

```

WITH
-- Step 1: Precompute the creator for each message
message_creators AS (
    SELECT
        m.m_messageid,
        m.m_creatorid
    FROM
        message m
),

-- Step 2: Precompute foreign likes for each message
foreign_likes AS (
    SELECT
        l.l_messageid AS message_id,
        COUNT(l.l_personid) AS foreign_like_count
    FROM
        likes l
    LEFT JOIN knows k ON l.l_personid = k.k_person2id -- Check if the
liker is in the "knows" table
    LEFT JOIN message_creators mc ON k.k_personlid = mc.m_creatorid
    WHERE
        mc.m_creatorid IS NULL -- Exclude likes from friends of the
creator
    GROUP BY
        l.l_messageid
),

-- Step 3: Precompute total likes for each message
total_likes AS (
    SELECT
        l_messageid AS message_id,
        COUNT(l_personid) AS total_like_count
    FROM
        likes
    GROUP BY
        l_messageid
),

-- Step 4: Get the first foreign liker for each message
first_foreign_likers AS (
    SELECT DISTINCT ON (l.l_messageid)
        l.l_messageid AS message_id,
        l.l_personid AS person_id,
        l.l_creationdate AS like_creation_date
    FROM
        likes l

```

```

LEFT JOIN knows k ON l.l_personid = k.k_person2id
LEFT JOIN message_creators mc ON k.k_personlid = mc.m_creatorid
WHERE
    mc.m_creatorid IS NULL -- Exclude likes from friends of the
creator
    ORDER BY
        l.l_messageid, l.l_creationdate ASC -- Get the earliest like
for each message
    )
SELECT
    m.m_messageid AS message_id,          -- The ID of the message
    tl.total_like_count,                  -- Total number of likes for the
message
    fl.foreign_like_count,                -- Number of foreign likes for
the message
    COALESCE(fl.foreign_like_count * 1.0 / NULLIF(tl.total_like_count, 0),
0) AS foreign_like_ratio,                -- Ratio of foreign likes to
total likes
    CONCAT(p.p_firstname, ' ', p.p_lastname) AS first_foreign_liker_name,
-- Full name of the first foreign liker
    ffl.like_creation_date                -- Time when the first foreign
like was created
FROM
    message m
JOIN
    total_likes tl ON m.m_messageid = tl.message_id
LEFT JOIN
    foreign_likes fl ON m.m_messageid = fl.message_id
LEFT JOIN
    first_foreign_likers ffl ON m.m_messageid = ffl.message_id
LEFT JOIN
    person p ON ffl.person_id = p.p_personid
WHERE
    m.m_length > 100                      -- Filter messages longer than
100 characters
    AND tl.total_like_count >= 20          -- Filter messages with at least
20 total likes
    AND COALESCE(fl.foreign_like_count, 0) >= tl.total_like_count / 2
-- Ensure at least half of the
likes are foreign
ORDER BY
    foreign_like_ratio DESC;              -- Sort by the foreign-like
ratio in descending order

```

ChatGPT

To analyze and understand the query, generating an execution plan in PostgreSQL involves using the `EXPLAIN` or `EXPLAIN ANALYZE` command. Here's how you can execute it:

1. Generate Execution Plan with `EXPLAIN`

Use the `EXPLAIN` command before your query to get a high-level understanding of how PostgreSQL intends to execute it.

sql

Copy code

`EXPLAIN`

`WITH`

```
-- Step 1: Precompute the creator for each message
message_creators AS (
    SELECT
        m.m_messageid,
        m.m_creatorid
    FROM
        message m
),

-- Step 2: Precompute foreign likes for each message
foreign_likes AS (
    SELECT
        l.l_messageid AS message_id,
        COUNT(l.l_personid) AS foreign_like_count
    FROM
        likes l
        LEFT JOIN knows k ON l.l_personid = k.k_person2id -- Check if
the liker is in the "knows" table
        LEFT JOIN message_creators mc ON k.k_person1id =
mc.m_creatorid
    WHERE
        mc.m_creatorid IS NULL -- Exclude likes from friends of
the creator
    GROUP BY
        l.l_messageid
),

-- Step 3: Precompute total likes for each message
```

```

total_likes AS (
    SELECT
        l_messageid AS message_id,
        COUNT(l_personid) AS total_like_count
    FROM
        likes
    GROUP BY
        l_messageid
)
SELECT
    m.m_messageid AS message_id,           -- The ID of the message
    tl.total_like_count,                   -- Total number of likes for
the message
    fl.foreign_like_count,                 -- Number of foreign likes
for the message
    COALESCE(fl.foreign_like_count * 1.0 / NULLIF(tl.total_like_count,
0), 0) AS foreign_like_ratio
                                           -- Ratio of foreign likes to
total likes
FROM
    message m
JOIN
    total_likes tl ON m.m_messageid = tl.message_id
LEFT JOIN
    foreign_likes fl ON m.m_messageid = fl.message_id
WHERE
    m.m_length > 100                      -- Filter messages longer
than 100 characters
    AND tl.total_like_count >= 20          -- Filter messages with at
least 20 total likes
    AND COALESCE(fl.foreign_like_count, 0) >= tl.total_like_count / 2
                                           -- Ensure at least half of
the likes are foreign
ORDER BY
    foreign_like_ratio DESC;               -- Sort by the foreign-like
ratio in descending order

```

2. Generate Detailed Execution Plan with **EXPLAIN ANALYZE**

To measure the actual execution time and resources used, execute the query with **EXPLAIN ANALYZE**:

sql

Copy code

EXPLAIN ANALYZE

WITH

-- Step 1: Precompute the creator for each message

message_creators AS (

SELECT

m.m_messageid,

m.m_creatorid

FROM

message m

),

-- Step 2: Precompute foreign likes for each message

foreign_likes AS (

SELECT

l.l_messageid AS message_id,

COUNT(l.l_personid) AS foreign_like_count

FROM

likes l

LEFT JOIN knows k ON l.l_personid = k.k_person2id -- Check if
the liker is in the "knows" table

LEFT JOIN message_creators mc ON k.k_person1id =
mc.m_creatorid

WHERE

mc.m_creatorid IS NULL -- Exclude likes from friends of
the creator

GROUP BY

l.l_messageid

),

-- Step 3: Precompute total likes for each message

total_likes AS (

SELECT

l_messageid AS message_id,

COUNT(l_personid) AS total_like_count

```

        FROM
            likes
        GROUP BY
            l_messageid
    )
SELECT
    m.m_messageid AS message_id,          -- The ID of the message
    tl.total_like_count,                  -- Total number of likes for
the message
    fl.foreign_like_count,                -- Number of foreign likes
for the message
    COALESCE(fl.foreign_like_count * 1.0 / NULLIF(tl.total_like_count,
0), 0) AS foreign_like_ratio
-- Ratio of foreign likes to
total likes
FROM
    message m
JOIN
    total_likes tl ON m.m_messageid = tl.message_id
LEFT JOIN
    foreign_likes fl ON m.m_messageid = fl.message_id
WHERE
    m.m_length > 100                      -- Filter messages longer
than 100 characters
    AND tl.total_like_count >= 20          -- Filter messages with at
least 20 total likes
    AND COALESCE(fl.foreign_like_count, 0) >= tl.total_like_count / 2
-- Ensure at least half of
the likes are foreign
ORDER BY
    foreign_like_ratio DESC;              -- Sort by the foreign-like
ratio in descending order

```

Output and Interpretation

- **Execution Steps:** PostgreSQL will show how it processes each part of the query, including the Common Table Expressions (CTEs), joins, and filters.
- **Costs and Rows:** Look for costs (startup and total) and estimated/actual rows returned for each step.

- **Indexes:** If indexes exist, the plan will show if they are used (e.g., Index Scan).
- **Potential Bottlenecks:** Examine steps with high cost or high actual time.

Run the above commands on your database to see the actual execution plan specific to your data and indexes. If needed, share the generated plan, and I can help you interpret it further.

4o

You said:

I understand that to get the execution plan, the EXPLAIN keyword has to be prepended. What I want you to do is to act as the postgresql server, With the code I gave you to generate me the output of the execution plan.

ChatGPT said:

ChatGPT

I can't simulate PostgreSQL's actual query execution engine directly, but I can describe what the execution plan would likely include and how it works step by step. Here's an explanation of the likely execution plan, as though PostgreSQL were analyzing the query:

Likely Execution Plan Breakdown

1. CTE: `message_creators`

Operation: Sequential scan on the `message` table to retrieve `m_messageid` and `m_creatorid`.

- **Cost:** Moderate if the table is large, but lower if there's an index on `m_messageid` or `m_creatorid`.

2. **Step Output:** A temporary table containing `m_messageid` and `m_creatorid` for every message.

2. CTE: `foreign_likes`

Operation:

- Sequential scan on the `likes` table to get `l_messageid` and `l_personid`.
- **Nested Loop Join:** Perform a `LEFT JOIN` with the `knows` table.
 - Check if `l_personid` in `likes` matches `k_person2id` in `knows`.
- Another `LEFT JOIN` with `message_creators` on `k_person1id` and `m_creatorid` to exclude friends of the creator.
- Aggregate with `COUNT` to calculate the number of "foreign likes."
- **Cost:** High if the `likes` and `knows` tables are large, especially without indexes. Joins are likely the most expensive step here.

3. **Step Output:** A temporary table with `message_id` and `foreign_like_count`.
-

3. **CTE: `total_likes`**

Operation:

- Sequential scan on the `likes` table to calculate the total number of likes (`COUNT`) grouped by `l_messageid`.
 - **Cost:** Moderate, with performance improved if there's an index on `l_messageid`.
4. **Step Output:** A temporary table with `message_id` and `total_like_count`.
-

4. **Main Query: Filtering and Joining**

- **Join 1:**
 - `INNER JOIN` between `message` and `total_likes` on `m_messageid = message_id`.
 - **Cost:** Moderate, especially with indexing on `m_messageid`.
 - **Join 2:**
 - `LEFT JOIN` with `foreign_likes` on `m_messageid = message_id`.
 - **Cost:** Moderate, depending on the size of the `foreign_likes` CTE output.
 - **Filter Conditions:**
 - `m_length > 100`: Applies a filter to `message` table.
 - `total_like_count >= 20`: Applies a filter to `total_likes`.
 - `foreign_like_count >= total_like_count / 2`: Applies an additional filter using the joined results.
 - **Sort:**
 - Sort results by `foreign_like_ratio DESC`.
 - **Cost:** Sorting is potentially expensive, especially if there's no preexisting order in the data.
-

Hypothetical Execution Plan Output (Structured Example)

plaintext

Copy code

```
Sort (cost=XXXXX..XXXXX rows=N width=W)
  Sort Key: COALESCE(fl.foreign_like_count * 1.0 /
NULLIF(tl.total_like_count, 0), 0) DESC
  -> Nested Loop Left Join (cost=XXXXX..XXXXX rows=N width=W)
    Join Filter: m.m_messageid = fl.message_id
    -> Nested Loop (cost=XXXXX..XXXXX rows=N width=W)
      Join Filter: m.m_messageid = tl.message_id
      -> Seq Scan on message m (cost=XXXXX..XXXXX rows=N width=W)
        Filter: (m_length > 100)
      -> Seq Scan on total_likes tl (cost=XXXXX..XXXXX rows=N
width=W)
        Filter: (total_like_count >= 20)
    -> Seq Scan on foreign_likes fl (cost=XXXXX..XXXXX rows=N
width=W)
      Filter: (foreign_like_count >= (total_like_count / 2))
```

Key Takeaways

1. **Expensive Joins:** The `LEFT JOIN` operations in the `foreign_likes` CTE can be resource-intensive, especially if the `knows` and `message_creators` tables are large. Indexes on relevant columns can help.
2. **Filter Efficiency:** Applying filters earlier (e.g., filtering messages with `m_length > 100` or likes with `total_like_count >= 20`) reduces the rows processed in later stages.
3. **Index Usage:** Adding indexes on frequently joined and filtered columns (`m_messageid`, `l_messageid`, `l_personid`, `k_person1id`, `k_person2id`) will greatly improve performance.
4. **Aggregate Optimization:** Precomputing aggregates (e.g., `COUNT` operations) in indexed materialized views can optimize this query.

For the actual output with costs and rows, run the query with `EXPLAIN ANALYZE` in your PostgreSQL environment. Let me know if you'd like to refine the query further based on its performance!