GROUP PROJECT

Team Name: Cluster Crafters (Mobile Phone Supply Chain Management)
PART 3: Query Processing and Optimization Techniques

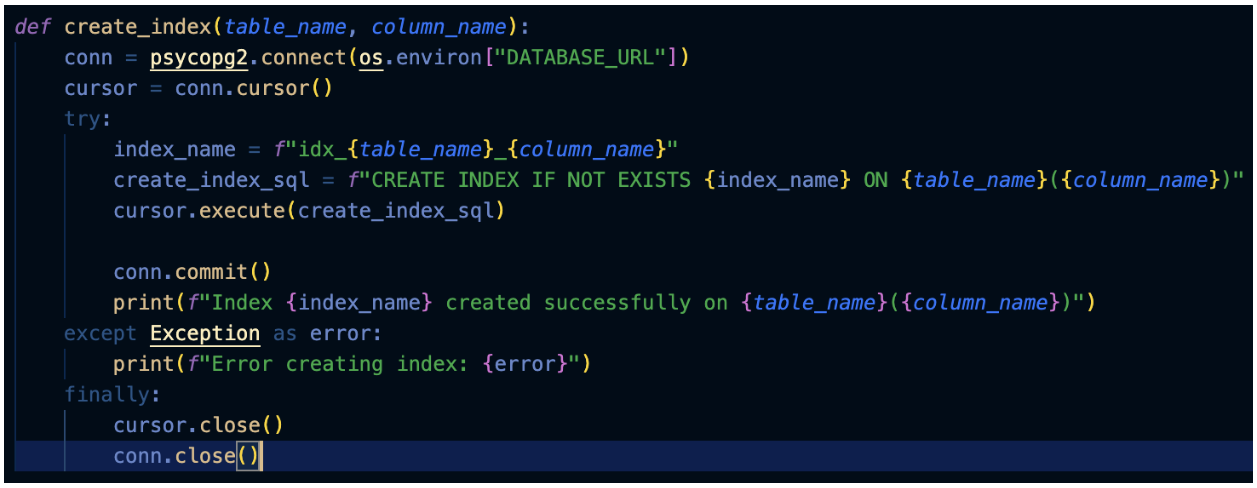
In this section of the project, we focus on improving data retrieval performance by implementing smart indexing and caching strategies. The goal is to make data searches quicker and reduce resource utilization, ultimately enhancing the efficiency of the distributed database system. As mentioned in the previous parts we have utilized CockroachDB Cloud as our database solution for this project.

Indexing Strategy

Index Creation Function

We have implemented a create_index, to create and manage indexes for specific columns in our database tables. This function takes two parameters: table_name and column_name. The key features of this index creation function are as follows:

- It dynamically generates a unique index name based on the provided table_name and column_name.
- The function checks whether an index with the same name exists and creates it only if it doesn't exist, ensuring that we do not create duplicate indexes.
- Upon successful index creation, it commits the changes to the database and logs the operation.
- It can also handle exceptions.



```python
def create_index(table_name, column_name):
    conn = psycopg2.connect(os.environ["DATABASE_URL"])
    cursor = conn.cursor()
    try:
        index_name = f"idx_{table_name}_{column_name}"
        create_index_sql = f"CREATE INDEX IF NOT EXISTS {index_name} ON {table_name}({column_name})"
        cursor.execute(create_index_sql)

        conn.commit()
        print(f"Index {index_name} created successfully on {table_name}({column_name})")
    except Exception as error:
        print(f"Error creating index: {error}")
    finally:
        cursor.close()
        conn.close()
```

The above image shows our indexing implementation.

Benefits of Indexing

By creating indexes on specific columns, we can significantly enhance query performance. Indexes provide a fast and efficient way to locate data, reducing the time and resources required for data retrieval operations. This indexing strategy helps in optimizing data searches for various aspects of mobile phone supply chain management, such as product tracking, inventory management, and order processing.

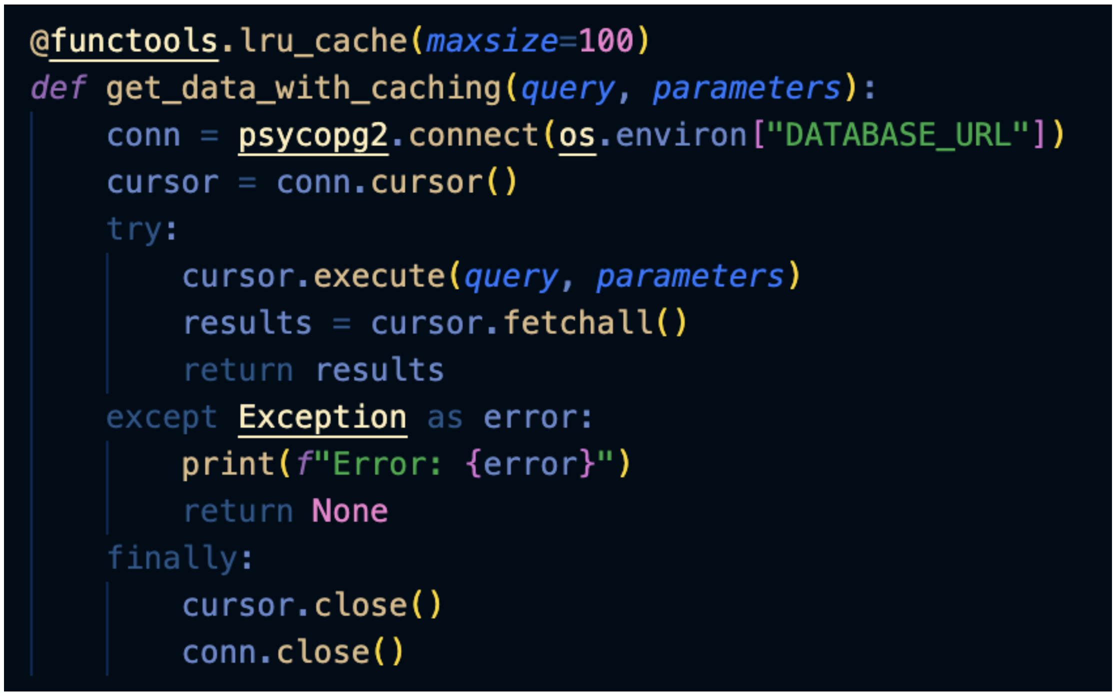
Caching Strategy

Caching Function

We have implemented a caching mechanism using the lru_cache from functools in Python. The get_data_with_caching function is responsible for caching query results, which results in quicker data access. Key features of this caching function include:

- We have used lru_cache(maxsize=100), which limits the cache size to 100 most recently used query results. This helps in managing memory consumption.
- It executes the provided SQL query with the given parameters and returns the results.
- Suppose the same query with the same parameters is requested again. In that case, the function retrieves the results from the cache instead of re-executing the query, improving response times and reducing resource usage.

- Exception handling ensures that any errors during query execution are logged without disrupting the application.



```
@functools.lru_cache(maxsize=100)
def get_data_with_caching(query, parameters):
    conn = psycopg2.connect(os.environ["DATABASE_URL"])
    cursor = conn.cursor()
    try:
        cursor.execute(query, parameters)
        results = cursor.fetchall()
        return results
    except Exception as error:
        print(f"Error: {error}")
        return None
    finally:
        cursor.close()
        conn.close()
```

The above image shows our caching implementation.

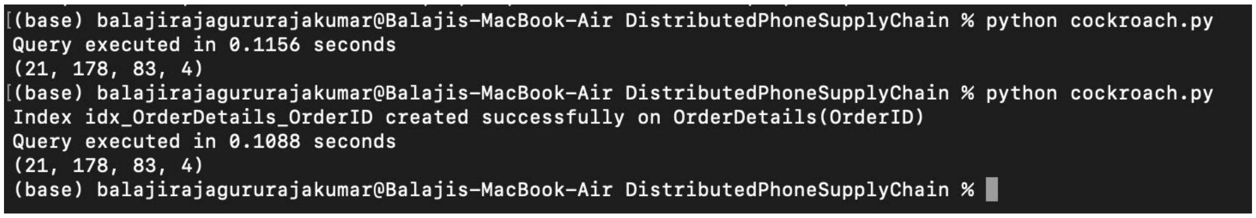
Benefits of Caching

Caching is crucial in optimizing data retrieval, especially for frequently accessed data. By storing query results in memory, we reduce the need for repetitive database queries, leading to faster response times and lower resource utilization. This is particularly beneficial in scenarios where data retrieval needs to be quick and efficient, such as tracking mobile phone shipments and managing real-time inventory levels.

Evaluation

To evaluate the effectiveness of our indexing and caching strategies, we will conduct performance testing and benchmarking. This evaluation will involve comparing query execution times and resource utilization before and after implementing these strategies. Additionally, we will assess the impact on the overall system performance as we scale up the database and workload.

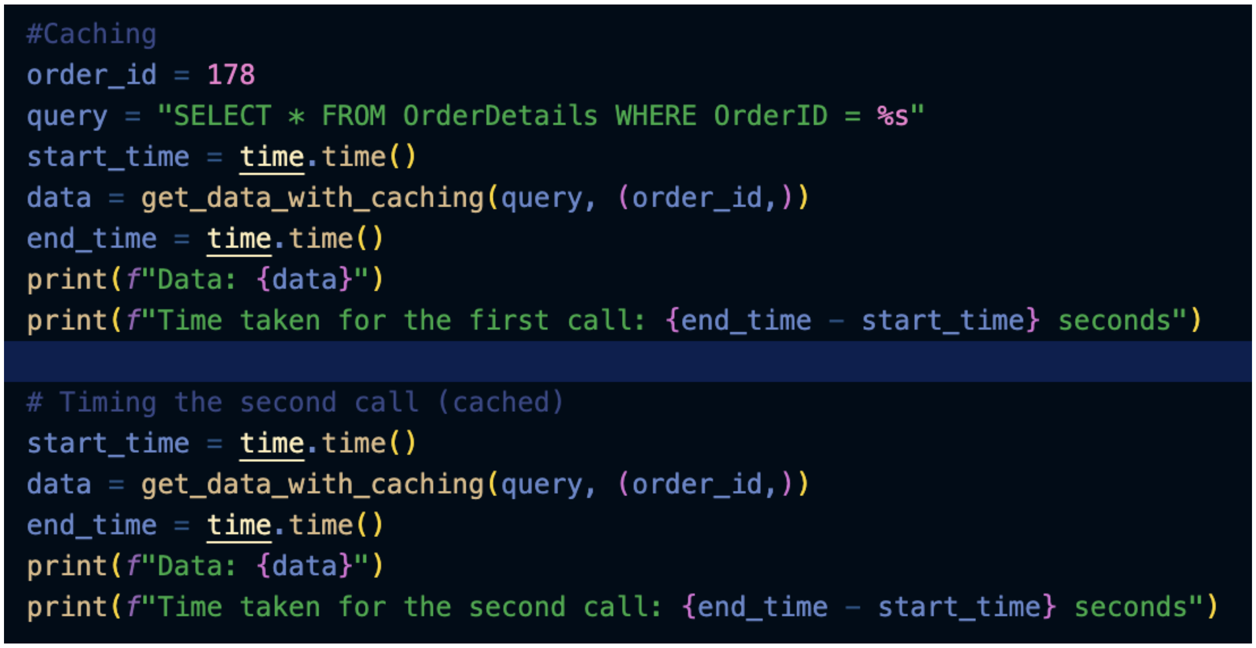
The following image shows how performance has improved by implementing indexing:



```
[(base) balajirajagururajakumar@Balajis-MacBook-Air DistributedPhoneSupplyChain % python cockroach.py
Query executed in 0.1156 seconds
(21, 178, 83, 4)
[(base) balajirajagururajakumar@Balajis-MacBook-Air DistributedPhoneSupplyChain % python cockroach.py
Index idx_OrderDetails_OrderID created successfully on OrderDetails(OrderID)
Query executed in 0.1088 seconds
(21, 178, 83, 4)
(base) balajirajagururajakumar@Balajis-MacBook-Air DistributedPhoneSupplyChain %
```

In the first run of the application, we did not implement indexing. However, in the subsequent run, we introduced indexing into the system and executed the program using the same query.

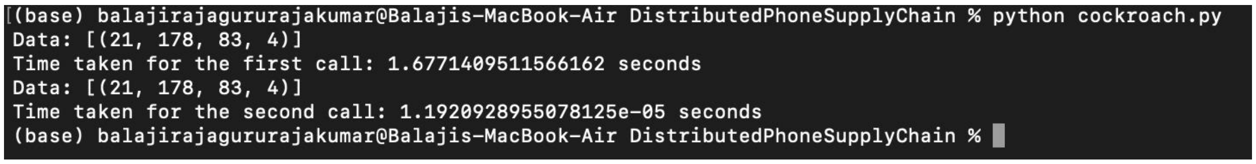
The image below shows how we have evaluated the caching function.



```
#Caching
order_id = 178
query = "SELECT * FROM OrderDetails WHERE OrderID = %s"
start_time = time.time()
data = get_data_with_caching(query, (order_id,))
end_time = time.time()
print(f"Data: {data}")
print(f"Time taken for the first call: {end_time - start_time} seconds")

# Timing the second call (cached)
start_time = time.time()
data = get_data_with_caching(query, (order_id,))
end_time = time.time()
print(f"Data: {data}")
print(f"Time taken for the second call: {end_time - start_time} seconds")
```

The next image shows how performance has improved by implementing Caching.



```
[(base) balajirajagururajakumar@Balajis-MacBook-Air DistributedPhoneSupplyChain % python cockroach.py
Data: [(21, 178, 83, 4)]
Time taken for the first call: 1.6771409511566162 seconds
Data: [(21, 178, 83, 4)]
Time taken for the second call: 1.1920928955078125e-05 seconds
(base) balajirajagururajakumar@Balajis-MacBook-Air DistributedPhoneSupplyChain %
```

Conclusion

In Part 3 of the project, we have successfully implemented indexing and caching strategies to improve data retrieval performance. These techniques aim to make data searches quicker and reduce resource usage, ultimately enhancing the efficiency of our distributed database system.