Paseo Posse – Hyperlocal Delivery System

Query optimization, distributed indexing, and the use of stored procedures are crucial aspects of improving the performance and efficiency of a database system, especially in a distributed environment like the hyperlocal delivery project we are doing.

Query Optimization:

Partitioning for Pruning:

Partitioning tables based on regions (ZIP codes) enables data pruning, where the database engine only accesses the partitions relevant to the specific query. This reduces the amount of data scanned and improves query performance.

Use of UUID in Inventory Table:

The use of UUIDs as primary keys in the Inventory table, along with the distributed structure, ensures unique identifiers across partitions. This can enhance distributed indexing efficiency.

Stored Procedures:

Avoiding N+1 Queries:

N+1 query problems occur when a script or application issues N+1 separate queries to fetch related data. Using stored procedures consolidates these queries into a single stored procedure call, reducing the number of round trips between the application and the database.

Avoiding Multiple Calls from Python Script:

By encapsulating complex logic and queries in stored procedures, you reduce the need for multiple calls from the Python script. This can result in better performance and code maintainability.

```
pratyushpandey@Pratyushs-Air part-3 % python3 postgres_indexing_script.py
hello tables...
Creating database...
<connection object at 0x101584040; dsn: 'user=postgres password=xxx dbname=delivery_system host=localhost port=5432', closed: 0>
Creating Order table...
Creating Order table...
Creating Order_Item table...
Creating Order_Item table...
Creating Delivery_Agent table...
Inserting data into Delivery_Agent table...
Creating Inventory table...
Inserting data into Inventory table...
========================================UnOptimized reserve order items========================================
Total time taken to reserve order items for 400 orders: 9.44685435295105 seconds
pratyushpandey@Pratyushs-Air part-3 % python3 postgres_indexing_script.py
hello tables...
Creating database...
<connection object at 0x102cc0040; dsn: 'user=postgres password=xxx dbname=delivery_system host=localhost port=5432', closed: 0>
Creating Order table...
Creating Order table...
Creating Order_Item table...
Creating Order_Item table...
Creating Delivery_Agent table...
Inserting data into Delivery_Agent table...
Creating Inventory table...
Inserting data into Inventory table...
========================================Optimized reserve order items by solving N+1 problem ========================================
Total time taken to reserve order items for 400 orders: 6.978225946426392 seconds
```

As we can see from the output screenshot, there is a 26.12% of improvement in execution time.

Distributed Indexing on Partitioned Tables:

Distributed indexing involves creating and managing indexes across multiple nodes or partitions in a distributed database. This approach is especially beneficial in scenarios where data is horizontally partitioned, as is the case in our hyperlocal delivery project where tables are partitioned based on regions (ZIP codes). Below we have explained how distributed indexing works and its impact on the Inventory and Orders tables:

Inventory Table:

How Indexing Improves Select Queries:

Scenario:

The Inventory table contains information about the availability of various items in different regions. Given the distributed nature of the system, it's essential to have efficient indexes on columns commonly used in SELECT queries.

Impact:

When a SELECT query is executed for a specific med_id and zip_code, the database engine can utilize the index on the relevant partition, significantly reducing the amount of data that needs to be scanned. This results in faster query performance.

We analyze the execution times using the query tool in pgadmin:

Output without Indexing the Partitioned Inventory Table:

```
explain analyze SELECT * FROM INVENTORY WHERE MED_ID = 1 AND ORDER_ID IS NULL
AND ZIP_CODE = 85200 LIMIT 5;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=0.00..10.25 rows=5 width=20) (actual time=0.036..0.037 rows=5 loops=1)
2	-> Seq Scan on inventory_zip_code_85200 inventory (cost=0.00..82.00 rows=40 width=20) (actual time=0.
3	Filter: ((order_id IS NULL) AND (med_id = 1) AND (zip_code = 85200))
4	Planning Time: 0.206 ms
5	Execution Time: 0.052 ms

Output after Indexing the Partitioned Inventory Table:

```
explain analyze SELECT * FROM INVENTORY WHERE MED_ID = 1 AND ORDER_ID IS NULL
AND ZIP_CODE = 85200 LIMIT 5;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=0.28..1.38 rows=5 width=20) (actual time=0.016..0.017 rows=5 loops=1)
2	-> Index Scan using inventory_zip_code_85200_index on inventory_zip_code_85200 inventory (cost=0.28..
3	Index Cond: (med_id = 1)
4	Filter: ((order_id IS NULL) AND (zip_code = 85200))
5	Planning Time: 1.749 ms
6	Execution Time: 0.034 ms

Result: As we can see, the execution time after indexing has reduced from 0.052 ms to 0.034 ms (34% decrease in time)