status = 'Operational' AND capacity IS NOT NULL;	✅	Unreviewed query correctly filters for PWR reactors with status 'Operational' (matching "existing"), uses proper string literals, and ensures a numeric comparison for MAX; the IS NOT NULL is redundant but harmless.	SELECT MAX(capacity::numeric) AS max_capacity_pwr_existing FROM nuclear_power_plants WHERE reactortype = 'PWR' AND status = 'Operational' AND capacity IS NOT NULL;	Unreviewed query and Reviewed query correctly filter for existing (status = 'Operational') PWR reactors and compute the numeric maximum capacity, safely casting capacity to numeric and excluding NULLs.	✅	

Which nation has the most nuclear plants under construction?	SELECT Country FROM nuclear_power_plants WHERE Status = "Under Construction" GROUP BY Country ORDER BY count(*) DESC LIMIT 1	WITH under_construction AS (SELECT country FROM nuclear_power_plants WHERE status = 'Under Construction'), counts AS (SELECT country, COUNT(*) AS plant_count FROM under_construction GROUP BY country), max_count AS (SELECT MAX(plant_count) AS max_plant_count FROM counts) SELECT c.country AS country, c.plant_count AS count_of_under_construction FROM counts c CROSS JOIN max_count m WHERE c.plant_count = m.max_plant_count ORDER BY c.plant_count DESC, c.country ASC;	✅	Unreviewed query correctly filters plants under construction, aggregates by country, finds the maximum count, and returns all countries tied for the maximum, which correctly answers the question (including ties) and uses proper string literals.	WITH under_construction AS (SELECT country FROM nuclear_power_plants WHERE status = 'Under Construction'), counts AS (SELECT country, COUNT(*) AS plant_count FROM under_construction GROUP BY country), max_count AS (SELECT MAX(plant_count) AS max_plant_count FROM counts) SELECT c.country AS country, c.plant_count AS count_of_under_construction FROM counts c CROSS JOIN max_count m WHERE c.plant_count = m.max_plant_count ORDER BY c.plant_count DESC, c.country ASC;	Reviewed query correctly filters status = 'Under Construction', counts plants by country, identifies the maximum, and returns all top country/countries (handles ties) using proper string literals.	✅	

Which country has the most under constriction power plants to date?	SELECT Country FROM nuclear_power_plants WHERE Status = "Under Construction" GROUP BY Country ORDER BY count(*) DESC LIMIT 1	WITH under_construction AS (
		  SELECT
		    country,
		    COUNT(id) AS under_construction_plant_count
		  FROM nuclear_power_plants
		  WHERE TRIM(status) = 'Planned'
		  GROUP BY country
		),
		max_count AS (
		  SELECT MAX(under_construction_plant_count) AS max_cnt
		  FROM under_construction
		)
		SELECT
		  u.country AS country,
		  u.under_construction_plant_count AS under_construction_plant_count
		FROM under_construction u
		JOIN max_count m ON u.under_construction_plant_count = m.max_cnt
		ORDER BY u.country;
			❌	Unreviewed query is incorrect because it filters status = 'Planned' instead of 'Under Construction'.	WITH under_construction AS (
					  SELECT
					    country,
					    COUNT(id) AS under_construction_plant_count
					  FROM nuclear_power_plants
					  WHERE TRIM(status) = 'Under Construction'
					  GROUP BY country
					),
					max_count AS (
					  SELECT MAX(under_construction_plant_count) AS max_cnt
					  FROM under_construction
					)
					SELECT
					  u.country AS country,
					
					u.under_construction_plant_count AS under_construction_plant_count
					FROM under_construction u
					JOIN max_count m ON u.under_construction_plant_count = m.max_cnt
					ORDER BY u.country;
						Reviewed query correctly counts under-construction plants per country, handles whitespace via TRIM, identifies the maximum, and returns all countries tied for the top count (appropriate for "has the most")	✅	

Where is the first BWR type power plant built and located?	SELECT Longitude, Latitude FROM nuclear_power_plants WHERE ReactorType = "BWR" ORDER BY ConstructionStartAt LIMIT 1	WITH bwr_plants AS (
		  SELECT
		    name AS plant_name,
		    country,
		    latitude,
		    longitude,
		    to_date(constructionstartat, 'YYYY-MM-DD') AS construction_start_date
		  FROM nuclear_power_plants
		  WHERE reactortype = 'BWR'
		    AND constructionstartat IS NOT NULL
		)
		SELECT
		  plant_name,
		  country,
		  latitude,
		  longitude
		FROM bwr_plants
		ORDER BY construction_start_date ASC, plant_name ASC
		LIMIT 1;
			✅	Correctly filters BWR plants, excludes NULL construction dates, converts the date for reliable ordering, and returns clear location info (country and coordinates) for the earliest-built BWR.	WITH bwr_plants AS (
					  SELECT
					    name AS plant_name,
					    country,
					    latitude,
					    longitude,
					
					to_date(constructionstartat, 'YYYY-MM-DD') AS construction_start_date
					  FROM nuclear_power_plants
					  WHERE reactortype = 'BWR'
					    AND constructionstartat IS NOT NULL
					)
					SELECT
					  plant_name,
					  country,
					  latitude,
					  longitude
					FROM bwr_plants
					ORDER BY construction_start_date ASC, plant_name ASC
					LIMIT 1;
						Reviewer didn't make any changes Unreviewed query and Reviewed query correctly identify the earliest BWR by ordering on a parsed construction start date, exclude NULL dates, and return meaningful location details (plant name, country, latitude, longitude).	✅	

Which country lead the total capacity of the power plants it held?	SELECT Country FROM nuclear_power_plants GROUP BY Country ORDER BY sum(Capacity) DESC LIMIT 1	WITH summed AS (
		  SELECT
		    country,
		    SUM(NULLIF(TRIM(capacity), ")::numeric) AS total_capacity
		  FROM nuclear_power_plants
		  GROUP BY country
		)
		SELECT
		  country,
		  total_capacity
		FROM summed
		ORDER BY total_capacity DESC
		LIMIT 1;
			✅	Unreviewed query correctly aggregates capacity by country, safely handles empty-string values by converting them to NULL, casts to numeric, orders by the summed capacity, and returns the top country (the leader).
				The Benchmark-Proposed Query assumes Capacity is already numeric and contains no empty strings; if Capacity is text or has blanks (common in such datasets), SUM(Capacity) will error or misaggregate, so it does not reliably answer the question.
					WITH summed AS (
					  SELECT
					    country,
					
					SUM(NULLIF(TRIM(capacity), ")::numeric) AS total_capacity
					  FROM nuclear_power_plants
					  GROUP BY country
					)
					SELECT
					  country,
					  total_capacity
					FROM summed
					ORDER BY total_capacity DESC
					LIMIT 1;
						Reviewer didn't make any changes. Unreviewed query (and identical Reviewed query) correctly sums capacity per country with trimming/casting to handle blank/text values, orders by total, and returns the top country.	✅	

What is the main source of the information for this table?	SELECT Source FROM nuclear_power_plants GROUP BY Source ORDER BY count(*) DESC LIMIT 1	WITH distinct_sources AS (SELECT DISTINCT source FROM nuclear_power_plants), ranked_sources AS (SELECT source, COUNT(*) OVER () AS total_sources, ROW_NUMBER() OVER (ORDER BY source) AS rn FROM distinct_sources) SELECT source AS main_source FROM ranked_sources WHERE total_sources = 1 OR source = 'WNA/IAEA' OR rn = 1;	❌	The Benchmark-Proposed Query correctly identifies the main source by selecting the most frequent Source value (GROUP BY Source, ORDER BY count(*) DESC, LIMIT 1), which aligns with the intent of "main source." Unreviewed query is incorrect because it doesn't consider frequency. It hard-codes a preference for 'WNA/IAEA' and otherwise returns the lexicographically first distinct source (rn = 1), which may not be the predominant source and can even return multiple rows in some cases.	SELECT source AS main_source FROM nuclear_power_plants GROUP BY source ORDER BY COUNT(*) DESC, source ASC LIMIT 1;	Reviewed query correctly identify the main source by grouping by source and selecting the one with the highest count (i.e., the predominant source).	✅	

Which reactor type has the largest average capacity?	SELECT ReactorType FROM nuclear_power_plants GROUP BY ReactorType ORDER BY avg(Capacity) DESC LIMIT 1	WITH typed_capacities AS (
		  SELECT
		    reactortype AS reactor_type,
		    NULLIF(TRIM(capacity), '')::numeric AS capacity_num
		  FROM nuclear_power_plants
		),
		averages AS (
		  SELECT
		    reactor_type,
		    AVG(capacity_num) AS avg_capacity
		  FROM typed_capacities
		  WHERE capacity_num IS NOT NULL
		  GROUP BY reactor_type
		)
		SELECT
		  reactor_type,
		  avg_capacity
		FROM averages
		ORDER BY avg_capacity DESC
		LIMIT 1;
			✅	Unreviewed query correctly computes the average by safely converting capacity values to numeric, trimming blanks, ignoring empty strings/nulls, grouping by reactor type, and returning the type with the highest average.
				The Benchmark-Proposed Query relies on avg(Capacity) without casting or cleaning; if Capacity is stored as text or contains empty strings, it can error or yield incorrect results, so it does not robustly answer the question.
					WITH typed_capacities AS (
					  SELECT
					    reactortype AS reactor_type,
					    NULLIF(TRIM(capacity), '')::numeric AS capacity_num
					  FROM nuclear_power_plants
					),
					averages AS (
					  SELECT
					    reactor_type,
					    AVG(capacity_num) AS avg_capacity
					  FROM typed_capacities
					  WHERE capacity_num IS NOT NULL
					  GROUP BY reactor_type
					)
					SELECT
					  reactor_type,
					  avg_capacity
					FROM averages
					ORDER BY avg_capacity DESC
					LIMIT 1;
						Unreviewed query and Reviewed query correctly compute the average capacity per reactor type after safely casting capacity to numeric and excluding blanks/nulls, then pick the reactor type with the highest average. They directly answer the question.
						Unreviewed query may fail or miscompute if Capacity is stored as text or contains blanks, since it doesn't cast/clean the values before averaging. It's not robust to common data issues, so it may not correctly answer the question.
							✅	It is a good example of how the systme cast/clean the values before averaging. It's robust to common data issues.

Which country is Kursk-1 in?	SELECT Country FROM nuclear_power_plants WHERE Name = "Kursk-1"	SELECT country AS country FROM nuclear_power_plants WHERE name = 'Kursk-1';	✅	Unreviewed query correctly selects the country from nuclear_power_plants where the plant name equals the string 'Kursk-1' using standard SQL string quoting.	SELECT country AS country FROM nuclear_power_plants WHERE name = 'Kursk-1';	Reviewer didn't make any changes System generated-queries correctly select the country for the plant named 'Kursk-1' and are identical.	✅	

WhatCDHipHop Database

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation	Annotator Comment
What are the top 100 torrent releases?	SELECT groupName FROM torrents ORDER BY totalSnatched DESC LIMIT 100	SELECT totalsnatched AS totalsnatched, groupname AS release_name, artist AS artist, groupyear AS release_year, releasetype AS release_type, groupid AS group_id FROM torrents ORDER BY totalsnatched DESC, groupid ASC LIMIT 100;	✅	Both queries rank releases by total snatches in descending order and return the top 100, matching the intent of “top 100 torrent releases.”	SELECT totalsnatched AS totalsnatched, groupname AS release_name, artist AS artist, groupyear AS release_year, releasetype AS release_type, groupid AS group_id FROM torrents ORDER BY totalsnatched DESC, groupid ASC LIMIT 100;	Reviewer didn't make any changes they correctly identify the top 100 releases by ordering on totalSnatched (descending) and limiting to 100.	✅	

What are name of top 10 artists or groups?	SELECT artist FROM torrents GROUP BY artist ORDER BY sum(totalSnatched) DESC LIMIT 10	WITH artist_snatches AS (
		  SELECT
		    artist,
		    SUM(totalsnatched) AS total_snatched
		  FROM torrents
		  GROUP BY artist
		)
		SELECT
		  artist AS artist_name
		FROM artist_snatches
		ORDER BY total_snatched DESC, artist ASC
		LIMIT 10;
			✅	Both queries aggregate total snatches per artist from the torrents table, sort by that aggregate descending, and return the top 10 artist names.Unreviewed query does this via a CTE with a tie-breaker on artist name; The Benchmark-Proposed Query does it inline with ORDER BY SUM(totalSnatched).	WITH artist_snatches AS (
					  SELECT
					    artist,
					    SUM(totalsnatched) AS total_snatched
					  FROM torrents
					  GROUP BY artist
					)
					SELECT
					  artist AS artist_name
					FROM artist_snatches
					ORDER BY total_snatched DESC, artist ASC
					LIMIT 10;
						All three queries correctly aggregate snatches by artist and order by the summed total in descending order, returning the top 10 artist names.	✅	

What are the top 10 torrent release types?	SELECT releaseType FROM torrents GROUP BY releaseType ORDER BY sum(totalSnatched) DESC LIMIT 10	WITH cleaned AS (
		  SELECT releasetype
		  FROM torrents
		  WHERE releasetype IS NOT NULL AND btrim(releasetype) <> ''
		)
		SELECT
		  releasetype AS release_type,
		  COUNT(*) AS torrent_count
		FROM cleaned
		GROUP BY releasetype
		ORDER BY torrent_count DESC, release_type ASC
		LIMIT 10;
			✅	Unreviewed query correctly interprets "top" as the most common release types by count, excludes null/blank values, groups by release type, and returns the top 10 by frequency.
				The Benchmark-Proposed Query ranks release types by sum(totalSnatched), which measures popularity rather than count and is not specified in the question; it also doesn't filter out null/blank release types, which could yield an invalid category in the top 10.
					WITH cleaned AS (
					  SELECT releasetype
					  FROM torrents
					  WHERE releasetype IS NOT NULL AND btrim(releasetype) <> ''
					)
					SELECT
					  releasetype AS release_type,
					  COUNT(*) AS torrent_count
					FROM cleaned
					GROUP BY releasetype
					ORDER BY torrent_count DESC, release_type ASC
					LIMIT 10;
						These queries count torrents per release type and return the 10 most frequent types, matching "top 10 release types" by count. They also exclude NULL/blank values. Reviewer didn't make any changes.	✅	

What are the top 20 pop albums?	SELECT T2.groupName FROM torrents as T2 JOIN tags as T1 ON T1.id = T2.id WHERE T1.tag = "pop" and T2.releaseType = "album" ORDER BY T2.totalSnatched DESC LIMIT 20	WITH pop_albums AS (
		 SELECT
		  t.groupid AS album_group_id,
		  t.groupname AS album_title,
		  t.groupyear AS release_year,
		  t.totalsnatched AS snatches
		 FROM torrents t
		 INNER JOIN torrent_tags tt ON t.groupid = tt.groupid
		 INNER JOIN tags tg ON tt.tagid = tg.id
		 WHERE tg.tag = 'pop'
		  AND t.releasetype = 'album'
		)
		SELECT album_group_id, album_title, release_year, snatches
		FROM pop_albums
		ORDER BY snatches DESC
		LIMIT 20;
			✅	Unreviewed query correctly filters to pop by joining torrents -> torrent_tags -> tags via group/tag IDs, restricts to albums, and ranks by total snatches to return the top 20. This aligns with tagging at the group level and the intent of “top pop albums.”
				The Benchmark-Proposed Query joins tags.id to torrents.id (tag ID ≠ torrent ID) and skips the association table, so it won’t correctly filter “pop” albums and may return incorrect or empty results.
					WITH pop_albums AS (
					 SELECT
					  t.groupid AS album_group_id,
					  t.groupname AS album_title,
					  t.groupyear AS release_year,
					  t.totalsnatched AS snatches
					 FROM torrents t
					 INNER JOIN torrent_tags tt ON t.groupid = tt.groupid
					 INNER JOIN tags tg ON tt.tagid = tg.id
					 WHERE tg.tag = 'pop'
					  AND t.releasetype = 'album'
					)
					SELECT album_group_id, album_title, release_year, snatches
					FROM pop_albums
					ORDER BY snatches DESC
					LIMIT 20;
						Reviewed query correctly joins torrents to tags through the association table (torrent_tags → tags), filters for tag = 'pop' and releaseType = 'album', orders by totalsnatched, and limits to 20, which matches “top 20 pop albums.”	✅	

What are the top 20 instrumental albums since 2000?	SELECT T2.groupName FROM torrents as T2 JOIN tags as T1 ON T1.id = T2.id WHERE T1.tag = "instrumental" and T2.releaseType = "album" and groupYear >= 2000 ORDER BY T2.totalSnatched DESC LIMIT 20	SELECT t.groupname AS album_name, t.groupyear AS release_year, t.totalsnatched AS popularity FROM torrents AS t WHERE t.releasetype = 'album' AND t.groupyear >= 2000 ORDER BY t.totalsnatched DESC, t.groupname ASC LIMIT 20;	✗	The Benchmark-Proposed Query correctly filters for albums tagged "instrumental," restricts to releases from 2000 onward, and orders by totalSnatched to return the top 20 by popularity. Unreviewed query lacks any filter for the "instrumental" tag, so it returns the most snatched albums since 2000 regardless of whether they are instrumental.	SELECT t.groupname AS album_name, t.groupyear AS release_year, MAX(t.totalsnatched) AS popularity FROM torrents AS t WHERE t.releasetype = 'album' AND t.groupyear >= 2000 AND EXISTS (SELECT 1 FROM tags AS tg WHERE tg.tag = 'instrumental' AND tg.id = t.groupid) GROUP BY t.groupname, t.groupyear ORDER BY popularity DESC, album_name ASC LIMIT 20;	Reviewed query is incorrect because it links tags via tg.id = t.groupid (mismatched key given the schema implied byUnreviewed query where tags.id matches torrents.id). It also groups by album_name and year and uses MAX(t.totalSnatched), which can collapse distinct albums sharing the same name/year and distort rankings.	✗	

What type of musical album is the most popular?	SELECT T1.tag FROM torrents as T2 JOIN tags as T1 ON T1.id = T2.id WHERE T2.releaseType = "album" GROUP BY T1.tag ORDER BY sum(T2.totalSnatched) DESC LIMIT 1	WITH valid_torrents AS (
		  SELECT
		    releasetype,
		    totalsnatched
		  FROM torrents
		  WHERE totalsnatched IS NOT NULL
		),
		popularity_by_type AS (
		  SELECT
		    releasetype AS release_type,
		    SUM(totalsnatched) AS total_snatches
		  FROM valid_torrents
		  GROUP BY releasetype
		)
		SELECT
		  release_type,
		  total_snatches
		FROM popularity_by_type
		ORDER BY total_snatches DESC
		LIMIT 1;
			✅	Unreviewed query correctly aggregates popularity (totalSnatched) by release type (releasetype) and returns the most popular type, which matches the intent of identifying the most popular type of musical album.
				The Benchmark-Proposed Query is incorrect because it interprets “type” as a tag/genre and restricts to albums only, answering a different question (“most popular tag among albums”). It also likely misjoins tags to torrents via T1.id = T2.id instead of a proper many-to-many mapping, which would produce incorrect results.
					WITH valid_torrents AS (
					  SELECT
					    releasetype,
					    totalsnatched
					  FROM torrents
					  WHERE totalsnatched IS NOT NULL
					),
					popularity_by_type AS (
					  SELECT
					    releasetype AS release_type,
					    SUM(totalsnatched) AS total_snatches
					  FROM valid_torrents
					  GROUP BY releasetype
					)
					SELECT
					  release_type,
					  total_snatches
					FROM popularity_by_type
					ORDER BY total_snatches DESC
					LIMIT 1;
						Unreviewed query and Reviewed query correctly aggregate popularity by release type (releasetype) using total snatches and return the most popular type. They directly answer "What type of musical album is the most popular?" where "type" refers to the release type (e.g., album, EP, single).	✅	

What are the top 100 single musics?	SELECT groupName FROM torrents WHERE releaseType = "single" ORDER BY totalSnatched DESC LIMIT 100	SELECT artist AS artist, groupname AS title, groupyear AS year, totalsnatched AS snatches FROM torrents WHERE releasetype = 'single' ORDER BY totalsnatched DESC LIMIT 100;	✅	Unreviewed query correctly filters singles (releasetype = 'single'), orders by total snatches descending, limits to 100, and returns meaningful identifiers (artist, title, year, snatches).	SELECT artist AS artist, groupname AS title, groupyear AS year, totalsnatched AS snatches FROM torrents WHERE releasetype = 'single' ORDER BY totalsnatched DESC LIMIT 100;	System generateds are identical and correctly filter singles (releasetype = 'single'), order by snatches (totalsnatched DESC), and limit to 100, returning relevant fields.	✅	
What is the torrent download statistics for each release year?	SELECT sum(totalSnatched), groupYear FROM torrents GROUP BY groupYear	SELECT groupyear AS release_year, SUM(totalsnatched) AS total_downloads FROM torrents GROUP BY groupyear ORDER BY release_year;	✅	Benchmark and unreviewed group by release year (groupYear) and sum totalSnatched, producing total downloads per release year.Unreviewed query adds helpful aliases and orders by year, but that isn’t required to answer the question.	SELECT groupyear AS release_year, SUM(totalsnatched) AS total_downloads FROM torrents GROUP BY groupyear ORDER BY release_year;	All three queries correctly aggregate total downloads (totalSnatched) by release year (groupYear), which answers the question. Unreviewed query and Reviewed query are identical and provide clear aliases and ordering	✅	

how many entries are of each type?	SELECT count(*), releaseType FROM torrents GROUP BY releaseType	SELECT
		  releasetype AS release_type,
		  COUNT(*) AS entry_count
		FROM torrents
		GROUP BY releasetype
		ORDER BY releasetype;
			✅	Benchmark and unreviewed queries group by the release type column and count rows per type, returning the required per-type counts.Unreviewed query adds aliases and ordering; The Benchmark-Proposed Query omits them but still answers the question.	SELECT
					  releasetype AS release_type,
					  COUNT(*) AS entry_count
					FROM torrents
					GROUP BY releasetype
					ORDER BY releasetype;
						Reviewer didn't make any changes. All three correctly group by release type and count rows, answering “how many entries are of each type?”.
						None are incorrect.
							✅	

how many unique entries are by the same Artist/group?	SELECT count(*), artist FROM torrents GROUP BY artist	WITH grouped AS (SELECT t.artist, t.groupname, COUNT(*) AS entries_count FROM torrents AS t WHERE t.artist IS NOT NULL AND t.groupname IS NOT NULL GROUP BY t.artist, t.groupname) SELECT COUNT(*) AS total_unique_pairs FROM grouped WHERE entries_count > 1;	✅	Unreviewed query is correct because it groups by the composite key (artist, groupname), counts how many rows exist per pair, and then returns a single scalar: the number of artist/group pairs that have more than one entry (i.e., the same artist/group appearing multiple times). This matches the intent of "how many unique entries are by the same Artist/group?" as counting unique pairs that recur. The Benchmark-Proposed Query is incorrect because it only groups by artist (ignoring groupname), returns multiple rows (one per artist) instead of a single total, and does not specifically identify or count cases where the same artist/group pair occurs more than once.	WITH w AS (SELECT id, artist, groupname, COUNT(*) OVER (PARTITION BY artist) AS cnt_artist, COUNT(*) OVER (PARTITION BY groupname) AS cnt_group FROM torrents) SELECT COUNT(*) AS entries_sharing_artist_or_group FROM w WHERE (artist IS NOT NULL AND cnt_artist > 1) OR (groupname IS NOT NULL AND cnt_group > 1);	Reviewed query correctly returns a single number: the count of entries that share the same artist or the same group with at least one other entry. It handles either dimension (artist or group), avoids double counting a row, and ignores nulls appropriately in each condition.	✅	

which entry have been downloaded the most?	SELECT groupName FROM torrents ORDER BY totalSnatched DESC LIMIT 1	WITH max_snatches AS (SELECT MAX(totalsnatched) AS max_total FROM torrents) SELECT t.groupid AS group_id, t.groupname AS group_name, t.totalsnatched AS total_downloads FROM torrents t JOIN max_snatches m ON t.totalsnatched = m.max_total;	✅	Unreviewed query correctly finds the maximum totalSnatched and returns all entries whose totalSnatched equals that maximum, which accurately identifies the entry/entries downloaded the most (including ties). The Benchmark-Proposed Query orders and LIMIT 1, which arbitrarily returns one entry when there are ties and thus can omit other equally “most downloaded” entries; it also provides less identifying information.	WITH max_snatches AS (SELECT MAX(totalsnatched) AS max_total FROM torrents) SELECT t.groupid AS group_id, t.groupname AS group_name, t.totalsnatched AS total_downloads FROM torrents t JOIN max_snatches m ON t.totalsnatched = m.max_total;	Unreviewed query and Reviewed query correctly compute the maximum totalsnatched and return all entries that match it, handling ties.	✅	
which entry have been downloaded the least?	SELECT groupName FROM torrents ORDER BY totalSnatched LIMIT 1	SELECT groupname AS entry_name, totalsnatched AS download_count FROM torrents WHERE totalsnatched IS NOT NULL ORDER BY totalsnatched ASC LIMIT 1;	✅	Unreviewed query correctly orders by the download count (totalsnatched) ascending and excludes NULLs, ensuring it returns the entry with the smallest known download count. It also returns the entry name.	SELECT groupname AS entry_name, totalsnatched AS download_count FROM torrents ORDER BY totalsnatched ASC LIMIT 1;	Reviewed query correctly orders by the download count (totalsnatched). It also returns both the entry name and its count.	✅	

what are titles for each unique entry?	SELECT DISTINCT groupName FROM torrents	SELECT DISTINCT groupname AS title FROM torrents;	✅	Unreviewed query selects the distinct values from the intended column and aliases them as title, matching the question's request for titles for each unique entry. The Benchmark-Proposed Query is not a valid SQL query as presented (it's wrapped in a list/string literal). Even if unwrapped, it lacks the alias aligning with "titles," and may have case-sensitivity issues depending on the SQL dialect.	SELECT groupname AS title FROM torrents GROUP BY groupname;	Reviewer didn't make any changes. All three return the set of unique titles (groupname) from torrents. Unreviewed query uses DISTINCT on groupName;Unreviewed query does the same and aliases it as title; Reviewed query uses GROUP BY groupname (equivalent to DISTINCT) and aliases it as title. None include extra columns, so all are correct.	✅	
what year was each specific entry released?	SELECT groupName, groupYear FROM torrents	SELECT groupname AS entry_name, artist AS artist, releasetype AS release_type, groupyear AS release_year FROM torrents;	✅	Benchmark and unreviewed queries return each entry (groupName) with its release year (groupYear), which answers the question.	SELECT groupname AS entry_name, artist AS artist, releasetype AS release_type, groupyear AS release_year FROM torrents;	All three queries return the release year for each entry from the torrents table.	✅	

what are the entries by each specific artist/group ?	SELECT groupName, artist FROM torrents	SELECT artist AS artist, groupname AS entry_name, groupid AS entry_id, releasetype AS release_type, groupyear AS release_year, totalsnatched AS total_snatched FROM torrents ORDER BY artist ASC, groupyear DESC, groupname ASC;	✅	Unreviewed query lists each entry (group) along with its artist and key identifying/details fields (name, ID, release type, year, total snatched) and organizes results by artist, which directly answers “what are the entries by each specific artist/group?”	SELECT artist AS artist, groupname AS entry_name, groupid AS entry_id, releasetype AS release_type, groupyear AS release_year, totalsnatched AS total_snatched FROM torrents ORDER BY artist ASC, groupyear DESC, groupname ASC;	Reviewer didn't make any changes. All three queries list entries (groupname) associated with each artist from the torrents table.	✅	
which tags exist?	SELECT DISTINCT tag FROM tags	SELECT DISTINCT tag AS tag FROM tag WHERE tag IS NOT NULL;	❌	Unreviewed query is incorrect because it selects from a table named tag instead of tags, likely referencing a non-existent or wrong table.	SELECT DISTINCT tag AS tag FROM tags WHERE tag IS NOT NULL;	Reviewed query is correct because it returns the distinct, non-null tag values from the tags table, aligning with the idea of tags that actually exist.	✅	This example shows how reviwer solve naming issues if exists

what release types are captured in this data set?	SELECT DISTINCT releaseType FROM torrents	SELECT DISTINCT releasetype AS release_type FROM torrents WHERE releasetype IS NOT NULL ORDER BY release_type;	✅	Unreviewed query correctly lists the distinct release types present while excluding NULLs, which are not meaningful as a release type. ORDER BY and aliasing are harmless and improve readability. The Benchmark-Proposed Query may include a NULL value in the results if any rows lack a release type, which does not answer the question of what release types are captured.	SELECT DISTINCT releasetype AS release_type FROM torrents ORDER BY release_type;	Reviewed query correctly returns the distinct set of actual release types present by filtering out NULLs (which are not a “type”), and the ordering/alias are harmless.	✅	

Find me top 5 most popular releases after 2000?	SELECT groupName FROM torrents WHERE groupYear > 2000 ORDER BY totalSnatched DESC LIMIT 5	SELECT groupname AS release_name, groupyear AS release_year, totalsnatched AS popularity FROM torrents WHERE groupyear > 2000 AND totalsnatched IS NOT NULL ORDER BY totalsnatched DESC LIMIT 5;	✅	Unreviewed query correctly filters releases after 2000, ranks by popularity (totalSnatched), and excludes NULL popularity values so the top 5 are truly the most popular with known counts. The Benchmark-Proposed Query can return rows with NULL totalSnatched among the top results in databases where NULLs sort first on DESC, which misrepresents “most popular.” It also omits the popularity metric and year, making validation harder.	SELECT groupname AS release_name, groupyear AS release_year, totalsnatched AS popularity FROM torrents WHERE groupyear > 2000 AND totalsnatched IS NOT NULL ORDER BY totalsnatched DESC LIMIT 5;	Reviewer didn't make any changes. they correctly filter releases after 2000, rank by totalsnatched (popularity) descending, exclude NULL popularity to ensure meaningful ordering, and return the top 5 with relevant fields.	✅	

Which albums have been downloade d more than 100 times?	SELECT DISTINCT groupName FROM torrents WHERE totalSnatched > 100 AND releaseType = "album"	SELECT
		  groupname AS album_name,
		  totalsnatched AS download_count
		FROM torrents
		WHERE releasetype = 'album'
		  AND totalsnatched > 100;
			✗	The Benchmark-Proposed Query returns the unique set of album names where the album-type torrents have been downloaded more than 100 times, which matches “Which albums…”. DISTINCT avoids duplicates when an album has multiple qualifying torrents. Unreviewed query operates at the torrent row level, can list the same album multiple times, and reports per-torrent counts rather than a unique list of albums, so it doesn’t directly answer the request.	SELECT
					  groupname AS album_name,
					  totalsnatched AS download_count
					FROM torrents
					WHERE releasetype = 'album'
					  AND totalsnatched > 100;
						Reviewed query is incorrect for the same reason.	✗	

Name the most popular release on houston.	SELECT T2.groupName FROM torrents as T2 JOIN tags as T1 ON T1.id = T2.id WHERE T1.tag = "houston" ORDER BY totalSnatched DESC LIMIT 1	WITH ranked_releases AS (SELECT groupname AS release_name, artist, totalsnatched AS total_snatched, groupyear AS release_year, ROW_NUMBER() OVER (ORDER BY totalsnatched DESC, groupname ASC) AS rn FROM torrents) SELECT release_name, artist, total_snatched, release_year FROM ranked_releases WHERE rn = 1;	❌	The Benchmark-Proposed Query correctly filters releases to those tagged "houston," orders by totalSnatched (popularity) descending, and returns the top release name. Unreviewed query does not filter by the "houston" tag; it ranks all releases globally, so it answers a different question.	SELECT groupname AS release_name, artist, totalsnatched AS total_snatched, groupyear AS release_year FROM torrents WHERE LOWER(artist) = 'houston' ORDER BY totalsnatched DESC, groupname ASC LIMIT 1;	Reviewed query is incorrect because it filters by artist = 'houston' rather than by the "houston" tag, changing the intent from “on houston” to “by Houston.”	❌	

Name the most popular and least popular releases of lasean camry?	SELECT groupName FROM torrents WHERE artist = "lasean camry" AND totalSnatched = (SELECT max(totalSnatched) FROM torrents WHERE artist = "lasean camry") UNION SELECT groupName FROM torrents WHERE artist = "lasean camry" AND totalSnatched = (SELECT min(totalSnatched) FROM torrents WHERE artist = "lasean camry")	WITH filtered AS (SELECT groupname AS release_name, groupyear AS release_year, releasetype AS release_type, totalsnatched AS snatches, RANK() OVER (ORDER BY totalsnatched DESC) AS popularity_rank_desc, RANK() OVER (ORDER BY totalsnatched ASC) AS popularity_rank_asc FROM torrents WHERE artist = 'lasean camry' AND totalsnatched IS NOT NULL), most_popular AS (SELECT release_name, release_year, release_type, snatches, 'most_popular' AS popularity FROM filtered WHERE popularity_rank_desc = 1), least_popular AS (SELECT release_name, release_year, release_type, snatches, 'least_popular' AS popularity FROM filtered WHERE popularity_rank_asc = 1) SELECT release_name, release_year, release_type, snatches, popularity FROM most_popular UNION ALL SELECT release_name, release_year, release_type, snatches, popularity FROM least_popular;	✅	Unreviewed query correctly filters to the artist, excludes NULL snatches, uses ranking to return all ties for both most and least popular, and clearly distinguishes categories. It robustly answers the question. The Benchmark-Proposed Query uses double quotes for string literals (invalid in many SQL dialects), provides no indication of which results are most vs least, and uses UNION (not ALL), which can collapse results if max equals min, obscuring categories.	WITH filtered AS (SELECT groupname AS release_name, groupyear AS release_year, releasetype AS release_type, totalsnatched AS snatches, RANK() OVER (ORDER BY totalsnatched DESC) AS popularity_rank_desc, RANK() OVER (ORDER BY totalsnatched ASC) AS popularity_rank_asc FROM torrents WHERE artist = 'lasean camry' AND totalsnatched IS NOT NULL), most_popular AS (SELECT release_name, release_year, release_type, snatches, 'most_popular' AS popularity FROM filtered WHERE popularity_rank_desc = 1), least_popular AS (SELECT release_name, release_year, release_type, snatches, 'least_popular' AS popularity FROM filtered WHERE popularity_rank_asc = 1) SELECT release_name, release_year, release_type, snatches, popularity FROM most_popular UNION ALL SELECT release_name, release_year, release_type, snatches, popularity FROM least_popular;	Unreviewed query and Reviewed query are correct. They: Filter to the artist. Use ranks over totalsnatched to identify both the maximum and minimum (handling ties). Exclude NULL snatch counts. Return both extremes with clear labeling, plus useful release context (name/year/type/snatches).	✅	

Find me top 10 albums ranked by their popularity.	SELECT groupName FROM torrents WHERE releaseType = "album" ORDER BY totalSnatched DESC LIMIT 10	SELECT
		  groupname AS album_name,
		  groupyear AS release_year,
		  totalsnatched AS popularity
		FROM torrents
		WHERE releasetype = 'album'
		  AND totalsnatched IS NOT NULL
		ORDER BY totalsnatched DESC, groupname ASC
		LIMIT 10;
			✅	Unreviewed query correctly filters albums, orders by the popularity metric (totalsnatched) in descending order, excludes NULL popularity values (preventing NULLs from incorrectly appearing at the top in some SQL dialects), and limits to 10. It also provides deterministic tie-breaking.
				The Benchmark-Proposed Query is unreliable: it uses "album" with double quotes (an identifier in many SQL dialects like PostgreSQL, causing errors), does not exclude NULL totals (which can surface first with DESC in some systems), and thus may not return the true top 10 by popularity.
					SELECT
					  groupname AS album_name,
					  groupyear AS release_year,
					  totalsnatched AS popularity
					FROM torrents
					WHERE releasetype = 'album'
					  AND totalsnatched IS NOT NULL
					ORDER BY totalsnatched DESC, groupname ASC
					LIMIT 10;
						Reviewer didn't make any changes.
						they correctly filter albums, use totalsnatched as popularity, sort descending by popularity (with a deterministic tie-breaker), and limit to 10.
							✅	

Find me the most popular topics since 2010.	SELECT T1.tag FROM torrents as T2 JOIN tags as T1 ON T1.id = T2.id WHERE T2.groupYear >= 2010 GROUP BY T1.tag ORDER BY T2.totalSnatched DESC LIMIT 10	WITH filtered_torrents AS (SELECT totalsnatched FROM torrents WHERE groupyear >= 2010) SELECT COALESCE(SUM(totalsnatched), 0) AS total_snatches_since_2010 FROM filtered_torrents;	✗	The Benchmark-Proposed Query returns topics (tags) for torrents since 2010 and ranks them by a popularity metric (totalSnatched), limiting to the top 10, which aligns with "most popular topics since 2010." Unreviewed query only sums total snatches across all torrents since 2010 and returns a single number, not a list of popular topics.	SELECT releasetype AS topic, SUM(totalsnatched) AS total_snatches FROM torrents WHERE groupyear >= 2010 GROUP BY releasetype ORDER BY total_snatches DESC;	Reviewed query uses releasetype instead of the tags table, so it may not reflect the actual topics/tags associated with the torrents.	✗	

Which year had the minimum number of releases?	SELECT groupYear FROM torrents GROUP BY groupYear ORDER BY count(groupName) LIMIT 1	WITH releases_by_year AS (
		  SELECT
		    groupyear AS year,
		    COUNT(*) AS release_count
		  FROM torrents
		  WHERE groupyear IS NOT NULL
		  GROUP BY groupyear
		)
		SELECT
		  year,
		  release_count
		FROM releases_by_year
		ORDER BY release_count ASC, year ASC
		FETCH FIRST 1 ROW ONLY;
			✅	Unreviewed query correctly counts releases per year (COUNT(*)), excludes NULL years, and returns the year with the smallest count (with a deterministic tie-break on year).
				The Benchmark-Proposed Query orders by COUNT(groupName), which ignores rows where groupName is NULL and includes the NULL groupYear bucket, potentially returning NULL or undercounting. It also lacks a tie-break and may not consistently identify the correct minimal year.
					WITH releases_by_year AS (
					  SELECT
					    groupyear AS year,
					    COUNT(*) AS release_count
					  FROM torrents
					  WHERE groupyear IS NOT NULL
					  GROUP BY groupyear
					)
					SELECT
					  year,
					  release_count
					FROM releases_by_year
					ORDER BY release_count ASC, year ASC
					FETCH FIRST 1 ROW ONLY;
						Reviewer didn't make any changes
						they correctly count releases per non-null year (COUNT(*)), order by the smallest count (with a deterministic year tiebreaker), and return the year with the minimum releases.
							✅	

Which release is being downloade d the most?	SELECT groupName FROM torrents ORDER BY totalSnatched DESC LIMIT 1	SELECT groupname AS release_name, totalsnatched AS download_count FROM torrents WHERE totalsnatched = (SELECT MAX(totalsnatched) FROM torrents);	✅	Unreviewed query correctly identifies the release(s) with the maximum totalsnatched and returns all ties, directly answering which release is downloaded the most. The Benchmark-Proposed Query returns only one row via LIMIT 1, arbitrarily excluding other releases that may share the top download count, so it can be incorrect in tie scenarios.	SELECT groupname AS release_name, totalsnatched AS download_count FROM torrents WHERE totalsnatched = (SELECT MAX(totalsnatched) FROM torrents);	Unreviewed query correctly return the release(s) with the maximum totalsnatched (handling ties) and include the download count.	✅	

Who or which group is most popular in 2015?	SELECT artist FROM torrents WHERE groupYear = 2015 GROUP BY artist ORDER BY totalSnatched DESC LIMIT 1	WITH filtered AS (
		  SELECT
		    groupname,
		    totalsnatched
		  FROM torrents
		  WHERE groupyear = 2015
		),
		aggregated AS (
		  SELECT
		    groupname,
		    SUM(totalsnatched) AS total_snatched
		  FROM filtered
		  GROUP BY groupname
		)
		SELECT
		  groupname AS group_name,
		  total_snatched
		FROM aggregated
		ORDER BY total_snatched DESC, groupname ASC
		LIMIT 1;
			✅	Unreviewed query correctly filters to 2015, aggregates totalSnatched per group (groupname), and returns the top group by summed popularity, matching the intent of “most popular in 2015.” The Benchmark-Proposed Query groups by artist but orders by totalSnatched without aggregation, which is invalid in standard SQL and, even where allowed, uses an arbitrary single row’s value rather than the artist’s total, so it does not correctly measure overall popularity.	WITH filtered AS (
					  SELECT
					    groupname,
					    totalsnatched
					  FROM torrents
					  WHERE groupyear = 2015
					),
					aggregated AS (
					  SELECT
					    groupname,
					    SUM(totalsnatched) AS total_snatched
					  FROM filtered
					  GROUP BY groupname
					)
					SELECT
					  groupname AS group_name,
					  total_snatched
					FROM aggregated
					ORDER BY total_snatched DESC, groupname ASC
					LIMIT 1;
						Reviewer didn't make any changes.
						they correctly sum totalsnatched per groupname for 2015 and return the top group, matching “which group is most popular in 2015?”.
							✅	

Which tag is used for which release title?	SELECT T1.tag, T2.groupName FROM torrents as T2 JOIN tags as T1 ON T1.id = T2.id	SELECT
		  t.groupname AS release_title,
		  g.tag AS tag
		FROM torrents AS t
		JOIN tags AS g
		  ON t.groupname = g.tag;
			❌	Unreviewed query is incorrect: It joins where the release title equals the tag text (t.groupname = g.tag), which is logically unrelated and will only match when a title string happens to be identical to a tag.	SELECT
					  t.groupname AS release_title,
					  g.tag AS tag
					FROM torrents AS t
					INNER JOIN tags AS g
					  ON t.groupid = g.index;
						Reviewed query correctly maps tags to release titles by joining at the release/group level (t.groupid to g.index) and returns the intended fields (release_title = groupname, tag = tag). This aligns with the intent: tags are associated with a release (group), not an individual torrent row ID.
						The Benchmark-Proposed Query is incorrect because it joins on T1.id = T2.id, equating a tag's ID with a torrent's ID. unrelated identifiers, so it does not represent the relationship between tags and release titles and will yield incorrect or empty matches.
							✅	

Which year has the most released song?	SELECT groupYear FROM torrents GROUP BY groupYear ORDER BY count(groupName) DESC LIMIT 1	WITH singles_by_year AS (SELECT groupyear AS year_with_most_songs, COUNT(DISTINCT groupid) AS song_count FROM torrents WHERE releasetype = 'single' GROUP BY groupyear) SELECT year_with_most_songs, song_count FROM singles_by_year ORDER BY song_count DESC, year_with_most_songs ASC LIMIT 1;	✅	Unreviewed query correctly counts unique single releases (songs) per year by using releasetype = 'single' and COUNT(DISTINCT groupid), then returns the year with the highest count. The Benchmark-Proposed Query counts rows (via count(groupName)) which can overcount due to multiple torrents per release and doesn't restrict to songs/singles, so it answers a different, incorrect metric.	WITH singles_by_year AS (SELECT groupyear AS year_with_most_songs, COUNT(DISTINCT groupid) AS song_count FROM torrents WHERE releasetype = 'single' GROUP BY groupyear) SELECT year_with_most_songs, song_count FROM singles_by_year ORDER BY song_count DESC, year_with_most_songs ASC LIMIT 1;	Reviewer didn't make any changes. Unreviewed query (and identical Reviewed query) correctly filters to songs (releasetype = 'single'), counts distinct song groups per year to avoid duplicate torrents, and returns the year with the highest count.	✅	

Which artist/group is most productive?	SELECT artist FROM torrents GROUP BY artist ORDER BY count(groupName) DESC LIMIT 1	WITH artist_counts AS (
		  SELECT
		    artist,
		    groupname AS group_name,
		    COUNT(*) AS release_count
		  FROM torrents
		  WHERE artist IS NOT NULL AND artist <> ''
		  GROUP BY artist, groupname
		),
		ranked AS (
		  SELECT
		    artist,
		    group_name,
		    release_count,
		    RANK() OVER (ORDER BY release_count DESC) AS rnk
		  FROM artist_counts
		)
		SELECT
		  artist,
		  group_name,
		  release_count
		FROM ranked
		WHERE rnk = 1
		LIMIT 1;
			❌	The Benchmark-Proposed Query groups by artist and orders by the number of releases (count of groupName) to return the artist with the most releases, matching the intent of “most productive” at the artist level.
				Unreviewed query aggregates by the (artist, groupname) pair, returning the single most frequent artist–groupname combination (e.g., a specific album by an artist), not the most productive artist or group overall. It also excludes rows with null/empty artist, so it cannot return a “group” on its own.
					SELECT artist, COUNT(*) AS release_count
					FROM torrents
					WHERE artist IS NOT NULL AND artist <> ''
					GROUP BY artist
					ORDER BY release_count DESC
					LIMIT 1;
						Reviewed query correctly aggregates total releases per artist, filters out null/empty artists, orders by the total count, and returns the top artist, matching “most productive” as most releases.	✅	

Which artist/group has the highest average download of songs?	SELECT artist FROM torrents GROUP BY artist ORDER BY avg(totalSnatched) DESC LIMIT 1	SELECT
		  artist,
		  AVG(totalsnatched) AS avg_downloads
		FROM torrents
		WHERE artist IS NOT NULL
		  AND totalsnatched IS NOT NULL
		GROUP BY artist
		ORDER BY avg_downloads DESC
		LIMIT 1;
			✅	Unreviewed query correctly computes the per-artist average downloads, excludes NULL artists and NULL download counts, orders by that average, and returns the top artist, aligning with “Which artist/group.” The Benchmark-Proposed Query does not exclude NULL artists; it can return a NULL “artist” and may also surface groups with NULL averages (depending on DB NULL sort behavior), so it isn’t reliably correct.	SELECT
					  artist,
					  AVG(totalsnatched) AS avg_downloads
					FROM torrents
					WHERE artist IS NOT NULL
					  AND totalsnatched IS NOT NULL
					GROUP BY artist
					ORDER BY avg_downloads DESC
					LIMIT 1;
						Reviewer didn't make any changes. Unreviewed query and Reviewed query correctly compute the average downloads per artist (AVG(totalsnatched)), exclude NULL artists and NULL download counts, sort descending, and return the top artist.		✅	

Which year has the most CDs released?	SELECT groupYear FROM torrents GROUP BY groupYear ORDER BY count(groupName) DESC LIMIT 1	WITH cd_releases AS (SELECT groupYear AS year FROM torrents WHERE releasetype = 'CD') SELECT year, COUNT(*) AS cd_count FROM cd_releases GROUP BY year ORDER BY cd_count DESC, year ASC LIMIT 1;	✅	The Benchmark-Proposed Query does not filter to CDs and counts groupName (which may exclude NULLs), so it returns the year with the most releases overall, not specifically CDs.	WITH cd_releases AS (SELECT t.groupyear AS year FROM torrents AS t WHERE t.releasetype = 'CD') SELECT year, COUNT(*) AS cd_count FROM cd_releases GROUP BY year ORDER BY cd_count DESC, year ASC LIMIT 1;	Reviewed query is correct because it filters to CDs (releasetype = 'CD'), groups by year, counts CD releases, and returns the year with the highest count, breaking ties by earliest year.	✅	

Which CD has been downloaded the most times?	SELECT groupName FROM torrents ORDER BY totalSnatched DESC LIMIT 1	SELECT
		  groupid AS cd_id,
		  groupname AS cd_name,
		  groupyear AS release_year,
		  totalsnatched AS download_count
		FROM torrents
		WHERE releasetype = 'album'
		  AND totalsnatched IS NOT NULL
		ORDER BY totalsnatched DESC, groupid ASC
		LIMIT 1;
			✅	Unreviewed query correctly targets CDs by restricting to albums (releasetype = 'album'), ignores NULL download counts, orders by downloads descending with a deterministic tiebreak, and returns the top result, matching “downloaded the most times.”
				The Benchmark-Proposed Query does not restrict to CDs/albums, so it could return a non-CD release; it also doesn’t handle NULLs or tie-breaking.
					SELECT
					  t.groupid AS cd_id,
					  t.groupname AS cd_name,
					  t.groupyear AS release_year,
					  SUM(t.totalsnatched) AS download_count
					FROM torrents t
					WHERE t.totalsnatched IS NOT NULL
					GROUP BY t.groupid, t.groupname, t.groupyear
					ORDER BY download_count DESC, cd_id ASC
					LIMIT 1;
						Reviewed query correctly aggregates totalsnatched across all torrents within each CD (group/groupid), then selects the CD with the highest combined downloads, which matches “Which CD has been downloaded the most times?”	✅	

Which kind of release type is the most popular?	SELECT releaseType FROM torrents GROUP BY releaseType ORDER BY sum(totalSnatched) DESC LIMIT 1	WITH popularity_by_type AS (SELECT releasetype AS release_type, SUM(totalsnatched) AS total_snatches FROM torrents WHERE totalsnatched IS NOT NULL GROUP BY releasetype) SELECT release_type, total_snatches FROM popularity_by_type ORDER BY total_snatches DESC LIMIT 1;	✅	Benchamrk and unreviewed queries compute total popularity as the sum of totalSnatched per releaseType and select the type with the highest sum, which answers the question.	WITH popularity_by_type AS (SELECT releasetype AS release_type, SUM(totalsnatched) AS total_snatches FROM torrents WHERE totalsnatched IS NOT NULL GROUP BY releasetype) SELECT release_type, total_snatches FROM popularity_by_type ORDER BY total_snatches DESC LIMIT 1;	Reviewer didn't make any changes All three aggregate total snatches by release type and select the top one, which correctly identifies the most popular release type.	✅	

Whose CDs sells best?	SELECT artist FROM torrents GROUP BY artist ORDER BY sum(totalSnatched) DESC LIMIT 1	WITH cds AS (
		  SELECT
		    groupname AS artist_or_group,
		    totalsnatched
		  FROM torrents
		  WHERE releasetype = 'album'
		),
		group_sums AS (
		  SELECT
		    artist_or_group,
		    SUM(totalsnatched) AS total_snatches
		  FROM cds
		  GROUP BY artist_or_group
		)
		SELECT artist_or_group, total_snatches
		FROM group_sums
		ORDER BY total_snatches DESC
		LIMIT 1;
			✅	Unreviewed query filters to albums (interpreting “CDs”), sums totalSnatched per artist/group, and returns the top aggregate, matching “Whose CDs sells best?”
				The Benchmark-Proposed Query does not restrict to CDs/albums, aggregating across all release types, so it answers a different question.
					SELECT artist, SUM(totalsnatched) AS total_snatches
					FROM torrents
					GROUP BY artist
					ORDER BY total_snatches DESC
					LIMIT 1;
						Reviewer didn't make any changes
						Correct: they both aggregate totalsnatched by artist and return the top artist, which answers “Whose CDs sells best?” at the artist level.
						Incorrect:bemnchmark query filters to albums but groups by groupname (likely release/album title), not the artist, so it answers “which album sells best,” not “whose CDs.”
							✅	

What are the downloade d numbers and their release types?	SELECT sum(totalSnatched), releaseType FROM torrents GROUP BY releaseType	SELECT torrents.totalsnatched AS downloaded_numbers, torrents.releasetype AS release_type FROM torrents;	✅	Unreviewed query returns each torrent's download count (totalSnatched) alongside its release type, directly listing the downloaded numbers and their release types as asked. The Benchmark-Proposed Query aggregates downloads by releaseType (SUM + GROUP BY), answering a different question: total downloads per release type, not the per-torrent downloaded numbers with their release types.	SELECT torrents.totalsnatched AS downloaded_numbers, torrents.releasetype AS release_type FROM torrents;	Reviewer didn't make any changes. Correct: they return each torrent's downloaded number (totalsnatched) alongside its release type, matching the request. Incorrect:benchamark query aggregates downloads by release type (SUM and GROUP BY), which doesn't list the downloaded numbers per torrent.	✅	

What are the downloaded numbers of 2000s and before 2000?	SELECT sum(totalSnatched) FROM torrents WHERE groupYear BETWEEN 2000 AND 2010 UNION SELECT sum(totalSnatched) FROM torrents WHERE groupYear < 2000	WITH bucketed AS (SELECT CASE WHEN groupyear BETWEEN 2000 AND 2009 THEN '2000s' WHEN groupyear < 2000 THEN 'before 2000' ELSE NULL END AS period_bucket, totalsnatched FROM torrents) SELECT period_bucket, SUM(totalsnatched) AS total_downloads FROM bucketed WHERE period_bucket IS NOT NULL GROUP BY period_bucket ORDER BY period_bucket ASC;	✅	Unreviewed query correctly buckets years into 2000–2009 as "2000s" and <2000 as "before 2000", then sums downloads per bucket, returning both numbers with clear labels. The Benchmark-Proposed Query is incorrect because it includes 2010 in the "2000s" bucket (BETWEEN 2000 AND 2010), uses UNION (risking loss of a row if the two sums are equal), and provides unlabeled results.	WITH bucketed AS (SELECT CASE WHEN groupyear BETWEEN 2000 AND 2009 THEN '2000s' WHEN groupyear < 2000 THEN 'before 2000' ELSE NULL END AS period_bucket, totalsnatched FROM torrents) SELECT period_bucket, SUM(totalsnatched) AS total_downloads FROM bucketed WHERE period_bucket IS NOT NULL GROUP BY period_bucket ORDER BY period_bucket ASC;	Reviewer didn't make any changes. they are identical and correctly bucket years into "2000s" (2000–2009) and "before 2000", then sum totalSnatched per bucket.	✅	

How many same release titles are there in the table?	SELECT count(*) FROM (SELECT groupName FROM torrents GROUP BY groupName HAVING count(*) > 1)	WITH duplicate_titles AS (SELECT groupname AS title, COUNT(*) AS duplicate_count FROM torrents GROUP BY groupname HAVING COUNT(*) > 1) SELECT COUNT(*) AS total_duplicate_titles FROM duplicate_titles;	✅	Both queries group by groupname and count groups with COUNT(*) > 1, then count those groups, yielding the number of distinct release titles that appear more than once.	WITH duplicate_titles AS (SELECT groupname AS title, COUNT(*) AS duplicate_count FROM torrents GROUP BY groupname HAVING COUNT(*) > 1) SELECT COUNT(*) AS total_duplicate_titles FROM duplicate_titles;	Reviewer didn't make any changes. they correctly count the number of distinct titles that appear more than once (duplicates) via a CTE and COUNT(*) over groups with COUNT(*) > 1.	✅	

Which artist release the most CDs?	SELECT artist FROM torrents GROUP BY artist ORDER BY count(groupName) DESC LIMIT 1	WITH artist_counts AS (SELECT artist, COUNT(*) AS cd_count FROM torrents GROUP BY artist), max_count AS (SELECT MAX(cd_count) AS max_cd_count FROM artist_counts) SELECT ac.artist AS Artist, ac.cd_count AS CD_Count FROM artist_counts ac JOIN max_count mc ON ac.cd_count = mc.max_cd_count ORDER BY ac.artist ASC;	❌	Unreviewed query is incorrect because it does not filter to CDs; it counts all releases per artist, not specifically CDs. The Benchmark-Proposed Query does not filter to CDs and orders by COUNT(groupName), which may ignore NULLs and counts all release types, so it does not answer the question.	WITH cd_releases AS (SELECT artist, COUNT(*) AS cd_count FROM torrents WHERE LOWER(releasetype) = 'cd' GROUP BY artist), max_count AS (SELECT MAX(cd_count) AS max_cd_count FROM cd_releases) SELECT cr.artist AS Artist, cr.cd_count AS CD_Count FROM cd_releases cr CROSS JOIN max_count mc WHERE cr.cd_count = mc.max_cd_count ORDER BY cr.cd_count DESC, cr.artist ASC LIMIT 1;	Reviewed query is correct because it filters to CDs (WHERE LOWER(releasetype) 'cd'), aggregates counts per artist, finds the maximum, and returns the artist with the most CD releases.	✅	

How many downloads of ep and album respectively ?	SELECT sum(totalSnatched) FROM torrents WHERE releaseType = "ep" UNION SELECT sum(totalSnatched) FROM torrents WHERE releaseType = "album"	WITH filtered AS (
		  SELECT
		    releasetype,
		    totalsnatched
		  FROM torrents
		  WHERE releasetype IN ('album') -- Corrected to match the schema
		    AND totalsnatched IS NOT NULL
		)
		SELECT
		  releasetype AS release_type,
		  SUM(totalsnatched) AS total_downloads
		FROM filtered
		GROUP BY releasetype
		ORDER BY releasetype;
			✗	Unreviewed query only includes 'album' and omits 'ep', so it doesn’t answer the question.	WITH filtered AS (
					  SELECT
					    releasetype,
					    totalsnatched
					  FROM torrents
					  WHERE releasetype IN ('ep', 'album')
					)
					SELECT
					  SUM(totalsnatched) AS total_downloads
					FROM filtered;
						Rreviewed query is incorrect because it sums totalsnatched for both 'ep' and 'album' together without grouping by releasetype.	✗	

What are the artist/groups that released only one CD?	SELECT artist FROM torrents GROUP BY artist HAVING count(*) = 1	WITH album_counts AS (SELECT artist AS artist_name, COUNT(DISTINCT groupid) AS cd_count FROM torrents WHERE releasetype = 'album' GROUP BY artist) SELECT artist_name FROM album_counts WHERE cd_count = 1;	✅	The Benchmark-Proposed Query counts total torrent rows per artist regardless of release type and without deduplicating by release group, so it (a) excludes artists with one album but multiple torrents for it, and (b) may include artists with a single non-album row.	WITH album_counts AS (SELECT artist AS artist_name, COUNT(DISTINCT groupid) AS cd_count FROM torrents WHERE releasetype = 'album' GROUP BY artist) SELECT artist_name FROM album_counts WHERE cd_count = 1;	Reviewed query is correct because it counts distinct album (groupid) releases per artist restricted to CDs (interpreted as releasetype = 'album') and returns artists with exactly one such release. Using COUNT(DISTINCT groupid) prevents multiple torrents/encodes of the same CD from inflating the count.	✅	
What are the actors who have had releases after 2010?	SELECT artist FROM torrents WHERE groupYear > 2010 GROUP BY artist	SELECT DISTINCT artist AS actor FROM torrents WHERE groupyear > 2010;	✅	Benchamrk and unreviwed queries filter rows with groupyear/groupYear > 2010 and return a de-duplicated list of names, satisfying “actors who have had releases after 2010.”Unreviewed query uses DISTINCT (and aliases as actor); The Benchmark-Proposed Query uses GROUP BY artist. These are equivalent for this purpose.	SELECT DISTINCT artist AS actor FROM torrents WHERE groupyear > 2010;	All three correctly return the unique set of artists (actors) with releases after 2010.Benchamrk query uses GROUP BY artist; system generateds use DISTINCT with an alias. The groupYear/groupyear casing difference is inconsequential for unquoted identifiers, and aliasing to actor is cosmetic.	✅	

TheHistoryofBaseball Database

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation	Annotator Comment
What is the average pay for players inducted into the hall of fame?	SELECT avg(T2.salary) FROM salary as T2 JOIN hall_of_fame as T1 ON T1.player_id = T2.player_id WHERE T1.inducted = "Y"	SELECT AVG(s.salary) AS avg_salary_hof_players FROM hall_of_fame h INNER JOIN salary s ON s.player_id = h.player_id WHERE h.inducted = 'Y' AND h.category = 'Player';	❌	Unreviwed query correctly joins hall_of_fame to salary by player_id, filters to inducted = 'Y', and computes AVG(salary), yielding the average pay for Hall-of-Fame inductees across their salary records.	SELECT AVG(s.salary) AS avg_salary_hof_players FROM hall_of_fame h INNER JOIN salary s ON s.player_id = h.player_id WHERE h.inducted = 'Y' AND h.category = 'Player';	Reviewed query is same as the unreviwed query and filters on plyers against benchmark proposed query	❌	
What is the average pay for players not inducted into the hall of fame?	SELECT avg(T2.salary) FROM salary as T2 JOIN hall_of_fame as T1 ON T1.player_id = T2.player_id WHERE T1.inducted = "N"	SELECT AVG(s.salary) FROM salary s WHERE NOT EXISTS (SELECT 1 FROM salary h WHERE h.player_id = s.player_id AND h.inducted = 'Y');	❌	The Benchmark-Proposed Query correctly joins salaries to hall_of_fame by player_id and averages salaries where inducted = 'N', matching "players not inducted." Unreviewed query is not a valid SQL query.	SELECT AVG(s.salary) FROM salary s WHERE NOT EXISTS (SELECT 1 FROM salary h WHERE h.player_id = s.player_id AND h.inducted = 'Y');	Unreviewed query and Reviewed query are incorrect because they look for inducted = 'Y' in the salary table (wrong table) and thus fails to correctly identify Hall of Fame induction status; the NOT EXISTS should reference hall_of_fame, not salary.	❌	

What is the average lifespan of a baseball player?	SELECT avg(death_year - birth_year) FROM player	WITH parsed_players AS (SELECT p.birth_year, p.birth_month, p.birth_day, p.death_year, p.death_month, p.death_day, NULLIF(trim(p.birth_year), '')::float8 AS byear_f, NULLIF(trim(p.birth_month), '')::float8 AS bmonth_f, NULLIF(trim(p.birth_day), '')::float8 AS bday_f, NULLIF(trim(p.death_year), '')::float8 AS dyear_f, NULLIF(trim(p.death_month), '')::float8 AS dmonth_f, NULLIF(trim(p.death_day), '')::float8 AS dday_f FROM player p), valid_dates AS (SELECT make_date(byear_f::int, GREATEST(1, LEAST(12, COALESCE(bmonth_f::int, 1))), GREATEST(1, LEAST(28, COALESCE(bday_f::int, 1)))) AS birth_date, make_date(dyear_f::int, GREATEST(1, LEAST(12, COALESCE(dmonth_f::int, 1))), GREATEST(1, LEAST(28, COALESCE(dday_f::int, 1)))) AS death_date FROM parsed_players WHERE byear_f IS NOT NULL AND bmonth_f IS NOT NULL AND bday_f IS NOT NULL AND dyear_f IS NOT NULL AND dmonth_f IS NOT NULL AND dday_f IS NOT NULL AND byear_f > 0 AND dyear_f > 0), ages AS (SELECT EXTRACT(EPOCH FROM	✗	The Benchmark-Proposed Query merely averages death_year birth_year, ignoring months/days (off by up to almost a year per player), and doesn't guard against invalid/placeholder values. It's an imprecise approximation, not a correct computation of lifespan.	WITH parsed_players AS (SELECT p.birth_year, p.birth_month, p.birth_day, p.death_year, p.death_month, p.death_day, NULLIF(trim(p.birth_year), '')::float8 AS byear_f, NULLIF(trim(p.birth_month), '')::float8 AS bmonth_f, NULLIF(trim(p.birth_day), '')::float8 AS bday_f, NULLIF(trim(p.death_year), '')::float8 AS dyear_f, NULLIF(trim(p.death_month), '')::float8 AS dmonth_f, NULLIF(trim(p.death_day), '')::float8 AS dday_f FROM player p), valid_dates AS (SELECT make_date(byear_f::int, GREATEST(1, LEAST(12, COALESCE(bmonth_f::int, 1))), GREATEST(1, LEAST(28, COALESCE(bday_f::int, 1)))) AS birth_date, make_date(dyear_f::int, GREATEST(1, LEAST(12, COALESCE(dmonth_f::int, 1))), GREATEST(1, LEAST(28, COALESCE(dday_f::int, 1)))) AS death_date FROM parsed_players WHERE byear_f IS NOT NULL AND bmonth_f IS NOT NULL AND bday_f IS NOT NULL AND dyear_f IS NOT NULL AND dmonth_f IS NOT NULL AND dday_f IS NOT NULL AND byear_f > 0 AND dyear_f > 0), ages AS (SELECT EXTRACT(EPOCH FROM (death_date - birth_date)) / (365.2425*24*60*60) AS lifespan_years FROM valid_dates WHERE death_date >= birth_date)	Unreviewed query and Reviewed query correctly compute average lifespan by constructing valid birth/death dates, filtering to complete and plausible records, and calculating age in years using date arithmetic.	✗	

		(death_date - birth_date)) / (365.2425*24*60*60) AS lifespan_years FROM valid_dates WHERE death_date >= birth_date) SELECT AVG(lifespan_years) AS avg_lifespan_years FROM ages;			SELECT AVG(lifespan_years) AS avg_lifespan_years FROM ages;				

Which is the most popular voting method for Hall of Fame in 2000?	SELECT votedby FROM hall_of_fame WHERE yearid = "2000" GROUP BY votedby ORDER BY count(*) DESC LIMIT 1	WITH votes_2000 AS (SELECT votedby AS voting_method, COUNT(*) AS vote_count FROM votedby WHERE yearid = 2000 GROUP BY votedby) SELECT voting_method, vote_count FROM votes_2000 ORDER BY vote_count DESC, voting_method LIMIT 1;	❌	Unreviewed query queries a non-existent table (votedby) instead of the hall_of_fame table.	WITH votes_2000 AS (SELECT votedby AS voting_method, COUNT(*) AS vote_count FROM hall_of_fame WHERE yearid = 2000 GROUP BY votedby) SELECT voting_method, vote_count FROM votes_2000 ORDER BY vote_count DESC, voting_method LIMIT 1;	Reviewed query correctly filters hall_of_fame for year 2000, groups by votedby, counts votes, orders by count (with a deterministic tie-breaker), and returns the top voting method (and its count).	✅	

How many players weigh over 200 pounds?	SELECT count(*) FROM player WHERE weight > 200	WITH parsed_players AS (
		  SELECT
		    p.player_id,
		    NULLIF(TRIM(p.weight), ''):: numeric AS weight_num
		  FROM player p
		),
		hof_over_200 AS (
		  SELECT
		    h.player_id
		  FROM hall_of_fame h
		  INNER JOIN parsed_players p ON p.player_id = h.player_id
		  WHERE p.weight_num > 200
		)
		SELECT COUNT(*) AS count_over_200
		FROM hof_over_200;
			✗	The Benchmark-Proposed Query correctly counts all players in the player table with weight > 200, matching the question's scope.
				Unreviewed query incorrectly limits the count to players who appear in the hall_of_fame table and can double-count players with multiple hall_of_fame records, so it does not answer the general question.
					SELECT COUNT(*) AS count_over_200
					FROM player p
					WHERE NULLIF(TRIM(p.weight), ''):: numeric > 200;
						Reviewed query is unreliable: it assumes weight is stored as text and uses TRIM/cast, which would fail or be unnecessary if weight is a numeric column (common), and may error on non-numeric strings.	✗	

Who is the winner of Rookie of the Year in 2010?	SELECT player_id FROM player_award WHERE year = 2010 AND award_id = "Rookie of the Year"	SELECT p.name_first AS first_name, p.name_last AS last_name
		FROM player_award_vote pa
		JOIN player p ON pa.player_id = p.player_id
		WHERE pa.award_id = 'Rookie of the Year'
		  AND pa.year = 2010;
			✗	Unreviewed query is incorrect because player_award_vote contains all vote-getters, not just the winner; without filtering to rank=1 or max points, it will include non-winners.
				The Benchmark-Proposed Query only returns player_id (not names), so it doesn't answer "who," and it uses double quotes for the string literal, which can be invalid in standard SQL.
					SELECT p.name_first AS first_name, p.name_last AS last_name
					FROM player_award_vote pa
					JOIN player p ON pa.player_id = p.player_id
					WHERE pa.award_id = 'Rookie of the Year'
					  AND pa.year = 2010;
						This query is incorrect because it queries the player_award_vote table instead of player_award.	✗	

Who is the highest paid player since 2010?	SELECT player_id FROM salary WHERE year >= 2010 ORDER BY salary DESC LIMIT 1	AI Query Generation Failed	❌	The Benchmark-Proposed query filters salaries to year >= 2010, orders by salary descending, and returns the top player_id, correctly identifying the player with the highest single-season salary since 2010. Unreviewed query is not a valid SQL query and cannot answer the question.	SELECT p.name_first, p.name_last, s.salary, s.year FROM salary s JOIN player p ON p.player_id = s.player_id WHERE s.year >= 2010 ORDER BY s.salary DESC LIMIT 1;	Reviewed query correctly filters years since 2010, orders by salary descending to find the highest single-season salary, and joins to return the player's name (who), satisfying the question.	✅	In this example LLM has generated on expected results after several try by orchestration. To be clear, Agents are prompted to generate response in json format but sometimes they don't generated response in the asked format.

Which birth place has the most player awards?	SELECT birth_country FROM player as T1 JOIN player_award as T2 ON T1.player_id = T2.player_id GROUP BY T1.birth_country ORDER BY count(*) DESC LIMIT 1	WITH awards_by_birthplace AS (SELECT p.birth_city, p.birth_state, p.birth_country, COUNT(pa.award_id) AS award_count FROM player_award_vote pa INNER JOIN player p ON p.player_id = pa.player_id GROUP BY p.birth_city, p.birth_state, p.birth_country), ranked AS (SELECT birth_city, birth_state, birth_country, award_count, RANK() OVER (ORDER BY award_count DESC) AS rnk FROM awards_by_birthplace) SELECT birth_city, birth_state, birth_country, award_count FROM ranked WHERE rnk = 1;	❌	Unreviewed query counts entries from player_award_vote, which are votes, not awards, so it answers a different question. The Benchmark-Proposed Query groups only by birth_country, answering a different question (“Which country...”), and uses LIMIT 1 to arbitrarily break ties.	WITH awards_by_birthplace AS (SELECT p.birth_country, COUNT(pa.award_id) AS award_count FROM player_award pa INNER JOIN player p ON p.player_id = pa.player_id GROUP BY p.birth_country) SELECT birth_country, award_count FROM awards_by_birthplace WHERE award_count > 10;	Reviewed query filters by award_count > 10 instead of selecting the highest count, so it may exclude the true birthplace with the most awards.	❌	
What were the years when any special elections happened in hall of fame?	SELECT DISTINCT yearid FROM hall_of_fame WHERE needed_note != ""	SELECT DISTINCT yearid AS special_election_year FROM hall_of_fame WHERE category = 'Special' ORDER BY special_election_year ASC;	❌	Unreviewed query incorrectly targets special elections by filtering rows where the election is labeled as Special (category = 'Special') and returns the distinct years those occurred.	SELECT DISTINCT yearid FROM hall_of_fame WHERE votedby = 'Centennial' ORDER BY yearid ASC;	Reviewed query restricts to votedby = 'Centennial', capturing only a specific special case and not all special elections.	❌	

<table>
<tr>
<td>For every award, who is the oldest winner?</td>
<td>SELECT T1.player_id, T1.award_id , max(T1.year - T2.birth_year) FROM player_award as T1 JOIN player as T2 ON T1.player_id = T2.player_id GROUP BY T1.award_id</td>
<td>WITH winners AS (
SELECT award_id, year, league_id, player_id
FROM player_award_vote
),
ages AS (
SELECT
w.award_id,
w.player_id,
p.name_first,
p.name_last,
w.year,
(MAKE_DATE(w.year, 7, 1) - MAKE_DATE(
p.birth_year,

COALESCE(p.birth_month, 7),
COALESCE(p.birth_day, 1)
)) AS age_interval
FROM winners w
JOIN player p ON p.player_id = w.player_id
WHERE p.birth_year IS NOT NULL
),
ranked AS (
SELECT
award_id,
player_id,
name_first,
name_last,
RANK() OVER (PARTITION BY award_id ORDER BY age_interval DESC) AS rnk
FROM ages
)
SELECT
award_id AS award,
name_first || ' ' || name_last AS winner_name
FROM ranked
WHERE rnk = 1
ORDER BY award, winner_name;</td>
<td>❌</td>
<td>Unreviewed query is incorrect: it doesn’t restrict to winners (includes anyone who received votes), so it can return non-winners; it also uses an arbitrary reference date and imputes missing birth components.</td>
<td>WITH winners_per_award_year AS (
SELECT
pav.award_id,
pav.year,
pav.player_id,
pav.points_won,
RANK() OVER (PARTITION BY pav.award_id, pav.year ORDER BY pav.points_won DESC) AS rnk
FROM player_award_vote pav
),
tied_winners AS (
SELECT award_id, year, player_id
FROM winners_per_award_year
WHERE rnk = 1
),
winner_ages AS (
SELECT
tw.award_id,
tw.year,
tw.player_id,
p.name_first,
p.name_last,
AGE(
MAKE_DATE(tw.year::int, 12, 31),

MAKE_DATE(NULLIF(p.birth_year, '')::float8::int, NULLIF(p.birth_month, '')::float8::int, NULLIF(p.birth_day, '')::float8::int)
) AS age_interval
FROM tied_winners tw
JOIN player p ON p.player_id = tw.player_id
WHERE
NULLIF(p.birth_year, '') IS NOT NULL
AND NULLIF(p.birth_month, '') IS NOT NULL
AND NULLIF(p.birth_day, '') IS NOT NULL
),
oldest_winners AS (
SELECT
award_id,
player_id,
name_first,
name_last,
year,
RANK() OVER (PARTITION BY award_id</td>
<td>Reviewed query correctly identifies the actual winners per award and year (top points_won with ties), computes precise ages using full birthdates, and then selects the oldest winner per award.</td>
<td>✅</td>
<td></td>
</tr>
</table>

					ORDER BY age_interval DESC) AS age_rank FROM winner_ages) SELECT award_id AS award, name_first || ' ' || name_last AS winner_name FROM oldest_winners WHERE age_rank = 1 ORDER BY award, winner_name;				

Average lifespan of hall of fame players	SELECT avg(T1.death_year - T1.birth_year) FROM player as T1 JOIN hall_of_fame as T2 ON T1.player_id = T2.player_id WHERE T2.inducted = "Y"	WITH hof_players AS (
		  SELECT p.player_id,
		        NULLIF(p.birth_year, '') AS birth_year,
		        NULLIF(p.birth_month, '') AS birth_month,
		        NULLIF(p.birth_day, '') AS birth_day,
		        NULLIF(p.death_year, '') AS death_year,
		
		NULLIF(p.death_month, '') AS death_month,
		        NULLIF(p.death_day, '') AS death_day
		  FROM hall_of_fame h
		  INNER JOIN player p ON h.player_id = p.player_id
		  WHERE h.inducted = 'Y'
		),
		parsed_dates AS (
		  SELECT
		    player_id,
		    -- Parse birth date parts;
			✗	The Benchmark-Proposed Query correctly joins hall_of_fame to player, filters to inducted = 'Y', and computes the average of death_year birth_year. Rows for living players (NULL death_year) are naturally excluded from the AVG, matching the notion of "lifespan."
				Unreviewed query is incomplete/invalid (the CTE is truncated and no final computation/selection is made), so it does not produce the requested result.
					SELECT
					AVG(CAST(p.death_year AS NUMERIC) - CAST(p.birth_year AS NUMERIC)) AS avg_lifespan_years
					FROM hall_of_fame h
					JOIN player p ON p.player_id = h.player_id
					WHERE h.inducted = 'Y';
						Reviewed query does not account for month/day adjustments, so it overestimates age if the death date is before the birthday in that year	✗	

For award winners, what's average weight for each position	SELECT avg(T1.weight) FROM player as T1 JOIN player_award as T2 ON T1.player_id = T2.player_id GROUP BY notes	WITH award_winners AS (
		  SELECT DISTINCT player_id
		  FROM player_award
		)
		SELECT AVG(CAST(p.weight AS NUMERIC)) AS avg_weight
		FROM award_winners aw
		JOIN player p ON p.player_id = aw.player_id
		WHERE p.weight IS NOT NULL AND p.weight ~ '^[0-9]+(\.[0-9]+)?$';
			✗	Unreviewed query is incorrect because it does not group by position; it returns a single overall average.	WITH award_winners AS (
					  SELECT DISTINCT pa.player_id
					  FROM player_award pa
					),
					clean_player AS (
					  SELECT p.player_id,
					        p.position,
					        NULLIF(trim(p.weight), ") AS weight_text
					  FROM player p
					)
					SELECT AVG(NULLIF(regexp_replace(cp.weight_text, '[^0-9.\-]+', ", 'g'), ")::numeric) AS avg_weight
					FROM award_winners aw
					JOIN clean_player cp ON cp.player_id = aw.player_id
					WHERE cp.weight_text IS NOT NULL
					  AND regexp_replace(cp.weight_text, '[^0-9.\-]+', ", 'g') ~ '^[+-]?[0-9]+(\.[0-9]+)?$';
						Reviewed is incorrect because it does not include a GROUP BY cp.position.	✗	

Which player get the highest score in 1971?	SELECT player_id FROM player_award_vote WHERE year = "1971" ORDER BY points_won DESC LIMIT 1	SELECT NULL::text AS name_first, NULL::text AS name_last, NULL::int AS highest_score WHERE FALSE;	❌	It filters records to 1971, orders by points_won (interpreting "score" as points won) descending, and returns the top player_id, which identifies the player with the highest score that year. Unreviewed query is a dummy query that always returns no rows and does not compute or retrieve any player or score.	SELECT p.name_first, p.name_last, pav.points_won AS highest_score FROM player_award_vote pav JOIN player p ON p.player_id = pav.player_id WHERE pav.year = 1971 ORDER BY pav.points_won DESC NULLS LAST LIMIT 1;	Reviewed query correctly filters for 1971, orders by points_won descending with NULLS LAST to avoid NULLs being selected, joins to get the player's name, and returns the top scorer with their score.	✅	
What defense position the player ID willite01 is?	SELECT DISTINCT notes FROM player_award WHERE player_id = "willite01"	SELECT NULL::text AS defense_position FROM hall_of_fame WHERE player_id = 'willite01' LIMIT 0;	❌	The Benchmark-Proposed Query targets a player-related table and returns textual information (notes) associated with player_id 'willite01', which is the only one of the two that could contain the player's defensive position; DISTINCT avoids duplicates. Unreviewed query is guaranteed to return no rows (LIMIT 0) and selects a constant NULL instead of any real column; hall_of_fame also doesn't store a defensive position, so it cannot answer the question.	SELECT DISTINCT pos AS defense_position FROM batting WHERE player_id = 'willite01';	Reviewed query is incorrect because it queries the batting table instead of the fielding table.	❌	

How many players were awarded more than ten times?	SELECT count(*) FROM (SELECT player_id FROM player_award GROUP BY player_id HAVING count(*) > 10)	SELECT COUNT(*) AS count_players_more_than_ten_awards FROM (SELECT pa.player_id, COUNT(*) AS award_count FROM player_award pa GROUP BY pa.player_id HAVING COUNT(*) > 10) t;	❌	Unreviewed query groups by player_id, filters players with COUNT(*) > 10, and then counts those groups, exactly returning the number of players awarded more than ten times. The Benchmark-Proposed Query lacks an alias for the derived table in the FROM clause, making it invalid in most SQL dialects. With an alias, it would be equivalent.	SELECT COUNT(*) AS count_players_more_than_ten_awards FROM (SELECT pa.player_id, COUNT(*) AS award_count FROM player_award pa GROUP BY pa.player_id HAVING COUNT(*) > 10) t;	Reviewer didn't make any changes. Unreviewed query and Reviewed query correctly group by player_id, filter players with COUNT(*) > 10, and count how many such players exist, with a proper alias for the derived table.	✅		
Who is the player had the highest salary in 2015?	SELECT player_id FROM salary WHERE year = "2015" ORDER BY salary DESC LIMIT 1	AI Query Generation Failed	❌	The Benchmark-Proposed Query correctly filters for 2015, orders by salary descending, and returns the top player_id, which identifies the player with the highest salary that year. Unreviewed query is not a valid SQL query.	SELECT p.name_first, p.name_last FROM salary s JOIN player p ON p.player_id = s.player_id WHERE s.year = 2015 ORDER BY s.salary ASC LIMIT 1;	Reviewed query is incorrect because it orders by salary ASC instead of descending.	❌		

StudentMathScore Database

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation	Annotator Comment
What are the top five states in descending order in terms of revenue provided to school districts?	SELECT T2.state FROM FINREV_FED_KEY_17 as T2 JOIN FINREV_FED_17 as T1 ON T1.state_code = T2.State_Code GROUP BY T1.state_code ORDER BY sum(t_fed_rev)	SELECT state_code, SUM(t_fed_rev) AS total_revenue FROM finrev_fed_17 WHERE state_code IS NOT NULL GROUP BY state_code ORDER BY total_revenue DESC LIMIT 5;	✅	Unreviewed query correctly sums revenue by state, orders by total revenue in descending order, and returns the top five states. The Benchmark-Proposed Query orders ascending (no DESC), has no LIMIT 5, and selects a non-aggregated column while grouping only by state_code (nondeterministic/invalid in many SQL dialects), so it does not answer the question.	SELECT k.state_code, SUM(f.t_fed_rev) AS total_revenue FROM finrev_fed_17 f JOIN finrev_fed_key_17 k ON k.state_code = f.state_code GROUP BY k.state_code ORDER BY total_revenue DESC LIMIT 5;	Reviewed query correctly aggregates federal revenue by state_code, orders the totals in descending order, and limits to the top five, matching the question's intent.	✅	

What are the top five states in descending order in terms of the number of school districts per state?	SELECT T2.state FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 ON T1.state_code = T2.state_code GROUP BY T2.state ORDER BY count(school_district) DESC LIMIT 5	WITH valid_districts AS (
		  SELECT
		    state_code,
		    nces_id
		  FROM finrev_fed_17
		  WHERE nces_id IS NOT NULL AND nces_id <> '' AND state_code IS NOT NULL
		),
		district_counts AS (
		  SELECT
		    state_code,
		    COUNT(DISTINCT nces_id) AS district_count
		  FROM valid_districts
		  GROUP BY state_code
		)
		SELECT
		  state_code,
		  district_count
		FROM district_counts
		ORDER BY district_count DESC, state_code ASC
		LIMIT 5;
			✅	Unreviewed query correctly counts distinct districts (via nces_id) per state_code, filters out null/empty IDs, orders by the count descending, and limits to five. This directly answers “top five states by number of school districts.” The Benchmark-Proposed Query joins on state_code only and orders by count(school_district), which can overcount due to join multiplicity and lacks COUNT(DISTINCT). It also counts a potentially non-unique/non-null-safe column and may be inflated if either table has multiple rows per state.	WITH valid_districts AS (
					  SELECT
					    state_code,
					    nces_id
					  FROM finrev_fed_17
					  WHERE nces_id IS NOT NULL AND nces_id <> '' AND state_code IS NOT NULL
					),
					district_counts AS (
					  SELECT
					    state_code,
					    COUNT(DISTINCT nces_id) AS district_count
					  FROM valid_districts
					  GROUP BY state_code
					)
					SELECT
					  state_code,
					  district_count
					FROM district_counts
					ORDER BY district_count DESC, state_code ASC
					LIMIT 5;
						Unreviewed query and Reviewed query correctly count distinct districts (via nces_id) per state_code, filter invalid IDs, sort by count descending, and limit to five, matching the intent.	✅	

For each state, when was the last time the average score data was collected?	SELECT state, max(year) FROM NDECoreExcel_Math_Grade8 GROUP BY state	WITH last_year AS (SELECT MAX(year) AS last_collection_year FROM ndecoreexcel_math_grade8) SELECT k.state, l.last_collection_year FROM finrev_fed_key_17 AS k CROSS JOIN last_year AS l;	❌	The Benchmark-Proposed Query correctly computes the latest year with data per state by grouping on state and taking MAX(year) from the average-score dataset. Unreviewed query returns the same global MAX(year) for all states and pulls the state list from a different table (finrev_fed_key_17), which may not match the math dataset and does not provide per-state last collection years.	SELECT state, MAX(year) AS last_collection_year FROM ndecor_excel_math_grade 8 GROUP BY state;	Reviewed query is incorrect due to referencing a misspelled/nonexistent table name (ndecor_excel_math_grade 8). Even though the aggregation logic matches	❌	The example shows that LLMs might make mistake in cases that the table or column name are tool long and include special caracters
How much federal funding did Faifaix County, VA schools receive in all forms?	SELECT t_fed_rev FROM FINREV_FED_17 WHERE school_district = "FAIRFAX CO SCHS"	SELECT SUM(t_fed_rev) AS total_federal_funding FROM finrev_fed_17 WHERE state_code = 51 AND UPPER(school_district) LIKE '%FAIFAIX%';	❌	The Benchmark-Proposed Query is correct because it selects the total federal revenue (t_fed_rev) for the specific district "FAIRFAX CO SCHS," which corresponds to Fairfax County Public Schools, matching the request for federal funding "in all forms." Unreviewed query is incorrect because it misspells "FAIRFAX" as "FAIFAIX," yielding no matches (and thus an incorrect total). Even if corrected, using LIKE '%FAIRFAX%' could incorrectly include Fairfax City schools, overcounting the county's funding.	SELECT SUM(t_fed_rev) AS total_federal_funding FROM finrev_fed_17 WHERE state_code = 51;	Reviewed query is incorrect because it sums federal revenue for all districts in Virginia (state_code = 51), not specifically Fairfax County.	❌	

How much did the federal government spend in No Child Left Behind funding in 2017?	SELECT sum(c14) FROM FINREV_FED_17 WHERE yr_data = 17	SELECT SUM(c14) AS total_nclb_spending_2017 FROM finrev_fed_17 WHERE c14 IS NOT NULL;	✅	Unreviewed query correctly sums the NCLB field (c14) across the 2017-specific table; SUM ignores NULLs, so the NULL filter is harmless and the table already scopes to 2017. The Benchmark-Proposed Query adds a yr_data = 17 filter on a table that appears year-specific; this column may not exist or may store 2017 (not 17), making the filter unnecessary or incorrect.	SELECT SUM(c14) AS total_nclb_spending_2017 FROM finrev_fed_17 WHERE c14 IS NOT NULL;	System-generateds sum c14 (No Child Left Behind funding) across the 2017 table finrev_fed_17. The c14 IS NOT NULL filter is redundant but harmless, and the table name indicates it already contains 2017 data.	✅	
How many Wisconsin school districts receive federal funding?	SELECT count(*) FROM FINREV_FED_17 WHERE school_district LIKE "%Wisconsin%"	SELECT COUNT(DISTINCT idcensus) AS wi_districts_with_federal_funding_count FROM finrev_fed_17 WHERE state_code = 55 AND t_fed_rev > 0;	✅	Unreviewed query correctly filters to Wisconsin using state_code = 55 (Wisconsin's FIPS), identifies districts receiving federal funds via t_fed_rev > 0, and counts unique districts with COUNT(DISTINCT idcensus). The Benchmark-Proposed Query does not check whether districts receive federal funding, relies on school_district names containing "Wisconsin" (which most district names will not), may miscount due to duplicates, and is semantically unreliable for state filtering.	SELECT COUNT(DISTINCT idcensus) AS wi_districts_with_federal_funding_count FROM finrev_fed_17 WHERE state_code = 55 AND t_fed_rev > 0;	Unreviewed query and Reviewed query correctly count distinct Wisconsin districts (state_code = 55) with positive federal revenue (t_fed_rev > 0), which matches "receive federal funding."	✅	

Which state performs best on the NAEP 8th grade math test?	SELECT state FROM NDECoreExcel_Math_Grade8 ORDER BY average_scale_score DESC LIMIT 1	WITH filtered_scores AS (SELECT state, year, average_scale_score FROM ndec0reexcel_math_grade8 WHERE year = 2017), ranked_states AS (SELECT state, year, average_scale_score, RANK() OVER (PARTITION BY year ORDER BY average_scale_score DESC) AS rnk FROM filtered_scores) SELECT state AS best_state, average_scale_score AS best_avg_score, year FROM ranked_states WHERE rnk = 1;	✅	Unreviewed query correctly identifies the top-performing state for a specific administration (2017), ranking states by average_scale_score and returning all top ties, which aligns with the usual intent of “performs best” (latest test cycle). The Benchmark-Proposed Query ignores the year, so it returns the single highest state-year observation across all years (not the best state for a given test cycle or the current year), does not handle ties, and can be non-deterministic with LIMIT 1.	WITH latest_year AS (SELECT MAX(year) AS year FROM ndecoreexcel_math_grade8 WHERE all_students = 'All students.'), scores AS (SELECT s.state, s.year, s.average_scale_score FROM ndecoreexcel_math_grade8 s JOIN latest_year ly ON s.year = ly.year WHERE s.all_students = 'All students.'), ranked AS (SELECT state, year, average_scale_score, RANK() OVER (ORDER BY average_scale_score DESC) AS rnk FROM scores) SELECT state AS best_state, average_scale_score AS best_avg_score, year FROM ranked WHERE rnk = 1;	Reviewed query is correct because it identifies the latest available year for the overall population (“All students.”), then ranks states by average_scale_score and returns the top performer(s), aligning with the likely intent to report the most recent overall state performance.	✅	

Which federally-funded state performs worst on the NAEP math exam for eighth graders?	SELECT state FROM NDECoreExcel_Math_Grade8 ORDER BY average_scale_score LIMIT 1	WITH funded_states AS (
		  SELECT DISTINCT state
		  FROM finrev_fed_key_17
		),
		naep_2017 AS (
		  SELECT year, average_scale_score
		  FROM ndecroreexcel_math_grade8
		  WHERE year = 2017
		)
		SELECT NULL::text AS state, NULL::numeric AS math_score
		WHERE FALSE
		UNION ALL
		SELECT s.state AS state, n.average_scale_score AS math_score
		FROM ndecroreexcel_math_grade8 n
		JOIN funded_states s ON FALSE
		WHERE n.year = 2017
		ORDER BY math_score ASC
		LIMIT 1;
			✗	The Benchmark-Proposed Query correctly identifies the worst-performing state by ordering states by average_scale_score and returning the lowest one, which answers the question of "which state performs worst" on the Grade 8 NAEP math exam.
				Unreviewed query is incorrect because it uses JOIN ... ON FALSE (and a preceding SELECT ... WHERE FALSE), guaranteeing no rows are returned, so it cannot produce the required state.
					SELECT n.state, n.average_scale_score
					FROM ndecoreexcel_math_grade8 n
					JOIN finrev_fed_key_17 f
					ON f.state = n.state
					WHERE n.year = (SELECT MIN(year)
					FROM ndecoreexcel_math_grade8)
					ORDER BY n.average_scale_score ASC
					LIMIT 1;
						Reviewed query is incorrect because it uses MIN(year) instead of the latest year.
						Benchmark query ignores the “federally-funded” constraint and doesn’t restrict to a single year, so it can return a non-funded state and/or a score from any year.
							✗	

What is the average NAEP math score for all eighth graders in federally-funded states?	SELECT avg(average_scale_score) FROM NDECoreExcel_Math_Grade8	WITH
		federally_funded_states AS
		(
		  SELECT DISTINCT
		finrev_fed_17.state_code
		  FROM finrev_fed_17
		  WHERE
		finrev_fed_17.t_fed_rev > 0
		),
		naep_2017 AS (
		  SELECT
		ndecoreexcel_math_grade8.
		average_scale_score
		  FROM
		ndecoreexcel_math_grade8
		  WHERE
		ndecoreexcel_math_grade8.
		year = 2017
		)
		SELECT
		AVG(naep_2017.average_scale_score)::numeric AS
		avg_math_score
		FROM naep_2017;
			✅	Unreviewed query computes the average NAEP Grade 8 math score for 2017, aligning with the year implied by the federally funded finance table. Although it defines the federally funded states CTE and doesn't join to it, this omission is likely benign because all states typically have positive federal revenue; thus the average across 2017 state records matches the intended result.
				The Benchmark-Proposed Query ignores the federally funded criterion and does not filter by year, averaging across the entire table (potentially multiple years/jurisdictions), so it does not specifically answer the question.
					WITH
					federally_funded_states
					AS (
					  SELECT DISTINCT
					k.state
					  FROM finrev_fed_17 f
					  JOIN finrev_fed_key_17
					k ON k.state_code =
					f.state_code
					  WHERE f.t_fed_rev > 0
					)
					SELECT
					AVG(n.average_scale_score)::numeric AS
					avg_math_score
					FROM
					ndecoreexcel_math_grade8
					n
					JOIN
					federally_funded_states s
					ON s.state = n.state;
						Reviewed query correctly restricts the calculation to states that receive federal funds (t_fed_rev > 0) by deriving the set of federally funded states and joining that set to the NAEP Grade 8 math data before averaging average_scale_score. This matches the intent: average score for eighth graders in federally funded states.	✅	

Which school district receive the most of federal revenue through state in Wisconsin?	SELECT T1.school_district FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 ON T1.state_code = T2.state_code WHERE T2.state = "Wisconsin" ORDER BY T1.t_fed_rev DESC LIMIT 1	WITH unavailable AS (SELECT NULL::text AS district_name, NULL::numeric AS federal_revenue_through_state WHERE FALSE) SELECT district_name, federal_revenue_through_state FROM unavailable;	❌	The Benchmark-Proposed Query joins to the state key to filter Wisconsin, orders districts by the federal revenue metric (t_fed_rev) in descending order, and returns the top district, which answers the question. Unreviewed query deliberately returns no rows (the CTE filters everything out), so it does not answer the question.	SELECT f.school_district, f.c14 AS federal_revenue_through_state FROM finrev_fed_17 f WHERE f.state = 'Wisconsin' ORDER BY f.c14 DESC LIMIT 1;	Reviewed query correctly filters Wisconsin, uses the specific "federal revenue through state" measure (C14), orders by that amount, and returns the top district.	✅	It is an example that the system sometimes return unxpected codes that are always true or false like WHERE FALSE
What is the average Title 1 fund in Virginia?	SELECT avg(T1.c14) FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 ON T1.state_code = T2.state_code WHERE T2.state = "Virginia"	SELECT AVG(c14)::numeric AS avg_title1_va FROM finrev_fed_17 WHERE state_code = 51;	✅	Unreviewed query correctly filters Virginia by its FIPS state_code (51) and computes AVG(c14), which represents Title I funds, yielding the requested average.	SELECT AVG(c14) AS avg_title1_va FROM finrev_fed_17 WHERE state_code = 51;	Reviewer didn't make any changes. System-generateds correctly compute the average of Title I funds for Virginia by filtering state_code = 51 (Virginia's FIPS code) and averaging c14.	✅	

What are the top 10 states that have the highest average math score and federal revenue they got through different categories?	SELECT T2.state, sum(c14),sum(c25) FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 ON T1.state_code = T2.state_code JOIN NDECoreExcel_Math_Grade8 as T3 ON T2.state = T3.state GROUP BY T2.state ORDER BY T3.average_scale_score DESC LIMIT 10	WITH rev_2017 AS (SELECT f.state_code, SUM(f.t_fed_rev) AS t_fed_rev, SUM(f.c14) AS c14, SUM(f.c25) AS c25 FROM finrev_fed_17 f WHERE f.yr_data = 17 GROUP BY f.state_code), math_2017 AS (SELECT m.state, AVG(m.average_scale_score)::NUMERIC(10,2) AS avg_math_score FROM ndecoreexcel_math_grade8 m WHERE m.year = 2017 GROUP BY m.state), joined AS (SELECT k.state, m.avg_math_score, r.t_fed_rev, r.c14, r.c25, RANK() OVER (ORDER BY m.avg_math_score DESC) AS rank_math, RANK() OVER (ORDER BY r.t_fed_rev DESC) AS rank_rev FROM rev_2017 r JOIN finrev_fed_key_17 k ON r.state_code = k.state_code JOIN math_2017 m ON m.state = k.state) SELECT state, avg_math_score, t_fed_rev, c14, c25 FROM joined ORDER BY (rank_math + rank_rev), rank_math, rank_rev LIMIT 10;	✗	Uneviewed query is incorrect because it: Uses a composite ranking (sum of ranks for math and revenue), changing the intended ordering (top by math) to a blended rank not asked for. Filters revenue with yr_data = 17, which likely mismatches a 2017-coded column. Joins math via state name instead of state_code, risking mismatches.	WITH rev_2017 AS (SELECT f.state_code, SUM(f.t_fed_rev) AS t_fed_rev, SUM(f.c14) AS c14, SUM(f.c25) AS c25 FROM finrev_fed_17 f WHERE f.yr_data = 2017 GROUP BY f.state_code), math_2017 AS (SELECT m.state_code, AVG(m.average_scale_score)::NUMERIC(10,2) AS avg_math_score FROM ndecoreexcel_math_grade8 m WHERE m.year = 2017 GROUP BY m.state_code) SELECT k.state AS state, m.avg_math_score, r.t_fed_rev, r.c14, r.c25 FROM rev_2017 r JOIN math_2017 m ON r.state_code = m.state_code JOIN finrev_fed_key_17 k ON r.state_code = k.state_code ORDER BY r.t_fed_rev DESC LIMIT 10;	Reviewed query is incorrect because it orders by t_fed_rev DESC instead of avg_math_score DESC	✗	

What are the schools with highest federal revenue in each federal revenue categories and what's their state average scores?	SELECT T1.school_district, max(T1.c14), T3.average_scale_score FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 ON T1.state_code = T2.state_code JOIN NDECoreExcel_Math_Grade8 as T3 ON T2.state = T3.state UNION SELECT T1.school_district, max(T1.c25), T3.average_scale_score FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 ON T1.state_code = T2.state_code JOIN NDECoreExcel_Math_Grade8 as T3 ON T2.state = T3.state	WITH TopTotal AS (SELECT 't_fed_rev' AS category, state_code, school_district, t_fed_rev AS top_value FROM finrev_fed_17 WHERE t_fed_rev IS NOT NULL AND t_fed_rev > 0 ORDER BY t_fed_rev DESC LIMIT 1), TopC14 AS (SELECT 'c14' AS category, state_code, school_district, c14 AS top_value FROM finrev_fed_17 WHERE c14 IS NOT NULL AND c14 > 0 ORDER BY c14 DESC LIMIT 1), TopC25 AS (SELECT 'c25' AS category, state_code, school_district, c25 AS top_value FROM finrev_fed_17 WHERE c25 IS NOT NULL AND c25 > 0 ORDER BY c25 DESC LIMIT 1), TopUnion AS (SELECT category, state_code, school_district, top_value FROM TopTotal UNION ALL SELECT category, state_code, school_district, top_value FROM TopC14 UNION ALL SELECT category, state_code, school_district, top_value FROM TopC25) SELECT T.category, T.school_district, T.state_code, T.top_value, N.average_scale_score AS state_average_scale_score FROM TopUnion T	✅	Unreviewed query correctly finds the top school district for each federal revenue category (t_fed_rev, c14, c25) using ORDER BY ... LIMIT 1 and then joins to retrieve their state's 2017 Grade 8 math average scale score for All students. The Benchmark-Proposed Query is invalid/nondeterministic: it uses MAX() alongside non-aggregated columns with no GROUP BY (so it can't reliably return the district associated with the max), omits the total federal revenue category, and does not constrain the state scores by year or subgroup.	WITH TopTotal AS (SELECT 't_fed_rev' AS category, state_code, school_district, t_fed_rev AS top_value FROM finrev_fed_17 WHERE t_fed_rev IS NOT NULL AND t_fed_rev > 0 ORDER BY t_fed_rev DESC LIMIT 1), TopC14 AS (SELECT 'c14' AS category, state_code, school_district, c14 AS top_value FROM finrev_fed_17 WHERE c14 IS NOT NULL AND c14 > 0 ORDER BY c14 DESC LIMIT 1), TopC25 AS (SELECT 'c25' AS category, state_code, school_district, c25 AS top_value FROM finrev_fed_17 WHERE c25 IS NOT NULL AND c25 > 0 ORDER BY c25 DESC LIMIT 1), TopUnion AS (SELECT category, state_code, school_district, top_value FROM TopTotal UNION ALL SELECT category, state_code, school_district, top_value FROM TopC14 UNION ALL SELECT category, state_code, school_district, top_value FROM TopC25) SELECT T.category, T.school_district, T.state_code, T.top_value, N.average_scale_score AS	Reviewed query is same as the unreviewed query.	✅	

		LEFT JOIN ndecoreexcel_math_grade8 N ON N.state_code = T.state_code AND N.year = 2017 AND N.all_students = 'All students';			state_average_scale_score FROM TopUnion T LEFT JOIN ndecoreexcel_math_grade8 N ON N.state_code = T.state_code AND N.year = 2017 AND N.all_students = 'All students';			

State with highest average math score	SELECT state FROM NDECoreExcel_Math_Grade8 ORDER BY average_scale_score DESC LIMIT 1	WITH scores_2017 AS (
		  SELECT
		    average_scale_score
		  FROM
		ndecoreexcel_math_grade8
		  WHERE year = 2017
		)
		SELECT
		
		MAX(average_scale_score)
		AS highest_average_score
		FROM scores_2017;
			❌	The Benchmark-Proposed Query correctly returns the state associated with the highest average_scale_score by ordering descending and limiting to one row, which matches the request for the state with the highest average math score.
				Unreviewed query is incorrect because it returns only the numeric maximum score (not the state) and arbitrarily restricts to year = 2017, which the question does not specify.
					SELECT state
					FROM (
					  SELECT state,
					AVG(average_scale_score
					) AS avg_score
					  FROM
					ndecoreexcel_math_grade8
					  GROUP BY state
					) s
					ORDER BY avg_score
					ASC
					LIMIT 1;
						Reviewed query is incorrect because it orders by avg_score ASC instead of descending.	❌	

School with highest ratio of total revenue and average scores	SELECT T1.school_district, max(T1.t_fed_rev / T3.average_scale_score) FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 on T1.state_code = T2.state_code JOIN NDECoreExcel_Math_Gra de8 as T3 ON T2.state = T3.state	WITH scores_2017 AS (SELECT state_code, average_scale_score FROM ndecoreexcel_math_grade8 WHERE year = 2017), joined AS (SELECT f.state_code, f.t_fed_rev AS total_revenue, s.average_scale_score AS avg_score, (f.t_fed_rev::numeric / NULLIF(s.average_scale_sc ore, 0)) AS revenue_to_score_ratio FROM finrev_fed_17 f INNER JOIN scores_2017 s ON f.state_code = s.state_code WHERE f.t_fed_rev IS NOT NULL AND f.t_fed_rev <> 0 AND s.average_scale_score IS NOT NULL AND s.average_scale_score <> 0) SELECT state_code, total_revenue, avg_score, revenue_to_score_ratio FROM joined ORDER BY revenue_to_score_ratio DESC LIMIT 1;	✗	Unreviewed query incorrectly computes the revenue-to-score ratio (guarding against null/zero), aligns years (2017), joins at a consistent grain (by state), and orders to return the highest ratio.	WITH scores_2017 AS (SELECT state_code, average_scale_score FROM ndecoreexcel_math_grade8 WHERE year = 2017), joined AS (SELECT f.state_code, f.t_fed_rev AS total_revenue, s.average_scale_score AS avg_score, (f.t_fed_rev::numeric / NULLIF(s.average_scale_s core, 0)) AS revenue_to_score_ratio FROM finrev_fed_17 f INNER JOIN scores_2017 s ON f.state_code = s.state_code WHERE f.t_fed_rev IS NOT NULL AND f.t_fed_rev <> 0 AND s.average_scale_score IS NOT NULL AND s.average_scale_score <> 0) SELECT state_code, total_revenue, avg_score, revenue_to_score_ratio FROM joined ORDER BY revenue_to_score_ratio DESC LIMIT 1;	Reviewed query is exactly same as the unireviwed query.	✗		

Which state spent the most revenue towards schools and whats the state average score	SELECT T2.state, T3.average_scale_score FROM FINREV_FED_KEY_17 as T2 JOIN FINREV_FED_17 as T1 ON T1.state_code = T2.state_code JOIN NDECoreExcel_Math_Grade8 as T3 ON T2.state = T3.state GROUP BY T2.state ORDER BY sum(T1.t_fed_rev) DESC LIMIT 1	WITH revenue_by_state AS (SELECT k.state AS top_state, SUM(f.t_fed_rev) AS top_state_total_fed_rev FROM finrev_fed_17 f INNER JOIN finrev_fed_key_17 k ON f.state_code = k.state_code GROUP BY k.state), top_state AS (SELECT top_state, top_state_total_fed_rev FROM revenue_by_state ORDER BY top_state_total_fed_rev DESC LIMIT 1), avg_score AS (SELECT AVG(average_scale_score)::numeric(10,2) AS avg_grade8_math_score_2017 FROM ndec0reexcel_math_grade8 WHERE year = 2017) SELECT t.top_state, t.top_state_total_fed_rev, a.avg_grade8_math_score_2017 FROM top_state t CROSS JOIN avg_score a;	❌	The Benchmark-Proposed Query aligns with the intent: it finds the state with the highest total federal revenue (sum of t_fed_rev by state) and returns that same state's average score via the join to the Grade 8 Math table. Note: it should also filter the score table to the matching year (2017) and aggregate or otherwise handle multiple rows per state to be fully robust. Unreviewed query is incorrect because it returns the national average Grade 8 math score for 2017 (AVG over all states) rather than the average score for the top-revenue state.	WITH revenue_by_state AS (SELECT k.state, SUM(f.t_fed_rev) AS total_fed_rev FROM finrev_fed_17 f JOIN finrev_fed_key_17 k ON f.state_code = k.state_code GROUP BY k.state), top_state AS (SELECT state, total_fed_rev FROM revenue_by_state ORDER BY total_fed_rev DESC LIMIT 1) SELECT t.state, t.total_fed_rev, AVG(s.average_scale_score)::numeric(10,2) AS avg_grade8_math_score_2017 FROM top_state t LEFT JOIN ndecoreexcel_math_grade8 s ON s.state = t.state GROUP BY t.state, t.total_fed_rev;	Reviewed query correctly identifies the top-revenue state and then computes that state's average score by joining to the Grade 8 math table on state and averaging that state's scores. This matches the intent: top state and that state's average score.	✅	

School with lowest ratio of total revenue and average scores	SELECT T1.school_district, min(T1.t_fed_rev / T3.average_scale_score) FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 on T1.state_code = T2.state_code JOIN NDECoreExcel_Math_Grade8 as T3 ON T2.state = T3.state	WITH scores_2017 AS (SELECT NULL::text AS nces_id, NULL::text AS school_district, AVG(average_scale_score) AS avg_score FROM ndecoreexcel_math_grade8 WHERE year = 2017), rev_valid AS (SELECT nces_id, school_district, t_fed_rev FROM finrev_fed_17 WHERE t_fed_rev IS NOT NULL AND t_fed_rev <> 0), joined AS (SELECT r.school_district, r.nces_id, r.t_fed_rev, s.avg_score FROM rev_valid r CROSS JOIN scores_2017 s WHERE s.avg_score IS NOT NULL AND s.avg_score <> 0) SELECT school_district, nces_id, t_fed_rev, avg_score AS average_scale_score, t_fed_rev / avg_score AS ratio_rev_to_score FROM joined ORDER BY ratio_rev_to_score ASC LIMIT 1;	✗	The Benchmark-Proposed Query aligns revenue to the appropriate average score by joining through state, computes the revenue-to-score ratio per district, and seeks the minimum, matching the intent of "school with lowest ratio." Unreviewed query uses a CROSS JOIN to a single overall 2017 average score (a constant denominator), so the "minimum ratio" is just the district with the smallest revenue, not the lowest revenue-to-its-score ratio. It also doesn't align scores to districts/states.	WITH scores_2017 AS (SELECT state_code AS state, AVG(average_scale_score) AS avg_score FROM ndecoreexcel_math_grade8 WHERE year = 2017 GROUP BY state_code), rev_valid AS (SELECT nces_id, school_district, t_fed_rev FROM finrev_fed_17 WHERE t_fed_rev IS NOT NULL AND t_fed_rev > 0), joined AS (SELECT r.school_district, r.nces_id, r.t_fed_rev, s.avg_score FROM rev_valid r JOIN scores_2017 s ON r.nces_id = s.state WHERE s.avg_score IS NOT NULL AND s.avg_score > 0) SELECT school_district, nces_id, t_fed_rev, avg_score AS average_scale_score, t_fed_rev / avg_score AS ratio_rev_to_score FROM joined ORDER BY ratio_rev_to_score ASC LIMIT 1;	Reviewed query is incorrect: it computes per-state averages but joins r.nces_id to s.state (state_code), which are incompatible keys; rev_valid doesn't even include state_code to enable a correct join, so the join will fail or produce wrong results.	✗		

Which state spent the least revenue towards schools and whats the state average score	SELECT T2.state, T3.average_scale_score FROM FINREV_FED_KEY_17 as T2 JOIN FINREV_FED_17 as T1 ON T1.state_code = T2.state_code JOIN NDECoreExcel_Math_Grade8 as T3 ON T2.state = T3.state GROUP BY T2.state ORDER BY sum(T1.t_fed_rev) LIMIT 1	WITH state_revenue AS (SELECT k.state AS state_with_least_revenue, SUM(f.t_fed_rev) AS min_state_total_revenue FROM finrev_fed_17 f JOIN finrev_fed_key_17 k ON TRUE GROUP BY k.state), least_revenue_state AS (SELECT state_with_least_revenue, min_state_total_revenue FROM state_revenue ORDER BY min_state_total_revenue ASC LIMIT 1), avg_score_2017 AS (SELECT AVG(average_scale_score)::NUMERIC AS avg_grade8_math_score_2017 FROM ndecoreexcel_math_grade8 WHERE year = 2017) SELECT l.state_with_least_revenue, l.min_state_total_revenue, a.avg_grade8_math_score_2017 FROM least_revenue_state l CROSS JOIN avg_score_2017 a;	❌	The Benchmark-Proposed Query correctly joins revenue to states via state_code, aggregates revenue by state (ORDER BY SUM(t_fed_rev)) to find the minimum, and returns that state's Grade 8 math average scale score by joining on state. (Ideally it would also filter the score table to the matching year, e.g., 2017, and aggregate or constrain the score if multiple rows per state exist.) Unreviewed query is incorrect because it cross-joins the revenue and key tables (ON TRUE), inflating sums so every state gets the same total, then arbitrarily picks one state. It also returns the national average score for 2017 rather than the identified state's score.	WITH state_totals AS (SELECT k.state, SUM(f.t_fed_rev) AS total_fed_revenue FROM finrev_fed_17 f JOIN finrev_fed_key_17 k ON k.state = f.state GROUP BY k.state), min_state AS (SELECT state, total_fed_revenue FROM state_totals ORDER BY total_fed_revenue ASC LIMIT 1) SELECT m.state, m.total_fed_revenue, s.average_scale_score AS avg_grade8_math_score_2017 FROM min_state m LEFT JOIN ndecoreexcel_math_grade8 s ON s.state = m.state;	Reviewed query correctly sums federal revenue by state, selects the state with the minimum total, and returns that state's average scale score by joining on state. It aligns with the intent: identify the least-funded state and report that state's average score.	✅		

Which state has the highest average score in math exam?	SELECT state FROM NDECoreExcel_Math_Grade8 ORDER BY average_scale_score DESC LIMIT 1	WITH per_state AS (SELECT m.state AS state, AVG(m.average_scale_score)::numeric AS avg_score FROM ndecocreexcel_math_grade8 AS m INNER JOIN finrev_fed_key_17 AS f ON f.state = m.state GROUP BY m.state) SELECT state, avg_score FROM per_state ORDER BY avg_score DESC LIMIT 1;	✅	Unreviewed query groups by state and computes the per-state average of average_scale_score, then selects the highest; this matches “Which state has the highest average score.” The join to finrev_fed_key_17 is unnecessary but does not change the per-state averages (though it could exclude states not present in that table). The Benchmark-Proposed Query does not compute a state-level average; it simply returns the state from the single row with the maximum average_scale_score, which can be just a particular year/subgroup and not the state’s overall average.	WITH per_state AS (SELECT m.state AS state, AVG(m.average_scale_score)::numeric AS avg_score FROM ndecocreexcel_math_grade8 AS m INNER JOIN finrev_fed_key_17 AS f ON f.state = m.state GROUP BY m.state) SELECT state, avg_score FROM per_state ORDER BY avg_score DESC LIMIT 1;	Reviewed query correctly selects the state with the maximum average_scale_score directly from the math Grade 8 table, which aligns with “Which state has the highest average score in math exam?” assuming the table stores the state-level average score.	❌	

What is the average match score of CA?	SELECT average_scale_score FROM NDECoreExcel_Math_Grade8 WHERE state = "California"	SELECT NULL::numeric AS average_match_score_CA WHERE FALSE;	❌	The Benchmark-Proposed Query retrieves the average math score (average_scale_score) for California from the Grade 8 math table, matching the intent of “average match (math) score of CA.” Unreviewed query deliberately returns no rows (WHERE FALSE) and does not compute or fetch any score, so it cannot answer the question.	SELECT AVG(average_scale_score) AS average_match_score_CA FROM ndecoreexcel_math_grade8 WHERE state = 'California';	Reviewed query correctly computes a single average value using AVG(average_scale_score) filtered to California, matching the question. Unreviewed query deliberately returns no rows (NULL WHERE FALSE), so it doesn't answer the question.	✅	
How many unique school districts in Indiana?	SELECT count(DISTINCT school_district) FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 ON T1.state_code = T2.state_code WHERE T2.state = "Indiana"	WITH indiana_state AS (SELECT k.state_code FROM finrev_fed_key_17 k WHERE k.state = 'Indiana'), indiana_districts AS (SELECT f.nces_id FROM finrev_fed_17 f INNER JOIN indiana_states s ON f.state_code = s.state_code WHERE f.nces_id IS NOT NULL) SELECT COUNT(DISTINCT nces_id) AS unique_districts_in_indiana FROM indiana_districts;	✅	Unreviewed query is correct because it counts DISTINCT nces_id (the unique NCES district identifier) for Indiana, ensuring each district entity is counted once regardless of naming or duplicate rows. The Benchmark-Proposed Query is incorrect because it counts DISTINCT school_district (a name field), which is not guaranteed to be unique or consistently formatted within a state and can over/under-count. It also uses a string literal with double quotes, which is not portable across SQL dialects.	WITH indiana_state AS (SELECT k.state_code FROM finrev_fed_key_17 k WHERE k.state = 'Indiana'), indiana_districts AS (SELECT f.nces_id FROM finrev_fed_17 f INNER JOIN indiana_state s ON f.state_code = s.state_code WHERE f.nces_id IS NOT NULL) SELECT COUNT(DISTINCT nces_id) AS unique_districts_in_indiana FROM indiana_districts;	Reviewed query correctly filters Indiana via state_code, and counts distinct nces_id (the proper unique district identifier), ensuring an accurate count of unique districts.	✅	

What is the state code of VA?	SELECT state_code FROM FINREV_FED_KEY_17 WHERE state = "Virginia"	SELECT DISTINCT state_code AS state_code_of_VA FROM finrev_fed_17 WHERE FALSE;	❌	The Benchmark-Proposed Query correctly selects the state_code from the key table by filtering for the state name "Virginia," which returns "VA," matching the question's intent. Unreviewed query uses WHERE FALSE, returning no rows, and doesn't filter for Virginia; thus it cannot answer the question.	SELECT DISTINCT state_code FROM finrev_fed_key_17 WHERE state = 'Virginia';	Reviewed query correctly filters by state = 'Virginia' using proper string quoting and returns the distinct state_code, yielding the state code for VA.	✅	
Which state get the highest revenue?	SELECT T2.state FROM FINREV_FED_KEY_17 as T2 JOIN FINREV_FED_17 as T1 ON T1.state_code = T2.state_code GROUP BY T2.state ORDER BY sum(t_fed_rev) DESC LIMIT 1	SELECT k.state AS state, f.t_fed_rev AS revenue FROM finrev_fed_17 AS f INNER JOIN finrev_fed_key_17 AS k ON f.state_code = k.state_code ORDER BY f.t_fed_rev DESC LIMIT 1;	❌	The Benchmark-Proposed Query correctly aggregates revenue by state (SUM(t_fed_rev)), orders by the total, and returns the top state, aligning with the intent to find the state with the highest overall revenue. Unreviewed query does not aggregate by state; it selects the single row with the highest t_fed_rev. If a state has multiple rows whose totals exceed another state's single largest row, Unreviewed query would return the wrong state.	SELECT k.state AS state, f.t_fed_rev AS revenue FROM finrev_fed_17 AS f INNER JOIN finrev_fed_key_17 AS k ON f.state = k.state ORDER BY f.t_fed_rev DESC LIMIT 1;	Reviewed query is incorrect because it joins on state name (f.state = k.state) rather than the key (state_code), which can fail or misjoin given the likely schema.	❌	

How much funding does Colorado schools receive from Title 1 and the Child Nutrition program respectively?	SELECT sum(T1.c14), sum(T1.c25) FROM FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2 ON T1.state_code = T2.state_code WHERE T2.state = "Colorado"	SELECT SUM(c14) AS title_1_funding, SUM(c25) AS child_nutrition_funding FROM finrev_fed_17 WHERE state_code = 8;	✅	Benchmark and unreviewed queries sum Title I (c14) and Child Nutrition (c25) for Colorado. Unreviewed query filters by Colorado's FIPS state_code = 8; The Benchmark-Proposed Query filters by state name via a join to the key table. Assuming FINREV_FED_KEY_17 maps state_code to the state name uniquely, both return the same totals.	SELECT SUM(c14) AS title_1_funding, SUM(c25) AS child_nutrition_funding FROM finrev_fed_17 WHERE state_code = 8;	System-generateds sum c14 (Title I) and c25 (Child Nutrition) for Colorado via state_code = 8, which corresponds to Colorado, yielding the required totals.	✅		
How much federal funding does Pecatonia Area School district get?	SELECT t_fed_rev FROM FINREV_FED_17 WHERE school_district LIKE "%Pecatonia Area%"	SELECT f.school_district AS district_name, f.t_fed_rev AS federal_funding_amount FROM finrev_fed_17 f WHERE lower(regexp_replace(f.school_district, '\s+', ' ', 'g')) IN (lower('Pecatonia Area School district'), lower('Pecatonica Area School District'));	✅	Unreviewed query correctly targets the total federal revenue (t_fed_rev) for the district and robustly matches the district name by normalizing case/whitespace and accounting for the likely misspelling ("Pecatonia" vs. correct "Pecatonica"). The Benchmark-Proposed Query is incorrect because it searches for the misspelled substring "Pecatonia," likely returning no rows if the correct name is "Pecatonica." It also uses double quotes around the LIKE pattern, which is invalid in many SQL dialects (e.g., PostgreSQL).	SELECT f.school_district AS district_name, f.t_fed_rev AS federal_funding_amount FROM finrev_fed_17 f WHERE lower(regexp_replace(f.school_district, '\s+', ' ', 'g')) IN (lower('Pecatonia Area School District'), lower('Pecatonica Area School District'));	Reviewer didn't make any changes. Unreviewed query and Reviewed query correctly match the district by normalizing case/whitespace and accounting for both the misspelled "Pecatonia" and the correct "Pecatonica," and they return the federal funding amount.	✅		

What are the student scores on the NAEP math exam for 8th graders in North Carolina and South Carolina?	SELECT average_scale_score FROM NDECoreExcel_Math_Grade8 WHERE state = "North Carolina" UNION SELECT average_scale_score FROM NDECoreExcel_Math_Grade8 WHERE state = "South Carolina"	SELECT year AS year, average_scale_score AS average_scale_score FROM ndecoreexcel_math_grade8 ORDER BY year;	❌	The Benchmark-Proposed Query is correct because it filters the Grade 8 NAEP math data to the two specified states (North Carolina and South Carolina) and returns their average_scale_score values (the student scores). Unreviewed query is incorrect because it does not filter by state; it returns scores for all states and omits the state identifier, so it doesn't answer the question about North Carolina and South Carolina specifically.	SELECT year, state, average_scale_score FROM ndec WHERE state IN ('North Carolina', 'South Carolina') ORDER BY year, state;	Reviewed query is correct because it filters to North Carolina and South Carolina and returns the relevant scores (average_scale_score) along with state and year, which properly answers the question.	✅	
What are the averge score of students in NC and in NY?	SELECT average_scale_score FROM NDECoreExcel_Math_Grade8 WHERE state = "North Carolina" UNION SELECT average_scale_score FROM NDECoreExcel_Math_Grade8 WHERE state = "New York"	SELECT NULL::text AS state_name, NULL::numeric AS average_scale_score WHERE FALSE;	❌	The Benchmark-Proposed Query filters the table to North Carolina and New York and returns their average_scale_score values, which answers the request for the average scores in NC and NY (though it would be clearer to include the state label and use single quotes; UNION could also drop a duplicate value if both scores are equal). Unreviewed query intentionally returns no rows (WHERE FALSE) and thus does not answer the question.	SELECT k.state_code AS state_name, AVG(m.average_scale_score) AS average_scale_score FROM ndecoreexcel_math_grade8 m JOIN finrev_fed_key_17 k ON k.state_code = k.state_code WHERE k.state_code IN (36) GROUP BY k.state_code ORDER BY k.state_code;	Reviewed query is incorrect because it only includes New York (36) and omits North Carolina (37).	❌	

How many schools get higher revenue via Child Nutrition A than revenue via child left school behind act?	SELECT count(*) FROM FINREV_FED_17 WHERE c25 > c14	SELECT COUNT(*) AS count_schools FROM finrev_fed_17 WHERE c25 IS NOT NULL AND c14 IS NOT NULL AND c25 > c14;	✅	Benchmark and unreviewed queries count rows where Child Nutrition Act revenue (c25) exceeds No Child Left Behind revenue (c14), matching the question. The condition c25 > c14 already excludes NULLs, so Unreviewed query's IS NOT NULL checks are redundant.	SELECT COUNT(*) AS count_schools FROM finrev_fed_17 WHERE c25 IS NOT NULL AND c25 > c14;	All three correctly count rows where Child Nutrition A revenue (c25) is greater than Child Left Behind Act revenue (c14). In SQL, comparisons with NULL evaluate to UNKNOWN and are filtered out by the WHERE clause, so explicit IS NOT NULL checks are redundant.	✅	

GreaterManchesterCrime Database

Question	Benchmark-Proposed Query	Pre-Review Query	Pre-Review Query Evaluation	Pre-Review Analysis	Post-Review Query	Post-Review Query Analysis	Post-Review Evaluation	Annotator Comment
Which streets in Manchester have the highest rates of crime?	SELECT Location FROM GreaterManchesterCrime WHERE Location like "%Street%" GROUP BY Location ORDER BY count(*) DESC LIMIT 1	WITH filtered AS (SELECT location, crimets::date AS crime_date FROM greatermanchestercrime WHERE lsoa LIKE 'Manchester %'), period AS (SELECT MIN(crime_date) AS start_date, MAX(crime_date) AS end_date, GREATEST(1, DATE_PART('year', AGE(MAX(crime_date) + INTERVAL '1 day', MIN(crime_date))) * 12 + DATE_PART('month', AGE(MAX(crime_date) + INTERVAL '1 day', MIN(crime_date))))::int AS months_span FROM filtered) SELECT f.location AS street_name, COUNT(*) AS crime_count, ROUND(COUNT(*)::numeric / p.months_span, 2) AS crimes_per_month FROM filtered f CROSS JOIN period p GROUP BY f.location, p.months_span ORDER BY crimes_per_month DESC, crime_count DESC, street_name LIMIT 10;	✅	Correct: It filters to Manchester (via LSOA), aggregates by location, and computes a crime rate (crimes per month) over the dataset period, then returns the top streets by rate.	WITH filtered AS (SELECT location, crimets::date AS crime_date FROM greatermanchestercrime WHERE lsoa LIKE 'Manchester %'), period AS (SELECT MIN(crime_date) AS start_date, MAX(crime_date) AS end_date, GREATEST(1, DATE_PART('year', AGE(MAX(crime_date) + INTERVAL '1 day', MIN(crime_date))) * 12 + DATE_PART('month', AGE(MAX(crime_date) + INTERVAL '1 day', MIN(crime_date))))::int AS months_span FROM filtered) SELECT f.location AS street_name, COUNT(*) AS crime_count, ROUND(COUNT(*)::numeric / p.months_span, 2) AS crimes_per_month FROM filtered f CROSS JOIN period p GROUP BY f.location, p.months_span ORDER BY crimes_per_month DESC, crime_count DESC, street_name LIMIT 10;	Reviewed query don't make any changes. So it is like Unreviewed query and is correct.	✅	

When was the last instance of a violent or sexual offense in Manchester?	SELECT CrimeID FROM GreaterManchesterCrime WHERE Type = "Violence and sexual offences" ORDER BY CrimeTS DESC LIMIT 1	WITH filtered AS (SELECT crimets AS last_crime_timestamp, type AS offense_type, location FROM greatermanchestercrime WHERE crimets IS NOT NULL AND location ILIKE '%Manchester%' AND type IN ('Violence and sexual offences', 'Violent crime', 'Sexual offence', 'Sexual offences', 'Violence', 'Violence and sexual offenses')) SELECT last_crime_timestamp, offense_type, location FROM filtered ORDER BY last_crime_timestamp DESC LIMIT 1;	✗	Unreviewed query returns the most recent timestamp (crimets) for records classified as violent or sexual offences, restricts to Manchester via the location filter, and orders by timestamp descending to get the latest instance.	SELECT MAX(crimets) AS last_crime_timestamp FROM greatermanchestercrime WHERE crimets IS NOT NULL AND type = 'Violence and sexual offences';	Reviewed query filters only for type = 'Violence and sexual offences', missing other relevant violent or sexual offence categories (e.g., ‘Violent crime’, ‘Sexual offence’).	✗	

Where was the last instance of a violent or sexual offense in Manchester?	SELECT Location FROM GreaterManchesterCrime WHERE Type = "Violence and sexual offences" ORDER BY CrimeTS DESC LIMIT 1	WITH violent_or_sexual AS (SELECT crimets, type, location FROM greatermanchestercrime WHERE type IN ('Violence and sexual offences', 'Violent crime', 'Sexual offences', 'Violence and sexual offense')) SELECT location AS last_offense_location, crimets AS last_offense_time, type AS offense_type FROM violent_or_sexual ORDER BY crimets DESC LIMIT 1;	✅	Unreviewed query correctly captures the intent (“violent or sexual”) by including multiple relevant category variants (e.g., 'Violence and sexual offences', 'Violent crime', 'Sexual offences') and then selects the most recent record to return the location. It uses proper string literals and will return the latest across all applicable categories. The Benchmark-Proposed Query is too narrow: it only considers records labeled exactly 'Violence and sexual offences', potentially missing separately labeled 'Violent crime' or 'Sexual offences'.	WITH violent_or_sexual AS (SELECT crimets, type, location FROM greatermanchestercrime WHERE type IN ('Violence and sexual offences', 'Violent crime', 'Sexual offences', 'Violence and sexual offense')) SELECT location AS last_offense_location, crimets AS last_offense_time, type AS offense_type FROM violent_or_sexual ORDER BY crimets DESC LIMIT 1;	Reviewer didn't make any changes. Unreviewed query and Reviewed query correctly capture the latest record across all relevant categories (violence and/or sexual), order by timestamp descending, and return the location of the most recent incident. .	✅	It is a good example of how the system considers all possibalities in the database that lead to a more robust output

Which neighborhood has the highest crime rate?	SELECT LSOA FROM GreaterManchesterCrime GROUP BY LSOA ORDER BY count(*) DESC LIMIT 1	WITH crimes_by_lsoa AS (SELECT lsoa AS neighborhood, COUNT(*) AS crime_count FROM greatermanchestercrime WHERE lsoa IS NOT NULL AND lsoa <> '' GROUP BY lsoa), ranked AS (SELECT neighborhood, crime_count, RANK() OVER (ORDER BY crime_count DESC) AS rank_order FROM crimes_by_lsoa) SELECT neighborhood, crime_count, rank_order FROM ranked WHERE rank_order = 1 ORDER BY neighborhood;	✅	Unreviewed query correctly aggregates crimes by neighborhood (LSOA), excludes null/empty values, ranks by count, and returns all top neighborhoods in case of ties, robustly answering “highest” based on crime counts. The Benchmark-Proposed Query can include NULL/blank LSOA as the “top” group and arbitrarily returns a single row with LIMIT 1, failing to handle ties and potentially returning a non-neighborhood value.	WITH agg AS (SELECT lsoa AS neighborhood, COUNT(*) AS crime_count, COUNT(DISTINCT CAST(crimets AS date)) AS active_days FROM greatermanchestercrime WHERE lsoa IS NOT NULL AND lsoa <> '' GROUP BY lsoa), rates AS (SELECT neighborhood, CASE WHEN active_days > 0 THEN 1.0 * crime_count / active_days ELSE NULL END AS daily_crime_rate FROM agg) SELECT neighborhood FROM rates ORDER BY daily_crime_rate DESC, neighborhood LIMIT 1;	Reviewed query computes a crime rate (crimes per active day) per neighborhood, filters out null/empty neighborhoods, and returns the neighborhood with the highest rate. This aligns with the notion of a “rate” rather than a raw count.	✅	It shows how the system treat NULL/blank values

What is the crime rate for Eden Street area?	SELECT count(*) FROM GreaterManchesterCrime WHERE Location like "%Eden Street%"	WITH filtered AS (SELECT location AS area_location, lsoa AS area_lsoa, DATE_TRUNC('month', crimets)::date AS month_start, crimeid FROM greatermanchestercrime WHERE location ILIKE '%Eden Street%' AND crimets IS NOT NULL), period AS (SELECT MIN(month_start) AS start_month, MAX(month_start) AS end_month, (DATE_PART('year', MAX(month_start)) - DATE_PART('year', MIN(month_start))) * 12 + (DATE_PART('month', MAX(month_start)) - DATE_PART('month', MIN(month_start)) + 1) AS months_in_period FROM filtered), counts AS (SELECT area_location, area_lsoa, COUNT(DISTINCT crimeid) AS crime_count FROM filtered GROUP BY area_location, area_lsoa) SELECT c.area_location, c.area_lsoa, c.crime_count, ROUND(c.crime_count / NULLIF(p.months_in_period, 0), 2) AS crime_count_per_month FROM counts c CROSS JOIN period p ORDER BY c.crime_count DESC, c.area_location, c.area_lsoa;	✅	Unreviewed query filters crimes for “Eden Street,” counts distinct incidents, and normalizes by the number of months in the observed period to produce a crime rate (crimes per month). This matches the intent of “crime rate” and uses appropriate case-insensitive matching. The Benchmark-Proposed Query only returns a raw count (not a rate), ignores time entirely.	WITH filtered AS (SELECT location AS area_location, lsoa AS area_lsoa, DATE_TRUNC('month', crimets)::date AS month_start, crimeid FROM greatermanchestercrime WHERE location ILIKE '%Eden Street%' AND crimets IS NOT NULL), period AS (SELECT MIN(month_start) AS start_month, MAX(month_start) AS end_month, (DATE_PART('year', MAX(month_start)) - DATE_PART('year', MIN(month_start))) * 12 + (DATE_PART('month', MAX(month_start)) - DATE_PART('month', MIN(month_start)) + 1) AS months_in_period FROM filtered), counts AS (SELECT area_location, area_lsoa, COUNT(DISTINCT crimeid) AS crime_count FROM filtered GROUP BY area_location, area_lsoa) SELECT c.area_location, c.area_lsoa, c.crime_count, ROUND(c.crime_count / NULLIF(p.months_in_period, 0), 2) AS crime_count_per_month FROM counts c CROSS JOIN period p ORDER BY c.crime_count DESC, c.area_location, c.area_lsoa;	Unreviewed query and Reviewed query correctly compute a crime rate by filtering Eden Street, counting distinct crimes, deriving the covered month range, and returning an average crimes-per-month metric.	✅	It is a example of the benchamrk incorrect query

What is the most common type of crime?	SELECT Type FROM GreaterManchesterCrime GROUP BY Type ORDER BY count(*) DESC LIMIT 1	WITH crime_counts AS (
		  SELECT
		    type AS crime_type,
		    COUNT(*) AS crime_count
		  FROM greatermanchestercrime
		  GROUP BY type
		)
		SELECT crime_type, crime_count
		FROM crime_counts
		ORDER BY crime_count DESC, crime_type
		FETCH FIRST 1 ROW WITH TIES;
			✅	Benchmark and unreviewed queries aggregate by crime type and order by the count descending to return the most common type. Unreviewed query also returns the count; its WITH TIES is neutralized by the secondary sort on crime_type, so it effectively returns a single top type, which still answers the question.	WITH crime_counts AS (
					  SELECT
					    type AS crime_type,
					    COUNT(*) AS crime_count
					  FROM greatermanchestercrime
					  GROUP BY type
					)
					SELECT crime_type, crime_count
					FROM crime_counts
					ORDER BY crime_count DESC, crime_type
					FETCH FIRST 1 ROWS WITH TIES;
						Reviewer didn't make any changes.
						All three aggregate by type and select the type with the highest count.
							✅	

Which neighborhood/area has the highest burglary rate?	SELECT Location FROM GreaterManchesterCrime WHERE Type = "Burglary" GROUP BY Location ORDER BY count(*) DESC LIMIT 1	WITH burglary_counts AS (SELECT lsoa AS area, COUNT(*) AS burglary_count FROM greatermanchestercrime WHERE type ILIKE 'burglary' GROUP BY lsoa) SELECT area, burglary_count FROM burglary_counts ORDER BY burglary_count DESC, area LIMIT 1;	❌	Unreviewed query aggregates burglaries by LSOA (a neighborhood/area unit), filters case-insensitively for burglary, and returns the area with the highest count, matching the intent at the area level. Note: it reports counts, not a true rate (which would require a denominator like population), but it's the appropriate area-based aggregation.	SELECT lsoa AS area, SUM(CASE WHEN type ILIKE 'Burglary' THEN 1 ELSE 0 END)::float / COUNT(*) AS burglary_rate FROM greatermanchestercrime GROUP BY lsoa ORDER BY burglary_rate DESC, area LIMIT 1;	Reviewed query correctly computes a burglary rate per area (burglaries divided by total crimes in the LSOA) and selects the highest, matching the request for "highest burglary rate."	✅		

Which area do most of the crimes happen?	SELECT Location FROM GreaterManchesterCrime GROUP BY Location ORDER BY count(*) DESC LIMIT 1	WITH crimes_by_area AS (
		  SELECT
		    lsoa AS area,
		    COUNT(*) AS crime_count
		  FROM greatermanchestercrime
		  GROUP BY lsoa
		),
		max_count AS (
		  SELECT MAX(crime_count) AS max_crime_count
		  FROM crimes_by_area
		)
		SELECT
		  c.area,
		  c.crime_count
		FROM crimes_by_area c
		JOIN max_count m ON c.crime_count = m.max_crime_count
		ORDER BY c.area;
			✅	Question is vague since in the database there are two columns that refer to "area": locain and lsoa. Unreviewed query aggregates by the area dimension (LSOA), finds the maximum crime count, and returns the area(s) with that maximum, correctly answering which area has the most crimes and handling ties. Benchmark uses locaiton column.	WITH crimes_by_area AS (
					  SELECT
					    lsoa AS area,
					    COUNT(*) AS crime_count
					  FROM greatermanchestercrime
					  GROUP BY lsoa
					),
					max_count AS (
					  SELECT MAX(crime_count) AS max_crime_count
					  FROM crimes_by_area
					)
					SELECT
					  c.area,
					  c.crime_count
					FROM crimes_by_area c
					JOIN max_count m ON c.crime_count = m.max_crime_count
					ORDER BY c.area;
						System generateds aggregate by LSOA (area), find the maximum crime count, and return the area(s) with the most crimes, handling ties.	✅	

Which Lower Layer Output Area do most of the crimes happen?	SELECT LSOA FROM GreaterManchesterCrime GROUP BY LSOA ORDER BY count(*) DESC LIMIT 1	WITH crimes_by_lsoa AS (SELECT lsoa, COUNT(crimeid) AS crime_count FROM greatermanchestercrime GROUP BY lsoa) SELECT lsoa, crime_count FROM crimes_by_lsoa ORDER BY crime_count DESC, lsoa LIMIT 1;	✅	The Benchmark-Proposed Query correctly groups by LSOA and orders by count(*) to count all crime records per area, returning the LSOA with the highest number of crimes.	WITH crimes_by_lsoa AS (SELECT lsoa, COUNT(crimeid) AS crime_count FROM greatermanchestercrime GROUP BY lsoa) SELECT lsoa, crime_count FROM crimes_by_lsoa ORDER BY crime_count DESC, lsoa LIMIT 1;	All three queries correctly aggregate crimes by LSOA and return the LSOA with the highest count. Unreviewed query returns the LSOA directly using COUNT(*). None of the queries is incorrect for the stated question.	✅	

What time do most of the crimes happen?	SELECT CrimeTS FROM GreaterManchesterCrime GROUP BY CrimeTS ORDER BY count(*) DESC LIMIT 1	WITH hourly_counts AS (
		  SELECT
		    EXTRACT(HOUR FROM crimets)::int AS crime_hour,
		    COUNT(*) AS crime_count
		  FROM greatermanchestercrime
		  WHERE crimets IS NOT NULL
		  GROUP BY EXTRACT(HOUR FROM crimets)
		)
		SELECT crime_hour, crime_count
		FROM hourly_counts
		ORDER BY crime_count DESC, crime_hour ASC
		LIMIT 1;
			✅	Unreviewed query correctly aggregates crimes by hour of day (EXTRACT(HOUR)) and selects the hour with the highest count, which matches the intent of “what time” as time-of-day. The Benchmark-Proposed Query groups by the exact timestamp, returning the single most frequently repeated timestamp (date+time), not the time-of-day. This does not answer the question and is likely dominated by data-entry duplicates rather than true temporal patterns.	WITH hourly_counts AS (
					  SELECT
					    EXTRACT(HOUR FROM crimets)::int AS crime_hour,
					    COUNT(*) AS crime_count
					  FROM greatermanchestercrime
					  WHERE crimets IS NOT NULL
					  GROUP BY EXTRACT(HOUR FROM crimets)
					)
					SELECT crime_hour, crime_count
					FROM hourly_counts
					ORDER BY crime_count DESC, crime_hour ASC
					LIMIT 1;
						Unreviewed query and Reviewed query correctly aggregate crimes by hour of day and return the hour with the highest count (time when most crimes happen). They also handle ties deterministically.	✅	

What are the most likely outcome of the police investigation if the crime happen on "street"?	SELECT Outcome FROM GreaterManchesterCrime WHERE Location LIKE "%Street%" GROUP BY Outcome ORDER BY count(*) DESC LIMIT 1	WITH street_crimes AS (SELECT outcome FROM greatermanchestercrime WHERE location ILIKE '%street%' AND outcome IS NOT NULL AND outcome <> ''), counts AS (SELECT outcome AS likely_outcome, COUNT(*) AS outcome_count FROM street_crimes GROUP BY outcome), ranked AS (SELECT likely_outcome, outcome_count, RANK() OVER (ORDER BY outcome_count DESC) AS rnk FROM counts) SELECT likely_outcome, outcome_count FROM ranked WHERE rnk = 1;	✅	Unreviewed query correctly filters street-related records case-insensitively (ILIKE '%street%'), excludes null/empty outcomes, counts frequencies, and returns all top outcomes in case of ties, matching the intent of “most likely outcome.”	WITH street_crimes AS (SELECT outcome FROM greatermanchestercrime WHERE location ILIKE '%street%' AND outcome IS NOT NULL AND outcome <> ''), counts AS (SELECT outcome AS likely_outcome, COUNT(*) AS outcome_count FROM street_crimes GROUP BY outcome), ranked AS (SELECT likely_outcome, outcome_count, RANK() OVER (ORDER BY outcome_count DESC) AS rnk FROM counts) SELECT likely_outcome, outcome_count FROM ranked WHERE rnk = 1;	Correct: System-generateds filter locations case-insensitively (ILIKE '%street%'), exclude NULL/empty outcomes, aggregate counts, and use RANK to return all top outcomes in case of ties, matching "most likely outcome" intent.	✅	

Which type of crime happen the most in Salford?	SELECT Type FROM GreaterManchesterCrime WHERE LSOA LIKE "%Salford%" GROUP BY Type ORDER BY count(*) DESC LIMIT 1	WITH salford_crimes AS (SELECT type AS crime_type FROM greatermanchestercrime WHERE LOWER(location) LIKE '%salford%'), type_counts AS (SELECT crime_type, COUNT(*) AS crime_count FROM salford_crimes GROUP BY crime_type) SELECT crime_type, crime_count FROM type_counts ORDER BY crime_count DESC LIMIT 1;	✗	The Benchmark-Proposed Query correctly filters Salford records via LSOA names containing "Salford", then groups by Type and orders by count to return the most frequent crime type. Unreviewed query filters on the free-text location field, which typically contains street/place descriptions (e.g., "On or near ...") and often omits the borough name, leading to many Salford crimes being excluded and possible false matches.	WITH salford_crimes AS (SELECT type AS crime_type FROM greatermanchestercrime WHERE LOWER(location) LIKE '%salford%'), type_counts AS (SELECT crime_type, COUNT(*) AS crime_count FROM salford_crimes GROUP BY crime_type) SELECT crime_type, crime_count FROM type_counts ORDER BY crime_count DESC LIMIT 1;	Unreviewed query and Reviewed query are incorrect because they filter on the free-text location field containing “salford,” which is not a reliable indicator of the Salford local authority area. Location typically stores street-level descriptions (e.g., “On or near ...”) and may omit “Salford,” causing undercounts or false matches.	✗	

How many crimes are under investigation?	SELECT count(*) FROM GreaterManchesterCrime WHERE Outcome = "Under investigation"	SELECT COUNT(DISTINCT crimeid) AS crimes_under_investigation FROM greatermanchestercrime WHERE outcome = 'Under investigation';	✅	Unreviewed query correctly counts unique crimes under investigation by using COUNT(DISTINCT crimeid) and proper string quoting, aligning with the intent to count crimes, not rows.	SELECT COUNT(DISTINCT crimeid) AS crimes_under_investigation FROM greatermanchestercrime WHERE outcome = 'Under investigation';	Both system-generateds correctly count distinct crimes under investigation using COUNT(DISTINCT crimeid) and proper string literals. Benchamrk query may overcount if multiple rows exist per crimeid.	✅	
How many crimes has been conducted?	SELECT count(*) FROM GreaterManchesterCrime	SELECT COUNT(crimeid) AS total_crimes FROM greatermanchestercrime WHERE crimeid IS NOT NULL;	❌	The Benchmark-Proposed Query (COUNT(*)) counts all rows in the table, which matches the intent to get the total number of crimes recorded, regardless of nulls in any column. Unreviewed query counts only non-null crimeid values and will undercount if any records have a null crimeid. It’s only equivalent if crimeid is guaranteed NOT NULL.	SELECT COUNT(*) AS total_crimes FROM greatermanchestercrime WHERE outcome = 'Investigation complete';	Reviewer query is incorrect because it counts only crimes with outcome = 'Investigation complete' rather than all crimes.	❌	

What is the top 3 area of crime conducted?	SELECT Location FROM GreaterManchesterCrime GROUP BY Location ORDER BY count(*) DESC LIMIT 3	WITH crime_counts AS (SELECT lsoa AS area, COUNT(*) AS crime_count FROM greatermanchestercrime GROUP BY lsoa), ranked AS (SELECT area, crime_count, ROW_NUMBER() OVER (ORDER BY crime_count DESC, area ASC) AS rank_order FROM crime_counts) SELECT area, crime_count, rank_order FROM ranked WHERE rank_order <= 3 ORDER BY rank_order;	✅	Question is vague since there are two columns refering "area". Unreviewed query aggregates by LSOA (an administrative area), counts crimes, ranks by count, and returns the top 3 areas, matching the intent of “top 3 area of crime conducted.”	WITH crime_counts AS (SELECT lsoa AS area, COUNT(*) AS crime_count FROM greatermanchestercrime GROUP BY lsoa), ranked AS (SELECT area, crime_count, ROW_NUMBER() OVER (ORDER BY crime_count DESC, area ASC) AS rank_order FROM crime_counts) SELECT area, crime_count, rank_order FROM ranked WHERE rank_order <= 3 ORDER BY rank_order;	Unreviewed query and Reviewed query aggregate by LSOA (area), count crimes, rank by count, and return the top 3 areas.	✅	

Which type of crime has the highest rate of "Investigation complete"?	SELECT Type FROM GreaterManchesterCrime WHERE Outcome LIKE "%Investigation complete%" GROUP BY Type ORDER BY count(*) DESC LIMIT 1	WITH investigation_complete AS (SELECT type AS crime_type FROM greatermanchestercrime WHERE outcome LIKE 'Investigation complete%') SELECT crime_type, COUNT(*) AS investigation_complete_count FROM investigation_complete GROUP BY crime_type ORDER BY investigation_complete_count DESC LIMIT 1;	✗	Benchmark query and Unreviewed query are incorrect because they only count the number of “Investigation complete%” outcomes per type and return the maximum count, which identifies the most frequent, not the highest rate relative to all crimes of that type.	SELECT type FROM (SELECT type, SUM(CASE WHEN outcome LIKE 'Investigation complete%' THEN 1 ELSE 0 END) AS investigation_complete_count FROM greatermanchestercrime GROUP BY type) t ORDER BY investigation_complete_count DESC LIMIT 1;	Reviewed query is incorrect because it orders by the raw count of “Investigation complete” outcomes instead of the rate.	✗	

How many crimes were classified as "Drugs"?	SELECT count(*) FROM GreaterManchesterCrime WHERE Type LIKE "%Drug%"	SELECT COUNT(DISTINCT crimeid) AS drugs_crime_count FROM greatermanchestercrime WHERE type = 'Drugs';	✅	Unreviewed query matches the classification exactly ('Drugs') and counts distinct crimes via crimeid, preventing double-counting. The Benchmark-Proposed Query uses LIKE "%Drug%", which can include non-exact categories (e.g., 'Drug dealing'), may overcount due to duplicates (no DISTINCT)	SELECT COUNT(DISTINCT crimeid) AS drugs_crime_count FROM greatermanchestercrime WHERE type = 'Drugs';	System-generateds exactly count crimes where type = 'Drugs' and use DISTINCT crimeid to avoid duplicates, matching the question's classification.	✅	
How many crimes are still "Under investigation" to date?	SELECT count(*) FROM GreaterManchesterCrime WHERE Outcome LIke "%Under investigation%"	SELECT COUNT(*) AS under_investigation_count FROM greatermanchestercrime WHERE outcome = 'Under investigation';	✅	Unreviewed query correctly counts rows where outcome exactly equals 'Under investigation', which matches the question's intent and the standard outcome label.	SELECT COUNT(*) AS under_investigation_count FROM greatermanchestercrime WHERE outcome = 'Under investigation';	System-generateds correctly count rows with outcome exactly equal to 'Under investigation', matching the question's intent. Benchmark query uses LIKE "%Under investigation%", which can overcount by matching any outcome containing that phrase	✅	

Which location has the top amount of "Drugs" crime conducted?	SELECT Location FROM GreaterManchesterCrime WHERE Type LIke "%Drug%" GROUP BY Location ORDER BY count(*) DESC LIMIT 1	WITH drugs_by_location AS (SELECT location, COUNT(*) AS drugs_crime_count FROM greatermanchestercrime WHERE type = 'Drugs' AND location IS NOT NULL AND location <> '' GROUP BY location) SELECT location, drugs_crime_count FROM drugs_by_location ORDER BY drugs_crime_count DESC, location ASC LIMIT 1;	✅	Unreviewed query correctly filters exactly Type = 'Drugs', excludes null/empty locations, aggregates by location, orders by count descending with a deterministic tie-break, and returns the top location (with count). The Benchmark-Proposed Query is unreliable: it uses LIKE '%Drug%' (may capture non-'Drugs' types) and doesn't exclude null/blank locations, which could return an uninformative result.	WITH drugs_by_location AS (SELECT location, COUNT(*) AS drugs_crime_count FROM greatermanchestercrime WHERE type = 'Drugs' AND location IS NOT NULL AND location <> '' GROUP BY location) SELECT location, drugs_crime_count FROM drugs_by_location ORDER BY drugs_crime_count DESC, location ASC LIMIT 1;	System-generateds correctly filter Type = 'Drugs', exclude null/empty locations, group by location, and select the highest count (with a deterministic tie-break).	✅	

What's the most common type of crime?	SELECT Type FROM GreaterManchesterCrime GROUP BY Type ORDER BY count(*) DESC LIMIT 1	WITH cleaned AS (
		  SELECT type
		  FROM greatermanchestercrime
		  WHERE type IS NOT NULL AND btrim(type) <> ''
		),
		counts AS (
		  SELECT
		    type AS crime_type,
		    COUNT(*) AS crime_count
		  FROM cleaned
		  GROUP BY type
		)
		SELECT crime_type, crime_count
		FROM counts
		ORDER BY crime_count DESC, crime_type ASC
		LIMIT 1;
			✅	Unreviewed query is correct because it excludes NULL and blank types, counts occurrences by type, and returns the highest-frequency crime type (with its count), which aligns with the intent of "most common type."
				The Benchmark-Proposed Query is incorrect because it does not filter out NULL or empty strings; if such values exist, it could return NULL/blank as the "most common type," which is not a valid crime type.
					WITH cleaned AS (
					  SELECT type
					  FROM greatermanchestercrime
					  WHERE type IS NOT NULL AND btrim(type) <> ''
					),
					counts AS (
					  SELECT
					    type AS crime_type,
					    COUNT(*) AS crime_count
					  FROM cleaned
					  GROUP BY type
					)
					SELECT crime_type, crime_count
					FROM counts
					ORDER BY crime_count DESC, crime_type ASC
					LIMIT 1;
						Correct: System-generateds clean the data (exclude NULL/blank types), count by type, and return the top type (with deterministic tie-break). This directly answers "most common type of crime."	✅	

Where is the most dangerous area?	SELECT Location FROM GreaterManchesterCrime GROUP BY Location ORDER BY count(*) DESC LIMIT 1	WITH crime_counts AS (SELECT lsoa AS area, COUNT(*) AS crime_count FROM greatermanchestercrime GROUP BY lsoa), ranked AS (SELECT area, crime_count, DENSE_RANK() OVER (ORDER BY crime_count DESC) AS rank_order FROM crime_counts) SELECT area, crime_count FROM ranked WHERE rank_order = 1;	✅	Unreviewed query correctly aggregates crimes by LSOA (an administrative area), identifies the maximum via dense_rank, and returns the area(s) with the highest crime count, matching the intent of “most dangerous area” and handling ties appropriately. The Benchmark-Proposed Query groups by Location, which is a street-level descriptor (“On or near …”) rather than an area, and uses LIMIT 1 to pick an arbitrary winner in case of ties, so it does not accurately answer the question about the most dangerous area.	WITH crime_counts AS (SELECT lsoa AS area, COUNT(*) AS crime_count FROM greatermanchestercrime GROUP BY lsoa), ranked AS (SELECT area, crime_count, DENSE_RANK() OVER (ORDER BY crime_count DESC) AS rank_order FROM crime_counts) SELECT area, crime_count FROM ranked WHERE rank_order = 1;	System-generateds correctly aggregate crimes by LSOA (area) and return the area(s) with the highest count, handling ties via DENSE_RANK.	✅	

Where is the safest area?	SELECT Location FROM GreaterManchesterCrime GROUP BY Location ORDER BY count(*) LIMIT 1	WITH crimes_by_area AS (SELECT lsoa AS area, COUNT(*) AS crime_count FROM greatermanchestercrime WHERE lsoa IS NOT NULL GROUP BY lsoa), min_count AS (SELECT MIN(crime_count) AS min_crime_count FROM crimes_by_area) SELECT c.area, c.crime_count FROM crimes_by_area c JOIN min_count m ON c.crime_count = m.min_crime_count ORDER BY c.area;	✅	Question is vague since two columns in the database might refer to "area". Unreviewed query correctly defines “area” as LSOA, aggregates crimes by area, finds the minimum, and returns the area(s) with the fewest crimes, matching “safest area.”	WITH crimes_by_area AS (SELECT lsoa AS area, COUNT(*) AS crime_count FROM greatermanchestercrime WHERE lsoa IS NOT NULL GROUP BY lsoa), min_count AS (SELECT MIN(crime_count) AS min_crime_count FROM crimes_by_area) SELECT c.area, c.crime_count FROM crimes_by_area c JOIN min_count m ON c.crime_count = m.min_crime_count ORDER BY c.area;	System-generateds correctly compute the safest area by aggregating crimes by LSOA (area), excluding nulls, finding the minimum crime count, and returning all areas tied for the minimum.	✅	

Where is the place with the largest number of sexual offenses crime events?	SELECT Location FROM GreaterManchesterCrime WHERE Type = "Violence and sexual offences" GROUP BY Location ORDER BY count(*) DESC LIMIT 1	WITH sexual_offenses AS (SELECT location, COUNT(*) AS cnt FROM greatermanchestercrime WHERE type = 'Sexual offenses' GROUP BY location), max_cnt AS (SELECT MAX(cnt) AS max_cnt FROM sexual_offenses) SELECT s.location AS place, s.cnt AS offense_count FROM sexual_offenses s JOIN max_cnt m ON s.cnt = m.max_cnt ORDER BY s.cnt DESC, s.location ASC LIMIT 1;	✗	The Benchmark-Proposed Query applies the correct category used in this dataset ("Violence and sexual offences"), groups by Location, and orders by the event count to return the place with the highest number. Unreviewed query filters on type = 'Sexual offenses', which does not match the dataset's category name/spelling and will likely return no rows, making it unable to identify the correct location.	WITH sexual_offences AS (SELECT location, COUNT(*) AS cnt FROM greatermanchestercrime WHERE type = 'Sexual offences' GROUP BY location), max_cnt AS (SELECT MAX(cnt) AS max_cnt FROM sexual_offences) SELECT s.location AS place, s.cnt AS offense_count FROM sexual_offences s JOIN max_cnt m ON s.cnt = m.max_cnt ORDER BY s.cnt DESC, s.location ASC LIMIT 1;	Reviewed query is incorrect: it filters for the correct category ('Sexual offences' with UK spelling), aggregates by location, finds the maximum count, and returns the place with the largest number of such crimes.	✗	

If it is possible to increase the number of police officers, which place is with the first priority?	SELECT Location FROM GreaterManchesterCrime GROUP BY Location ORDER BY count(*) DESC LIMIT 1	WITH filtered AS (SELECT lsoa, outcome FROM greatermanchestercrime WHERE outcome IN ('Investigation complete;	❌	The Benchmark-Proposed Query correctly identifies the place with the highest number of incidents by grouping by Location, ordering by the incident count descending, and taking the top result, matching the intent to prioritize the area with the most crime. Unreviewed query is syntactically incomplete/invalid (truncated IN clause and CTE), filters on a specific outcome category, and doesn't aggregate or rank places, so it doesn't answer which place should be prioritized.	SELECT location FROM greater_manchester_crime GROUP BY location ORDER BY COUNT(*) DESC LIMIT 1;	Reviewer groups by location and order by total incidents descending, returning the place with the most crimes, the sensible first priority for more officers.	✅	

Which type of crime has the most prosecution cases?	SELECT Type FROM GreaterManchesterCrime WHERE Outcome = "Awaiting court outcome" GROUP BY Type ORDER BY count(*) DESC LIMIT 1	WITH prosecution_outcomes AS (SELECT type AS crime_type, outcome FROM greatermanchestercrime WHERE outcome IN ('Court result unavailable', 'Offender sent to court', 'Suspect charged', 'Suspect charged as part of another case', 'Defendant sent to Crown Court', 'Local resolution – offender prosecuted', 'Prosecution')), counts AS (SELECT crime_type, COUNT(*) AS prosecution_case_count FROM prosecution_outcomes GROUP BY crime_type) SELECT crime_type, prosecution_case_count FROM counts ORDER BY prosecution_case_count DESC LIMIT 1;	✗	Unreviewed query is incorrect since doesn't return any results.	SELECT type AS crime_type, COUNT(*) AS prosecution_case_count FROM greatermanchestercrime WHERE outcome ILIKE '%investigation%' GROUP BY type ORDER BY prosecution_case_count DESC LIMIT 1;	This query is incorrect because it filters on outcomes containing “investigation” instead of outcomes related to prosecution or court.	✗	

Which type of crime has the most investigation complete cases?	SELECT Type FROM GreaterManchesterCrime WHERE Outcome = "Investigation complete; no suspect identified" GROUP BY Type ORDER BY count(*) DESC LIMIT 1	SELECT type AS crime_type, COUNT(*) AS investigation_complete_cases FROM greatermanchestercrime WHERE outcome = 'Investigation complete;	❌	The Benchmark-Proposed Query filters to the standard outcome "Investigation complete; no suspect identified," groups by crime type, orders by the count in descending order, and returns the top type, directly answering which type has the most such cases. Unreviewed query is invalid and incorrect: it has an unclosed string literal, uses an incomplete outcome value, and lacks a GROUP BY (and ORDER BY/LIMIT) to identify the crime type with the maximum count.	SELECT type AS crime_type, COUNT(*) AS investigation_complete_cases FROM greatermanchestercrime WHERE outcome = 'Investigation complete; no suspect identified' GROUP BY type ORDER BY investigation_complete_cases DESC LIMIT 1;	Reviewed query correctly filters for 'Investigation complete; no suspect identified', groups by crime type, orders by the count descending, and returns the top type (and count), which answers which type has the most such cases.	✅	

What is the result in case 6B:E2:54:C6:58:D2?	SELECT Outcome FROM GreaterManchesterCrime WHERE CrimeID = "6B:E2:54:C6:58:D2"	SELECT outcome AS outcome FROM greatermanchestercrime WHERE crimeid = '6B:E2:54:C6:58:D2';	✅	Unreviewed query correctly filters CrimeID using a proper string literal and returns the Outcome column, matching the question's intent ("result" = outcome).	SELECT outcome AS outcome FROM greatermanchestercrime WHERE crimeid = '6B:E2:54:C6:58:D2';	Unreviewed query and Reviewed query correctly select the outcome for the specified crimeid using proper single-quoted string literals and consistent table/column names; they are identical.	✅	