Explanation for the list of queries

Antonio José González Castillo

Charles University / Universidad de Málaga

NDBI040 - Modern Database Systems

Irena Holubová

INTRODUCTORY NOTES

- This is a comprehensive query documentation. Here, I have focused on explaining queries that might trip you up, providing clear insights into their workings. We've also tackled tricky queries, detailing exceptions and common stumbling blocks, with references to documentation and helpful error images. This document aims to demystify complex queries, making them more accessible and easier to understand.

  Those queries that are very easy to understand will only be displayed in the document.

- CrateDB doesn't handle index creation like other databases do. Instead, it's got this cool automatic system. It figures out which columns need indexing based on how you're querying and accessing your data. Basically, you don't have to worry about creating indexes yourself. CrateDB takes care of it for you, saving you time and hassle. This means you can focus more on your queries and analyzing your data, and less on managing indexes.

  For more info on how CrateDB's automatic indexing works, you can check out the docs on full-text indices here:
  https://cratedb.com/docs/crate/reference/en/latest/general/ddl/fulltext-indices.html

LIST OF QUERIES

 # 3.1.1 (Filtering on a non-indexed column)

"SELECT * FROM business_json WHERE state = 'AZ';",

 # 3.1.2 (Filtering on a non-indexed column, range query)

"SELECT * FROM review_json WHERE stars BETWEEN 3 AND 5;",

 # 3.1.3 (Filtering on indexed column, exact match)

"SELECT * FROM business_json WHERE stars = 4;",

 # 3.1.4 (Filtering on indexed column, range query)

"SELECT * FROM business_json WHERE stars BETWEEN '4' AND '5';",

 # 3.2.1 (Use aggregation function count)

"SELECT b.state, COUNT(*) AS total_reviews FROM business_json b JOIN review_json r ON b.business_id = r.business_id GROUP BY b.state;",

Explanation:

- We do a JOIN between business_json and review_json based on the business_id.
- We group the results by the status of each business.
- We use the COUNT(*) aggregation function to count the total number of reviews for each status.

 # 3.2.2 (Use aggregation function max)

"SELECT state, MAX(stars) AS max_stars FROM business_json GROUP BY state;",

Explanation:

- This query uses the MAX aggregation function to find the maximum review score in each city.
- A join is performed between the business_json and review_json tables to get the reviews for each business.
- Then, we group by city and find the maximum review score in each city.

 # 3.3.1 (Joining / traversal where two entities are connected by non-indexed columns)

"SELECT b.name AS business_name, r.stars AS review_stars, r.text AS review_text FROM business_json b JOIN review_json r ON b.business_id = r.business_id;",

Explanation:

- This query joins the business_json and review_json tables based on the business_id column, which is likely indexed for efficient joining.
- We select the business name, review stars, and review text.
- Since the business_id column is used for joining and likely indexed, the query should perform efficiently.

 # 3.3.2 (Joining / traversal over indexed column)

"SELECT r.review_id, r.text, b.name FROM review_json r JOIN business_json b ON r.business_id = b.business_id WHERE b.state = 'PA';",

Explanation:

- We're joining the business_json and tip_json tables based on the business_id column, which is likely indexed for efficiency.
- We select the business name and tip text.
- By leveraging the indexed column for joining, the query should execute efficiently.

 # 3.3.3 (Complex join involving multiple JOINS)

"SELECT u.name AS user_name, r.stars AS review_stars, b.name AS business_name FROM user_json u JOIN review_json r ON u.user_id = r.user_id JOIN business_json b ON r.business_id = b.business_id;",

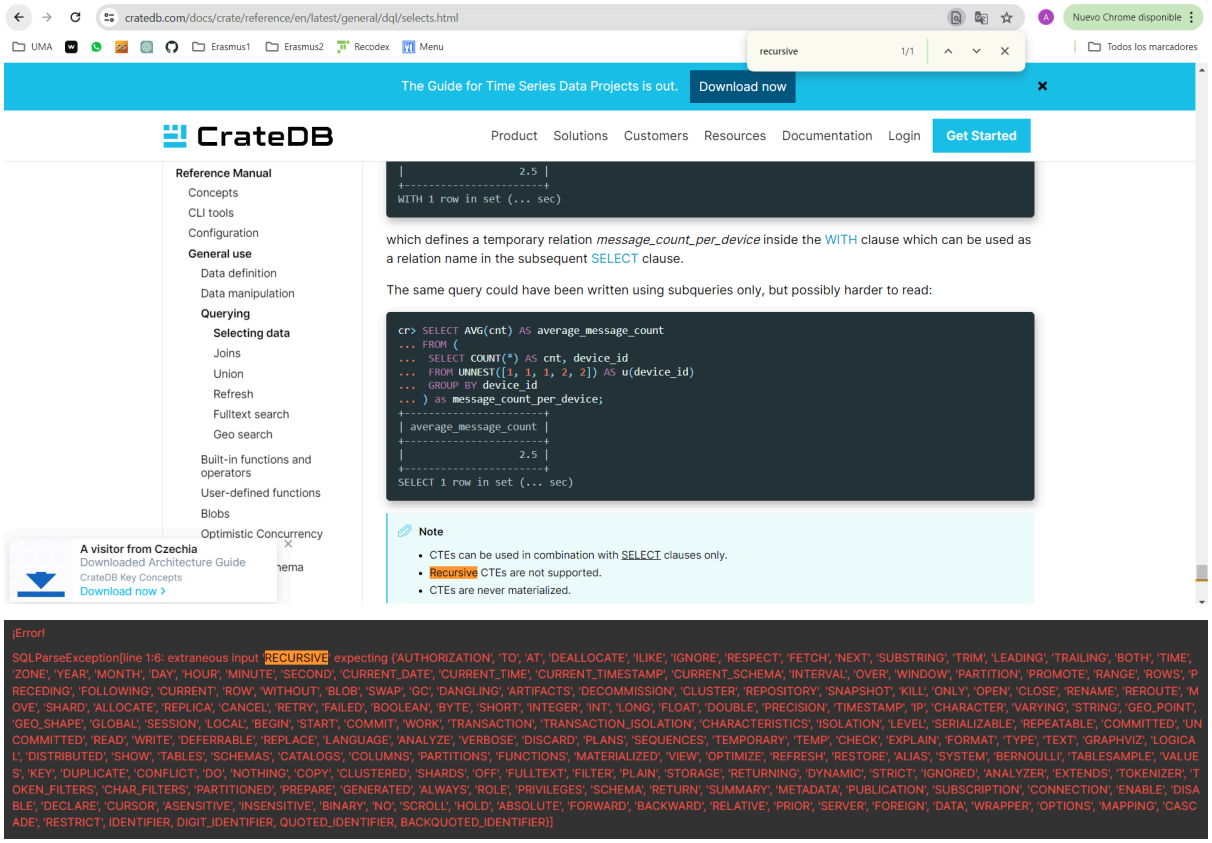
Explanation:

- This query involves joining three tables: user_json, review_json, and business_json, based on their respective IDs.
- We select the user name, review stars, and business name.
- While it involves multiple joins, if the ID columns are indexed, and assuming efficient hardware and query optimization, the query should execute reasonably quickly.

3.3.4 (Recursive query)

Not supported by crateDB

https://cratedb.com/docs/crate/reference/en/latest/general/dql/selects.html



3.3.5 (Optional traversal, SQL: LEFT OUTER JOIN)

"SELECT b.name AS business_name, c.date AS checkin_date FROM business_json b LEFT OUTER JOIN checkin_json c ON b.business_id = c.business_id;",

Explanation:

- This query uses a left outer join to include all records from business_json and matching records from checkin_json based on the business_id column.
- We select the business name and check-in date.
- By using a left outer join, we ensure that all businesses are included, even if they don't have corresponding check-in data.

 # 3.4.1 (Union)

"SELECT business_id FROM business_json UNION SELECT business_id FROM checkin_json;",

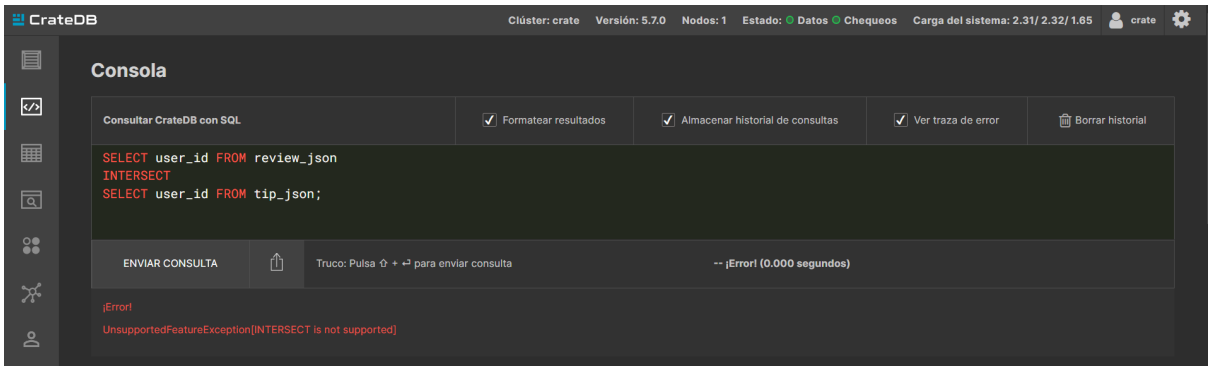
Explanation:

- This query retrieves all unique business_id values from both the business_json and checkin_json tables and combines them into a single result set.
- We use UNION to eliminate duplicates, you can use UNION instead.

 # 3.4.2 (Intersect)

"SELECT DISTINCT user_id FROM review_json WHERE EXISTS (SELECT 1 FROM tip_json WHERE review_json.user_id = tip_json.user_id);",

Intersect is not supported by CrateDB for this type of query. I used other ways to fulfil the task.



Explanation:

- This query uses a correlated subquery within a WHERE EXISTS clause to find common business_id values between the business_json and checkin_json tables.
- For each row in the business_json table, the subquery checks if there exists a corresponding business_id in the checkin_json table.
- If such a match is found, the business_id is included in the result set.
- This approach achieves the intersection of business_id values between the two tables without directly using the INTERSECT operator, making it compatible with CrateDB's functionality.

 # 3.4.3 (Difference)

"SELECT business_id FROM business_json WHERE business_id NOT IN (SELECT business_id FROM checkin_json);",

Explanation:

- This query selects business_id values from the business_json table that do not exist in the checkin_json table.
- It uses a subquery to find business_id values present in business_json but not in checkin_json.
- This approach efficiently identifies the set difference between the two tables.

 # 3.5.1 (Sorting over non-indexed column)

"SELECT * FROM business_json WHERE review_count >= 100 ORDER BY review_count

DESC LIMIT 100;",

 # 3.5.2 (Sorting over indexed column)

"SELECT * FROM business_json WHERE state = 'PA' ORDER BY name LIMIT 100;",

 # 3.6.1 (Apply distinct)

"SELECT DISTINCT city FROM business_json;",

 # 3.7 (MapReduce or equivalent aggregation)

"SELECT b.city, AVG(u.review_count) AS avg_reviews_per_user FROM business_json b JOIN review_json r ON b.business_id = r.business_id JOIN user_json u ON r.user_id = u.user_id GROUP BY b.city;"

To perform a MapReduce (or equivalent aggregation) type query in CrateDB, we can create a query that aggregates data across multiple tables to generate insights.

Explanation:

- We join the business_json, review_json, and user_json tables based on their respective keys (business_id and user_id).
- We group the results by the city from the business_json table.
- Then we calculate the average of the review_count column from the user_json table for each city.

This query effectively performs a distributed aggregation across the cluster, akin to the MapReduce paradigm, by distributing the data processing across multiple nodes in the CrateDB cluster. It aggregates the review counts from the users, grouped by the city they are in. This type of query is efficient for deriving insights from large datasets across distributed systems.