QUERY OPTIMIZATION

COMPLEX QUERY 1 :
(Top 5 restaurants reviewed most by users whose friends also reviewed them)

BAD QUERY -

```
WITH UserFriendsReviews AS (
       SELECT r.business_id, uf.user_id, COUNT(DISTINCT r.review_id) AS
review_count_by_friends
       FROM normalized_user_reviews r
       JOIN user_stats uf ON r.user_id = uf.user_id
       GROUP BY r.business_id, uf.user_id
    ),
    PopularWithFriends AS (
       SELECT ufr.business_id, SUM(ufr.review_count_by_friends) AS
total_reviews_by_friends
       FROM UserFriendsReviews ufr
       GROUP BY ufr.business_id
    ),
    PopularWithFriendsWithRank AS (
       SELECT business_id, total_reviews_by_friends,
              ROW_NUMBER() OVER (ORDER BY total_reviews_by_friends DESC) AS
rank
       FROM PopularWithFriends
    ),
    TopPopularWithFriends AS (
       SELECT business_id, total_reviews_by_friends
       FROM PopularWithFriendsWithRank
       WHERE rank <= 5
    )
    SELECT b.name AS restaurant_name, bl.city AS city,
pwf.total_reviews_by_friends
    FROM (
       SELECT tpwf.business_id, tpwf.total_reviews_by_friends
       FROM TopPopularWithFriends tpwf
       JOIN PopularWithFriends pf ON tpwf.business_id = pf.business_id
    ) pwf
    JOIN normalized_businesses b ON pwf.business_id = b.business_id
    JOIN business_locations bl ON b.location_id = bl.location_id;
```

EXECUTION TIME - 5.7sec

GOOD QUERY -

```
WITH UserFriendsReviews AS (
       SELECT r.business_id, uf.user_id, COUNT(DISTINCT r.review_id) AS
review_count_by_friends
       FROM normalized_user_reviews r
       JOIN user_stats uf ON r.user_id = uf.user_id
       GROUP BY r.business_id, uf.user_id
```

```
    ),
    PopularWithFriends AS (
       SELECT ufr.business_id, SUM(ufr.review_count_by_friends) AS
total_reviews_by_friends
       FROM UserFriendsReviews ufr
       GROUP BY ufr.business_id
       ORDER BY total_reviews_by_friends DESC
       LIMIT 5
    )
    SELECT b.name AS restaurant_name, bl.city as city,
pwf.total_reviews_by_friends
    FROM PopularWithFriends pwf
    JOIN normalized_businesses b ON pwf.business_id = b.business_id
    JOIN business_locations bl ON b.location_id = bl.location_id;
```

EXECUTION TIME - 1.3sec

CREATING INDEXES FOR OPTIMIZATION -

-- **Index to optimize joins and distinct operations on user reviews**

```
CREATE INDEX idx_reviews_business_user ON
normalized_user_reviews(business_id, user_id, review_id);
```

-- **Index to optimize the join on user_stats**

```
CREATE INDEX idx_user_stats_user ON user_stats(user_id);
```

-- **Composite index to optimize joins with business and location**

```
CREATE INDEX idx_business_location ON normalized_businesses(business_id,
location_id);
```

-- **Index to optimize lookups in business_locations**

```
CREATE INDEX idx_locations_location ON business_locations(location_id);
```

Here are two key ways caching was used for our query optimization:

Pre-aggregated Data Caching =>

- What We Did: Frequently accessed and computationally intensive aggregations, like review counts (`UserFriendsReviews`) or weekly check-ins (`TotalCheckins`), were precomputed and stored in **summary tables** or materialized views.
- How It Helped :- Instead of recalculating counts or totals every time the query ran, we fetched pre-aggregated results.
  - This reduced execution time significantly because the database only queried and joined smaller, precomputed datasets.

Application-Level Caching for Top Results =>

What We Did: Hot data, such as the top 5 businesses (by reviews or check-ins), was cached in-memory using the redis tool.
How It Helped:
- Avoided querying the database repeatedly for the same results.
- Cached data was refreshed periodically (e.g., hourly or daily), ensuring up-to-date results while improving response time from 5.7 seconds to 0.3 seconds.

These two approaches ensured faster analytics while reducing database strain.

BETTER QUERY -

```
WITH UserFriendsReviews AS (
       SELECT r.business_id,
              COUNT(DISTINCT r.review_id) AS review_count_by_friends
       FROM normalized_user_reviews r
       JOIN user_stats uf ON r.user_id = uf.user_id
       GROUP BY r.business_id
    ),
    PopularWithFriends AS (
       SELECT business_id,
              SUM(review_count_by_friends) AS total_reviews_by_friends
       FROM UserFriendsReviews
       GROUP BY business_id
       ORDER BY total_reviews_by_friends DESC
       LIMIT 5
    )
SELECT b.name AS restaurant_name,
     bl.city AS city,
     pwf.total_reviews_by_friends
FROM PopularWithFriends pwf
JOIN normalized_businesses b ON pwf.business_id = b.business_id
JOIN business_locations bl ON b.location_id = bl.location_id;
```

EXECUTION TIME - 1.1sec
EXECUTION TIME (USING INDEXING) - 0.3sec

INDEXES USED -

idx_reviews_business_user

- Speeds up GROUP BY r.business_id and COUNT(DISTINCT r.review_id) by allowing efficient access to reviews for each business.

idx_user_stats_user

- Optimizes the join between normalized_user_reviews and user_stats by facilitating fast lookups on user_id.

idx_business_location

- Accelerates the join between normalized_businesses and business_locations by quickly matching business_id and location_id.

idx_locations_location

- Optimizes lookups on location_id in the business_locations table.

Key Differences Between Better and Bad Query -

1. Granularity of Aggregation:
    - Query A:
        - In the UserFriendsReviews CTE, it aggregates directly at the business_id level, reducing the number of rows earlier in the process. It calculates COUNT(DISTINCT r.review_id) for each business_id without including user_id.
    - Query B:
        - In the UserFriendsReviews CTE, it aggregates at the business_id and user_id level, which increases the number of intermediate rows processed. It calculates COUNT(DISTINCT r.review_id) per user per business, leading to unnecessary granularity for this use case.

    Query A processes fewer rows in later stages since it aggregates at the appropriate level from the start.
2. Ranking vs. Direct Limit:
    - Query A:
        - Uses LIMIT 5 directly in the PopularWithFriends CTE to filter for the top 5 businesses with the highest total_reviews_by_friends.
    - Query B:
        - Introduces an unnecessary ROW_NUMBER() function in the PopularWithFriendsWithRank CTE to assign ranks, followed by a filter on rank <= 5.

    Query A avoids the computational overhead of ranking all rows and simply limits the result to the top 5, which is more efficient.
3. Unnecessary Joins:
    - Query A:
        - Joins only the required tables (normalized_businesses and business_locations) with the PopularWithFriends CTE.
    - Query B:
        - Introduces an additional join in the TopPopularWithFriends CTE, unnecessarily rejoining PopularWithFriends to filter the top-ranked businesses.

    Query A avoids redundant joins, reducing query complexity and execution time.
4. Nested Subqueries:
    - Query A:
        - Uses straightforward joins and avoids excessive nesting.
    - Query B:

        - Contains nested subqueries in the final SELECT statement, adding complexity and potential overhead.

    Query A is more streamlined and easier for the database optimizer to execute efficiently.

5. Focus on Simplicity:
    - Query A:
        - Uses a minimal number of steps to achieve the desired result.
    - Query B:
        - Contains unnecessary intermediate steps, such as the ROW_NUMBER() CTE and redundant joins, that do not add value to the final result.

    Query A is simpler, easier to maintain, and faster due to reduced processing.

COMPLEX QUERY 2 :
(Weekly Checkin Distribution)

BAD QUERY -

```
WITH TotalCheckins AS (
  SELECT
    c.business_id,
    COUNT(c.checkin_id) AS total_checkins
  FROM normalized_checkin c
  GROUP BY c.business_id
  HAVING COUNT(c.checkin_id) > (SELECT MIN(COUNT(checkin_id)) FROM
normalized_checkin GROUP BY business_id)
  ORDER BY total_checkins DESC
),
FilteredCheckins AS (
  SELECT tc.business_id, tc.total_checkins
  FROM TotalCheckins tc
  WHERE tc.total_checkins >= ALL (SELECT total_checkins FROM TotalCheckins)
),
WeeklyCheckinDistribution AS (
  SELECT
    fc.business_id,
    c.weekday AS checkin_weekday,
    SUM(DISTINCT c.checkins) AS total_checkins
  FROM normalized_checkin c
  JOIN FilteredCheckins fc ON c.business_id = fc.business_id
  GROUP BY fc.business_id, c.weekday, c.checkin_id
)
SELECT
  (SELECT b.name FROM normalized_businesses b WHERE b.business_id =
wcd.business_id) AS restaurant_name,
  wcd.checkin_weekday,
  wcd.total_checkins
```

```
  FROM WeeklyCheckinDistribution wcd
  JOIN normalized_businesses b ON wcd.business_id = b.business_id
  WHERE wcd.total_checkins > 0
  ORDER BY (SELECT name FROM normalized_businesses WHERE business_id =
wcd.business_id), wcd.checkin_weekday;
```

EXECUTION TIME - 9.3sec

BETTER QUERY -

```
WITH TotalCheckins AS (
  SELECT
    c.business_id,
    COUNT(c.checkin_id) AS total_checkins
  FROM normalized_checkin c
  GROUP BY c.business_id
  ORDER BY total_checkins DESC
  LIMIT 5
),
WeeklyCheckinDistribution AS (
  SELECT
    tc.business_id,
    c.weekday AS checkin_weekday,
    SUM(c.checkins) AS total_checkins
  FROM normalized_checkin c
  JOIN TotalCheckins tc ON c.business_id = tc.business_id
  GROUP BY tc.business_id, c.weekday
)
SELECT
  b.name AS restaurant_name,
  wcd.checkin_weekday,
  wcd.total_checkins
FROM WeeklyCheckinDistribution wcd
JOIN normalized_businesses b ON wcd.business_id = b.business_id
ORDER BY restaurant_name, wcd.checkin_weekday;
```

EXECUTION TIME - 0.5sec

SIMILARLY, THE REMAINING 3 COMPLEX QUERIES HAVE BEEN OPTIMIZED!