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

),

-- 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\_person1id = 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 teh 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.
3. **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.
4. **Step Output:** A temporary table with message\_id and foreign\_like\_count.
5. **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.
6. **Step Output:** A temporary table with message\_id and total\_like\_count.
7. **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!