

Execution Plan Explanation



Can you generate an execution plan like PostgreSQL Explain Analyze method for this query:

```
"EXPLAIN ANALYZE
WITH
  message_likes AS (
    SELECT
      l_messageid AS message_id,
      COUNT(*) AS total_likes
    FROM
      likes
    GROUP BY
      l_messageid
    HAVING
      COUNT(*) >= 20
  ),
  creator_friends AS (
    -- Direct friends of the message creator
    SELECT
      k_person2id AS friend_id,
      k_person1id AS creator_id
    FROM
      knows
  ),
  friends_of_friends AS (
    -- Friends of the creator's friends
    SELECT
      k.k_person2id AS person_id,
      cf.creator_id
    FROM
      knows k
    JOIN creator_friends cf ON k.k_person1id = cf.friend_id
  ),
  extended_network AS (
    -- Union of direct friends and friends of friends
    SELECT
      friend_id AS person_id,
      creator_id
    FROM
      creator_friends
    UNION
    SELECT
      person_id,
      creator_id
    FROM
      friends_of_friends
```

```

),
foreign_likes AS (
  SELECT
    l.l_messageid AS message_id,
    l.l_personid AS foreign_person_id,
    l.l_creationdate AS like_time
  FROM
    likes l
  JOIN message m ON l.l_messageid = m.m_messageid
  LEFT JOIN extended_network en ON en.person_id = l.l_personid
    AND en.creator_id = m.m_creatorid
  WHERE
    en.person_id IS NULL -- The liker is not in the extended network of the creator
),
first_foreign_likes AS (
  SELECT DISTINCT ON (fl.message_id)
    fl.message_id,
    fl.foreign_person_id,
    fl.like_time
  FROM
    foreign_likes fl
  ORDER BY
    fl.message_id, fl.like_time ASC
),
foreign_like_counts AS (
  SELECT
    fl.message_id,
    COUNT(*) AS foreign_likes
  FROM
    foreign_likes fl
  GROUP BY
    fl.message_id
),
filtered_messages AS (
  SELECT
    ml.message_id,
    ml.total_likes,
    flc.foreign_likes,
    ffl.foreign_person_id,
    ffl.like_time AS first_foreign_like_time
  FROM
    message_likes ml
  JOIN foreign_like_counts flc ON ml.message_id = flc.message_id
  JOIN first_foreign_likes ffl ON ml.message_id = ffl.message_id
  WHERE
    flc.foreign_likes >= ml.total_likes / 2
)
SELECT
  fm.message_id,
  fm.total_likes,
  fm.foreign_likes,

```

```
p.p_firstname || ' ' || p.p_lastname AS first_foreign_liker_name,
fm.first_foreign_like_time
FROM
  filtered_messages fm
JOIN message m ON fm.message_id = m.m_messageid
JOIN person p ON fm.foreign_person_id = p.p_personid
WHERE
  m.m_length > 100
ORDER BY
  (fm.foreign_likes::decimal / fm.total_likes) DESC; "
```



To generate an execution plan for the query you provided, the `EXPLAIN ANALYZE` method in PostgreSQL gives a detailed execution plan that helps optimize and debug complex queries. Since I can't run a PostgreSQL query directly in this environment, I will walk you through a simplified approximation of what the execution plan might look like, based on the common operations performed in the query.

Here's an approximate execution plan based on the query structure:

Execution Plan Explanation:

1. CTEs (Common Table Expressions)

- The query starts with several **WITH** clauses, creating intermediate results. The first step in an `EXPLAIN ANALYZE` output would involve the execution of these CTEs.

1.1 message_likes CTE:

plaintext

```
CTE Scan on message_likes (cost=.., rows=...)
-> GroupAggregate (cost=.., rows=...)
    -> Seq Scan on likes (cost=.., rows=...)
        Filter: (COUNT(*) >= 20)
```

Explanation:

- The `message_likes` CTE calculates the total likes per message, filtering out those with fewer than 20 likes.
- PostgreSQL will likely perform a **sequential scan** on the `likes` table with a **GroupAggregate** to count likes by `l_messageid`.

1.2 creator_friends CTE:

plaintext

```
CTE Scan on creator_friends (cost=.., rows=...)
-> Seq Scan on knows (cost=.., rows=...)
```

Explanation:

- The ``creator_friends`` CTE joins the ``knows`` table to find the direct friends of the message creator.

1.3 friends_of_friends CTE:

plaintext

```
CTE Scan on friends_of_friends (cost=.., rows=...)
-> Hash Join (cost=.., rows=...)
    -> Seq Scan on knows (cost=.., rows=...)
    -> Hash (cost=.., rows=...)
        -> CTE Scan on creator_friends
```

Explanation:

- The ``friends_of_friends`` CTE joins the ``knows`` table again with ``creator_friends`` to find friends of the creator's friends.
- A **Hash Join** might be used here to join the ``knows`` table and the ``creator_friends`` CTE.

1.4 extended_network CTE:

plaintext

```
CTE Scan on extended_network (cost=.., rows=...)
-> Append (cost=.., rows=...)
    -> Seq Scan on creator_friends (cost=.., rows=...)
    -> Seq Scan on friends_of_friends (cost=.., rows=...)
```

Explanation:

- The ``extended_network`` CTE creates a union of the direct friends and friends of friends.
- PostgreSQL uses an **Append** operation to combine the results from ``creator_friends`` and ``friends_of_friends``.

1.5 foreign_likes CTE:

plaintext

```
CTE Scan on foreign_likes (cost=.., rows=...)
-> Hash Join (cost=.., rows=...)
    -> Seq Scan on likes (cost=.., rows=...)
    -> Hash (cost=.., rows=...)
        -> CTE Scan on extended_network
```

Explanation:

- The ``foreign_likes`` CTE identifies messages liked by people outside the extended network.
- A **Hash Join** is performed between the ``likes`` table and the ``extended_network`` CTE.

1.6 first_foreign_likes CTE:

plaintext

```
CTE Scan on first_foreign_likes (cost=.., rows=...)
-> Unique (cost=.., rows=...)
    -> Sort (cost=.., rows=...)
        -> CTE Scan on foreign_likes
```

Explanation:

- The ``first_foreign_likes`` CTE retrieves the first foreign like for each message.
- A **Sort** operation is applied to order the likes, and then a **Unique** operation is used to select the first like per message.

1.7 foreign_like_counts CTE:

plaintext

```
CTE Scan on foreign_like_counts (cost=.., rows=...)
-> GroupAggregate (cost=.., rows=...)
    -> CTE Scan on foreign_likes
```

Explanation:

- The ``foreign_like_counts`` CTE counts the foreign likes for each message.

2. Final Query:

- After executing all CTEs, the query proceeds to join the results together in the main part of the query.

plaintext

```
Final Query (cost=.., rows=...)
-> Sort (cost=.., rows=...)
    -> Hash Join (cost=.., rows=...)
        -> Hash Join (cost=.., rows=...)
            -> Seq Scan on filtered_messages (cost=.., rows=...)
            -> Seq Scan on message (cost=.., rows=...)
        -> Seq Scan on person (cost=.., rows=...)
```

Explanation:

- The main query starts by joining the ``filtered_messages``, ``message``, and ``person`` tables. It also applies a **Sort** on the final result set based on the ratio of foreign likes to total likes.
- There will be **Hash Joins** between ``filtered_messages`` and ``message`` as well as ``foreign_person_id`` from the ``person`` table.
- PostgreSQL will likely use a **Sequential Scan** on both the ``message`` and ``person`` tables since no index is specified in the query.

Overall Cost Estimate:

The total cost of the query depends on factors like table sizes and whether there are any indexes on the relevant columns. It's important to note that PostgreSQL may use **Hash Joins**, **Sorts**, **Aggregations**, and **Seq Scans** to execute the query.

- The cost for the CTEs will likely involve sequential scans and some hash joins, which could be expensive if the ``likes``, ``knows``, or ``message`` tables are large.
- For large tables, consider adding indexes on frequently joined columns (e.g., ``l_messageid``, ``l_personid``, ``m_creatorid``, etc.).

This execution plan approximation outlines the logical flow PostgreSQL might follow. To see the actual execution plan with real-time statistics, you should run the query with ``EXPLAIN ANALYZE`` in your PostgreSQL environment.



Can you give it to me as one text file + add in values that you think would be plausible



I have created an execution plan text file with plausible values based on your query. You can download it from the link below: