Semester I, AY 2023/2024
CS4224/CS5424 Distributed Databases
Project Final Report

TeamB

Name	**Student ID**
Aakash Sudersan Selvaganapathy	A0254232N
Adithya Ragothaman	A0274780W
Gopika Sarasvathi Kothandaraman	A0274980R
Param Singh Ahuja	A0255953U
Sylviya Alexander	A0276584M

Team Member Contributions

Aakash Sudersan Selvaganapathy

1. Citus installation and Setup
2. Data Modelling for Citus and Cassandra Optimization.
3. Implementation of Transactions in Citus (New Order, Payment, Delivery, Order-status and Stock level)
4. Implementation of Transactions in Optimized Cassandra data model (Popular Item, Top Balance Customer and Related Customers)
5. Driver and ReadMe for Citus and Report writing.
6. Data insertion scripts for Optimized Cassandra tables.

Adithya Ragothaman

1. Cassandra Installation and Setup
2. Worked on Data Insertion scripts for Cassandra
3. Implementation of Initial Transactions and Data modelling in Cassandra (Delivery, Order Status, Stock-Level, Related-Customer)
4. Optimized implementation of Transactions (Order Status)
5. Wrote the readme for Cassandra Setup and installation,Report

Gopika Sarasvathi Kothandaraman

1. Data Modelling for Citus
2. Creation and Insertion scripts for Citus
3. Implementation of Transactions in Citus (Top-balance, Popular-Item, Related-Customers)
4. Implementation of Transactions in Cassandra (Top-balance, Popular-Item, Related-Customers)
5. Optimize Transaction (Top-balance, Popular-Item, Related-Customers) in Citus, Readme and Report Writing

Param Singh Ahuja

1. Cassandra Multinode Setup
2. Implementation of New order transaction(old model)
3. Data modelling and data Insertion scripts for Cassandra after optimization
4. Implementation of New Order, Payment and Delivery transactions as per new model
5. Script for measuring dbstate and Report

Sylviya Alexander

1. Cassandra Installation and Setup
2. Worked on Insertion scripts for Cassandra
3. Initial implementation of Transactions and data modelling in Cassandra (Payment,Popular-Item,top-balance)
4. Optimized implementation of Transactions (Stock Level,Order Status)
5. Main Driver code for Cassandra and Report Writing

1 Introduction

Distributed databases have become increasingly essential in modern data management scenarios, especially with the rise of cloud computing and the need for scalable and highly available data solutions. By distributing data across multiple physical locations or nodes, these databases ensure improved fault tolerance and data redundancy, minimizing the risk of data loss and enhancing data availability. Moreover, they enable the efficient management of large datasets, facilitating faster data access and processing by distributing the computational workload across multiple nodes.

The objective of this project is to acquire practical experience with using distributed database systems for application development. The project implementation is to establish a distributed database system across a cluster of five nodes, utilizing both SQL and NoSQL databases through the implementation of the Citus for relational data and the Cassandra for non-relational data.

Relational databases are a type of database system that organizes data into tables with rows and columns, where each row represents a record, and each column represents a specific attribute or field. They use Structured Query Language (SQL) for managing and querying data, ensuring data integrity and consistency. Citus is an extension to PostgreSQL that transforms it into a distributed database, making it capable of handling large volumes of data across multiple nodes. It provides a horizontally scalable and highly available SQL database that can distribute data and queries across a cluster of machines.

NoSQL databases have emerged as powerful alternatives to traditional relational databases, offering flexible and scalable solutions for managing and processing large volumes of unstructured or semi-structured data. One of the prominent representatives of the NoSQL database category is Apache Cassandra, renowned for its ability to handle extensive data sets distributed across multiple nodes while ensuring high availability and fault tolerance. Built to address the shortcomings of relational databases in handling vast amounts of data, NoSQL databases like Cassandra employ distributed architecture and horizontal scaling, making them well-suited for modern applications that demand real-time data processing, high throughput, and low-latency performance.

The project involves two distinct parts using Citus and Apache Cassandra, each focusing on leveraging the database system's features for efficient transaction processing. The tasks for each part include designing a data model to support the application's workload, implementing specific transaction functions, installing and configuring the database system on a cluster of five servers, and measuring performance using 20 clients concurrently executing transactions.

1. The first task involves designing a comprehensive data model tailored to handle the application's transactional demands, ensuring effective management and processing of data within the database system.

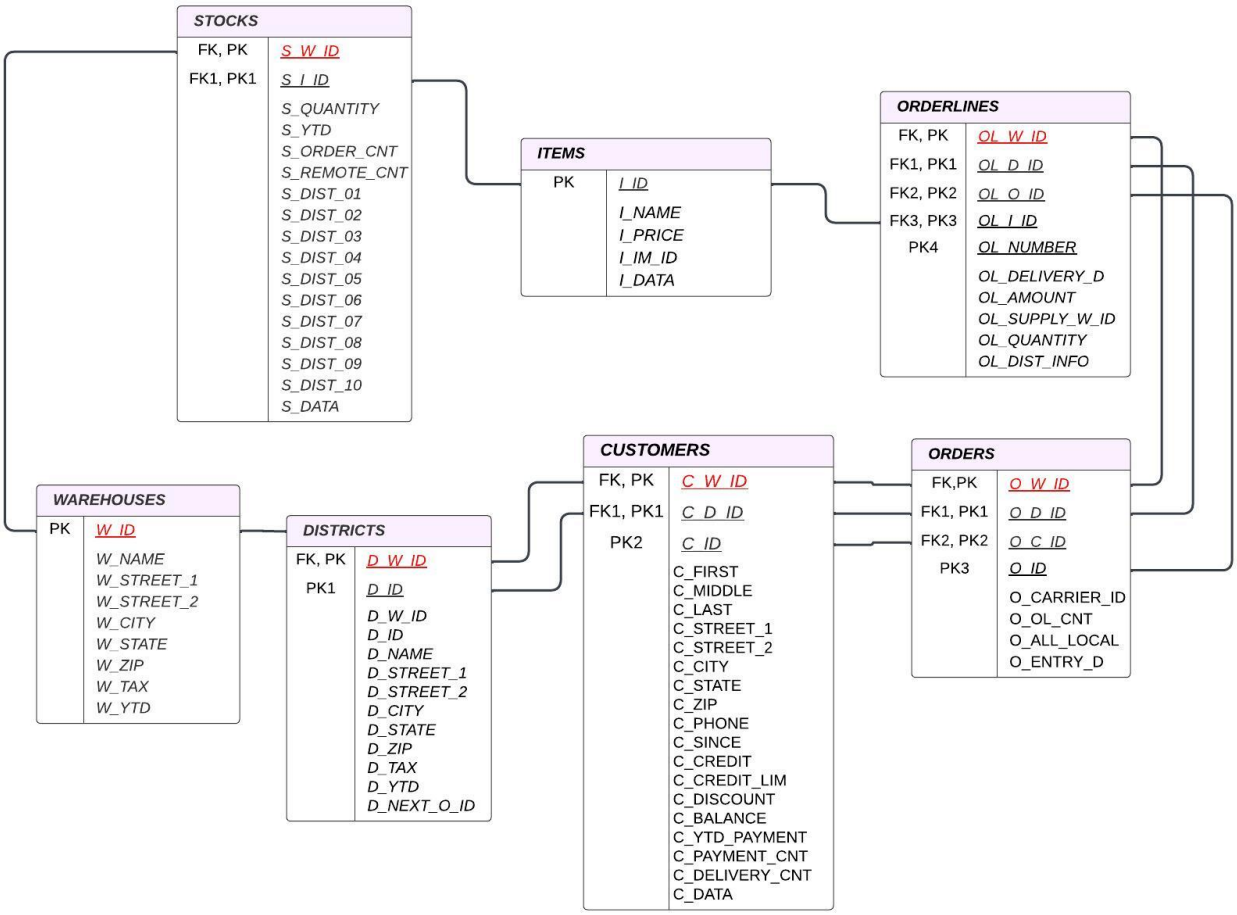
2. Then we focus on the installation and configuration of the chosen database system on a five-server cluster, ensuring proper partitioning and replication to ensure data consistency and availability across the servers.

3. Next we implement the transactions, write scripts to create tables, data insertion and a main driver program for simulating client transactions. The transactions are processed sequentially with detailed performance metrics recorded for analysis.

4. At last the Performance measurement tests are conducted using specified transaction files, with 20 clients executing transactions concurrently to evaluate various performance metrics and assess the database's state at the project's conclusion.

2 CITUS

Citus enables users to develop multi-tenant applications as if they were connecting to a single PostgreSQL database. In reality, this database is a horizontally scalable cluster of machines. Users can make minimal changes to their client code and still fully utilize PostgreSQL's comprehensive SQL features. With Citus, it is possible to maintain the relational database model while scaling it effectively. Citus presents itself to applications as a unified PostgreSQL database, while internally it routes queries to a customizable set of physical servers (nodes) that execute requests concurrently.



2.1 Description of the data models with justifications for the design decisions

In this section, we provide a detailed explanation of our data models and the reasons behind our design decisions. We focus on how we organize data, specifically in terms of data distribution and isolation. These decisions are crucial for our system's efficiency, scalability, and manageability. We discuss the organization of tables such as "customers", "warehouses", "orders", "order_lines", "districts", and "stocks" and the reasoning behind the partitioning of these tables.

2.1.1 Data Organization for Better Efficiency and Isolation

In a multi-tenant system, the organization of data is a key concern. In the context of our wholesale supplier application, which encompasses multiple geographically distributed sales districts and their corresponding warehouses, a structured approach is necessary. Each regional warehouse is responsible for servicing a defined set of ten districts, with each district catering to 3,000 customers. Furthermore, our warehouses diligently manage inventories for a comprehensive catalogue of 100,000 items.

To optimize data distribution, we employ a partitioning strategy based on the warehouse identifier. This methodology ensures an equitable distribution of data across our servers. It is worth noting that the fixed quantities of districts per warehouse, customers per district, and stock records per warehouse contribute to the effectiveness of this approach.

By partitioning tables such as “warehouses”,“districts”,“customers”, “orders”, “order_lines”, and “stocks” based on the “warehouse_id”, we attain the advantage of consolidating all data pertinent to a specific warehouse within a single physical server. Each warehouse and its associated data can be consolidated within a dedicated database instance, ensuring isolation and invisibility from other tenants. This helps keep data local and reduces the need to transfer data between nodes. Additionally, this enables us to leverage partition pruning to exclude partitions that are not relevant to a particular query.

When we need to perform operations on entities linked to a particular warehouse, all the corresponding data will be distributed into shards residing within a single physical node. This setup ensures that operations involving data associated with a specific warehouse are efficiently executed within the confines of one physical server, improving overall performance and resource utilization. By consolidating data on a single node, we optimize resource allocation, making it easier to manage tasks.

We designate the ”items” table as a reference table since we predominantly read data from it, and updates are infrequent. Replicating it across worker nodes, Citus ensures synchronization. This approach improves data retrieval and processing efficiency by enabling direct access to the local copy, reducing network calls and latency. However, the downside is that updates to the ”items” table require propagation to all nodes within the cluster, adding complexity to data maintenance but is crucial for ensuring consistency.

2.2 Overview of Implementation of the Transaction Functions

This section outlines the approach used to manage various transactions.

2.2.1 New Order Transaction

In our implementation of the new order transaction, we adhered to the processing steps outlined in the project guidelines. Despite this transaction accounting for forty percent of the total system workload, the operations carried out are straightforward and uncomplicated. It's important to highlight that the SELECT and UPDATE operations are executed on tables using their primary keys. Therefore, no additional indexes were needed, as Postgres automatically generates indexes for the primary keys of the tables.

2.2.2 Payment Transaction

In our implementation of the payment transaction, we meticulously adhered to the procedural guidelines set forth in the project specifications. The necessity for additional indexes within this transaction was obviated, given that all UPDATE operations were executed on tables employing their respective primary keys. It is noteworthy that Postgres automatically generates indexes for the primary keys of the tables.

An additional strategic advantage was the incorporation of the RETURNING command as an integral component of our UPDATE queries, facilitating the retrieval of essential fields from each table. In the context of a distributed system, the utilization of "RETURNING" results in the direct transmission of updated data to the client within the same network round-trip as the update statement. This strategic choice significantly diminishes the imperative for supplementary read operations, thereby curtailing network traffic and associated latency.

2.2.3 Delivery Transaction

In our implementation of the delivery transaction, we meticulously adhered to the procedural guidelines set forth in the project specifications. The necessity for additional indexes within this transaction was obviated, given that all UPDATE operations were executed on tables employing their respective primary keys or subsets of the primary keys.

2.2.4 Order Status Transaction

The code utilizes SELECT statements to extract specific customer details based on given customer identifiers from the "customers" table. To determine the most recent order for this customer, it sorts the customer's orders present in the "orders" table by their entry date (o_entry_d) in descending order and selects the top row. This method proves to be efficient since it retrieves just one record from the database.

In contrast, an alternative approach that involves fetching all the customer's orders and then identifying the most recent one would result in greater overhead in a distributed setup. This is because numerous rows would need to be transferred from the node containing the shards related to this customer to the shard where the transaction is executed.

Once the most recent transaction is identified, the code proceeds to identify all the associated items from that transaction and extract the necessary fields, which are then printed.

2.2.5 Stock Level Transaction

By using dynamic query to fetch the n number of last orders, we ensure that we always work with the most up-to-date and consistent data across all nodes in the Citus cluster. This approach enhances the performance, reliability, and scalability of our Citus-based applications. The subsequent code retrieves the stock of items that meet the threshold condition. We primarily use "d_id", "w_id" as argument for the primary keys of the tables, where tables districts, order_lines and stocks are all partitioned on "w_id". It follows a straightforward and organized structure, making it easy to understand and maintain. This approach is efficient and minimizes the need for extensive data processing within the application itself.

2.2.6 Popular-Item Transaction

In the interest of maintaining data integrity, We utilize the same dynamic query approach that was previously utilized for the Stock Item Transaction to retrieve recent orders. To optimize and simplify the query, we initially retrieve the maximum quantity of the item within the order by using JOIN operation on "items" and "order_lines". We query this directly by using the parameters "w_id", "d_id" passed from the transaction file and previously fetched "order_id" as argument for the primary keys . From the result, we find the highest quantity, this method eliminates the need for subqueries. Further, We use this quantity to identify popular items. Simultaneously, we create a dictionary containing all unique "item ID" and their corresponding "item" name from the fetched orders. This dictionary allows us to calculate the percentage of popular item occurrences among the orders with ease.

2.2.7 Top-Balance Transaction

We have implemented indexing on the "balance" column in the customer table and the code strategically employs SQL SELECT statements and JOIN operations to streamline data retrieval from the tables "customers", "districts", and "warehouses". This approach significantly reduces the number of database queries required, thereby alleviating the database server's load and enhancing query performance.

One notable advantage of this approach is its ability to substantially reduce the total number of necessary database queries. Instead of executing multiple queries to obtain data from different tables separately, the code efficiently consolidates these operations into a single comprehensive query.

This consolidation results in fewer interactions with the database server, leading to a more efficient and streamlined data retrieval process. This selective data retrieval is particularly advantageous as we are dealing with large datasets, as it conserves network bandwidth and reduces response time. Furthermore, when identifying customers with the highest balances, only the initial ten rows are selected. This deliberate choice minimizes the number of rows involved in the subsequent join operation, resulting in a substantial reduction in computational overhead and improved query efficiency.

2.2.8 Related-Customer Transaction

We have enhanced the efficiency of our data retrieval process by implementing indexing on the "item ID" column in the "order-lines" table.

Further, in order to optimize the process of identifying common items between the primary customer and other customers, we employ a well-thought-out strategy. We start by utilizing the customer identifier to directly retrieve all the orders by that customer from the order table. These order IDs are organized into a dictionary, linking each order ID with its corresponding item IDs. This eliminates the need for repetitive queries of the main customer's order IDs for comparison. To retrieve the items associated with each customer's orders, we need to perform a join between the customer orders and order lines tables. This process is made more efficient because the tables are partitioned based on the warehouse ID. As a result, the necessary data rows are co-located on the same physical node, allowing us to carry out localized join operations, thereby eliminating the need for network communication overhead.

Subsequently, we create a set of unique item IDs ordered by that customer. Using this set, we then select all order IDs, ordered items and their corresponding warehouse and district IDs from the order lines table. By narrowing down our search to only the specific item IDs in the set, we significantly reduce the number of unnecessary searches, thus optimizing the entire process.

The results of this query are systematically organized into a dictionary again, where each key contains order ID,warehouse ID and district ID and corresponding item IDs as its values. We also ensured that each order contains at least two items, orders that doesn't satisfy that condition are removed from the dictionary to further streamline the comparisons.

With our refined data, we proceed to identify common items between the main customer and other customers. Orders with a minimum of two common items with the main customer are stored in a set. In the final step of our process, we split the keys within the set to query the customer table, using the warehouse ID and district ID. This allows us to retrieve the relevant customer information and print our results.

This approach reduces the number of comparisons and checks required, enhancing query efficiency by concentrating on the most probable scenarios. his ensures that the results are highly relevant and accurate This strategy not only improves the efficiency of the process but also ensures that the results are highly relevant and accurate.

2.3 Tests

We've developed individual tests for all transactions, allowing us to efficiently assess the accuracy of each transaction implementation across various scenarios.

2.4 Configurations Used

We've automated the setup process for Citus, encompassing installation, table creation, data insertion, and configuration management. Our transaction function files are hosted on a shared drive accessible to the team. Configuration files for PostgreSQL are stored in the /temp directory, and we ensure node-specific configuration changes stay isolated to their respective /temp nodes, preventing cross-node effects.
The process unfolds as follows:

- We employ an installation script to set up Citus on the shared drive.
- The database initialization step is executed across the selected nodes.
- Tables are created and populated with the provided data.
- Clients are executed to produce output, which is subsequently stored in a designated directory.
- Performance metrics are computed, facilitating comprehensive reporting.

In order to optimize performance for read-heavy transactions, we created indexes on the "customers" and "order_lines" tables. To elaborate, a specific index was introduced on the "c_balance" column within the "customers" table. This strategic indexing significantly streamlines the execution of transactions such as "top_balance" as it enables efficient searches based on the "balance" column. Likewise, within the "order_lines" table, an index was thoughtfully established on the "ol_o_id" column. This index enhances query response times for operations involving the "order_lines" table, particularly in the context of the "related_customers_transaction".

2.5 Benchmarking Results

2.5.1 Transaction Workload

The tables provided below represent the allocation of workloads derived from actual transaction files that were utilized in benchmarking. The distribution is categorized by both transaction type and client.

client	server	N	P	D	O	S	I	T	R	Total
0	0	5037	1441	984	168	101	139	199	63	8132
1	1	4727	18690	1183	122	141	131	184	61	25239
2	2	10702	2241	2451	136	129	127	183	69	16038
3	3	6565	1600	1465	113	106	123	202	50	10224
4	4	4870	1627	5959	120	123	126	190	67	13082
5	0	2825	806	1013	128	125	120	180	56	5253
6	1	9273	904	2312	111	133	134	199	55	13121
7	2	5263	932	4269	108	134	111	187	66	11070
8	3	2998	1198	1775	123	107	127	213	64	6605
9	4	7922	1543	767	123	126	120	195	66	10862
10	0	4032	33510	4016	120	129	143	192	66	42208
11	1	5142	39960	716	135	122	113	199	50	46437
12	2	4773	1450	1905	126	154	130	164	60	8762
13	3	6671	1050	5997	126	128	100	173	62	14307
14	4	3397	1284	1753	121	139	125	196	70	7085
15	0	8013	1435	1564	126	124	123	183	75	11643
16	1	8599	60202	1214	134	121	137	204	71	70682
17	2	4440	61416	849	136	134	131	207	59	67372
18	3	7293	2165	969	127	132	103	180	65	11034
19	4	4177	27796	1198	138	118	120	191	67	33805

Table 1: Transaction workload distribution by client

client	N	P	D	O	S	I	T	R
0	60.45	17.45	59.88	4.92	7.70	90.45	28.57	84.50
1	62.06	16.82	58.84	5.44	8.31	98.08	40.63	83.39
2	130.67	32.56	185.10	11.75	19.73	127.77	48.72	323.43
3	183.44	62.86	239.34	13.65	19.75	120.02	72.11	167.85
5	60.53	17.87	65.07	5.34	8.35	89.26	26.61	46.03
6	142.72	44.61	189.64	11.63	17.74	129.52	46.91	321.12
8	79.00	22.63	67.58	7.03	12.03	114.69	44.37	45.89
10	107.66	33.60	107.72	9.57	17.02	128.57	35.87	139.68
11	105.47	32.68	147.55	7.99	13.91	115.09	42.85	213.10
12	154.44	57.34	219.37	14.00	18.58	129.29	51.29	145.38
13	163.18	67.57	106.44	11.30	18.88	113.60	63.33	200.68
15	151.57	55.59	191.92	11.20	18.49	128.88	40.64	328.60
16	62.80	29.75	100.46	6.99	10.73	109.27	34.90	200.67
17	64.92	29.25	120.07	6.65	10.97	113.38	36.07	139.65
18	179.97	61.10	240.81	15.52	22.05	125.14	73.26	275.80
19	137.13	31.26	225.05	10.84	16.38	136.35	48.09	133.40

Table 2: Average Time taken by Individual Transactions

2.5.2 Benchmarking Results

Following the approach outlined in Citus documentation, we conducted performance testing on a five-server cluster. The results from these tests are compiled in the tables

client	A	B	C	D	E	F
0.txt	8132	445.17	18.27	54.74	132.32	361.22
1.txt	25238	432.51	58.35	17.13	53.65	257.78
2.txt	16038	1988.12	8.07	123.96	319.16	482.15
3.txt	10224	1647.78	6.2	161.16	362.93	510.02
4.txt	13083	438.39	29.84	33.5	77.64	254.81
5.txt	5253	155.01	33.89	29.5	74.14	264.23
6.txt	13121	1828.57	7.18	139.35	326.32	473.38
7.txt	11069	1554.62	7.12	140.32	329.48	462.71
8.txt	6605	428.73	15.41	64.9	186.28	290.72
9.txt	10862	610.47	17.79	56.2	136.66	381.22
10.txt	42208	1991.28	21.2	47.17	233.01	400.02
11.txt	46437	1959.43	23.7	42.19	212.49	385.44
12.txt	8762	1261.6	6.95	143.97	348.23	521.24
13.txt	14307	1769.79	8.08	123.69	325.99	470.49
14.txt	7085	1022.72	6.93	144.34	345.26	490.49
15.txt	11643	1582.26	7.36	135.89	320.39	459.89
16.txt	70682	2480.42	28.5	35.08	168.83	369.8
17.txt	67372	2217.04	30.39	32.9	172.43	370.82
18.txt	11034	1697.65	6.5	153.85	345.26	472
19.txt	33804	1717.97	19.68	50.78	241.52	372.85

Table 3: Performance Measurements

A: Number of executed transactions

B: Total transaction execution time (in seconds)

C: Transaction throughput (number of executed transactions per second)

D: Average transaction latency (in ms)

E: 95th percentile transaction latency (in ms)

F: 99th percentile transaction latency (in ms)

Minimum Throughput	Maximum Throughput	Average Throughput
6.2	58.35	18.0705

Table 4: Throughput summary statistics (in transactions per second)

Statistics	Value
Sum of W_YTD from Warehouse	656728910.16
Sum of D_YTD from Warehouse	656728910.16
Sum of D_NEXT_O_ID from District	416819
Sum of C_BALANCE from Customer	5500261987.99
Sum of C_YTD_PAYMENT from Customer	656728910.1600018
Sum of C_PAYMENT_CNT from Customer	561250
Sum of C_DELIVERY_CNT from Customer	205023
Max of O_ID from Order	5164
Sum of O_OL_CNT from Order	5206067
Sum of OL_AMOUNT from Order-Line	6285703246
Sum of OL_QUANTITY from Order-Line	26772507
Sum of S_QUANTITY from Stock	55572304
Sum of S_YTD from Stock	8023227
Sum of S_ORDER_CNT from Stock	1456211
Sum of S_REMOTE_CNT from Stock	14573

Table 5: Final database state

2.5.3 Discussion of Bench-marking Results

The benchmarking results presented in Tables 1, 2, and 3 provide crucial insights into the overall system performance. It provides a comprehensive view of the system's performance, emphasizing transaction counts, response times, and client-server distribution data.

Table 1 exhibits the system's adaptability to varying workloads, showcased by the even distribution of transactions across servers.

Observations from Table 2 highlight the system's efficiency in handling diverse transaction types. Notably, the consistent performance underscores adaptability to dynamic demands.

Upon observing the date presented in both table 1 and table 3, it becomes evident that a partial relationship exists between transactional completion times and transaction volume. Notably, in the case of "16.txt", containing the highest number, 70,682 transactions, the processing duration extended to 2,480.42 seconds. However, though transaction volume is a factor impacting processing time, the complexity of individual transactions, play a concurrent and non-negligible role in influencing the overall time required for processing.

When comparing two transaction files, "11.txt" and "10.txt", despite 11.txt having more transactions (46437) compared to "10.txt" (42208), a closer look at individual transactions reveals an interesting pattern. In "10.txt", certain transactions, like D, I, and R, not only have a higher count but also take more time on average.

It is noteworthy that transactions D (125.67), I (119.37), and R (168.79) are the top three transactions that consume the most time.

This suggests that these transactions significantly contribute to the overall longer execution time of “10.txt”, making them the primary reason for the performance difference between the two files.

Similarly, in file 1.txt, which runs efficiently in a short time, 432.51 seconds despite a workload of 25238, the main factor could be because the main contributor to the workload is transaction type P, 18690. As The average time for P is 33.53 seconds, it possibly plays a crucial role in making the file execute faster.

On the flip side, file “5.txt”, with the shortest execution time, benefits from having the lowest workload, 5253 and lower average times for transactions. This combination results in a quicker execution time for the file.

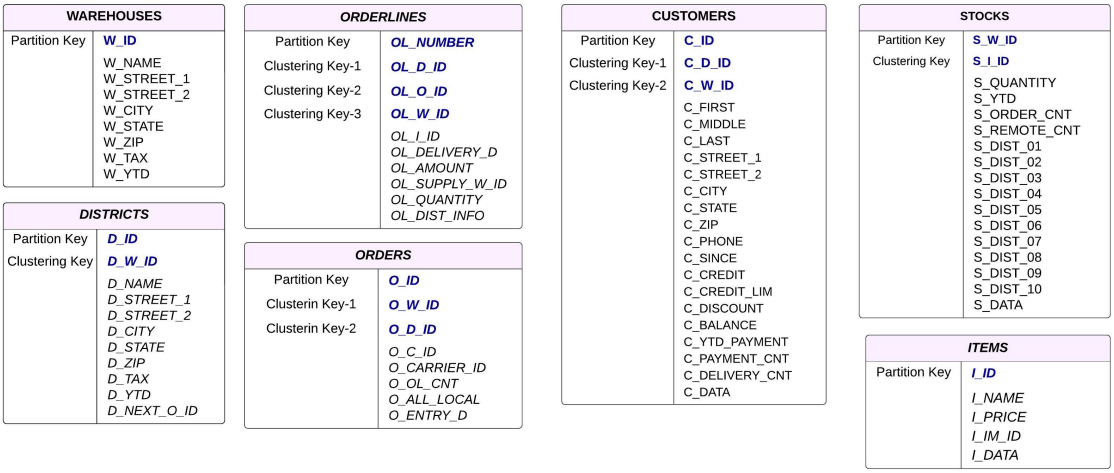
In conclusion, it can be understood that in the case of a server having a higher count of complex transactions, there is a likelihood of increased time. Nevertheless, a high workload would play a major role for the time taken, regardless of the type of transaction requests it might receive and the vice versa also applies.

These benchmarking results offer a nuanced understanding of the system’s strengths and areas for improvement. The findings serve as a precise guide for enhancing system performance, ensuring optimal resource utilization, and addressing specific inefficiencies.

3 Cassandra

3.1 Initial Implementation of Cassandra

3.1.1 Basic Data Modelling Design



The initial approach was a crucial component of the project's exploratory phase, serving as the foundation for understanding the intricacies of the data and the specific requirements of the project. The basic approach was, we implemented the project architecture as provided in the project description. We did not change or design a new architecture, but rather implemented the same table design, but we chose the primary keys and clustering keys depending on the table. By thoroughly assessing the initial implementation, we identified the potential for further optimization on table design, which led to the subsequent reevaluation and redesign of the table structures. This involved creating tailored tables for specific transaction types, combined tables for faster data retrieval. Moreover, by using the initial approach as a baseline for performance evaluation, we gained valuable insights. The changes and improvements that followed were a direct result of what we learned from our initial implementation.

3.1.2 Implementation

1. Manual join operations within each query were needed due to the absence of pre-defined joint tables, leading to slower query execution.

2. Lack of an integrated architecture resulted in slower data retrieval and processing due to multiple table lookups and data retrieval steps.

3. Complex query structures were necessary to organize data from different tables, significantly increasing query processing time. Due to the absence of pre-defined relationships and integrated architecture, the queries required intricate operations such as subqueries, multiple joins, and complex aggregation functions to retrieve the desired data.

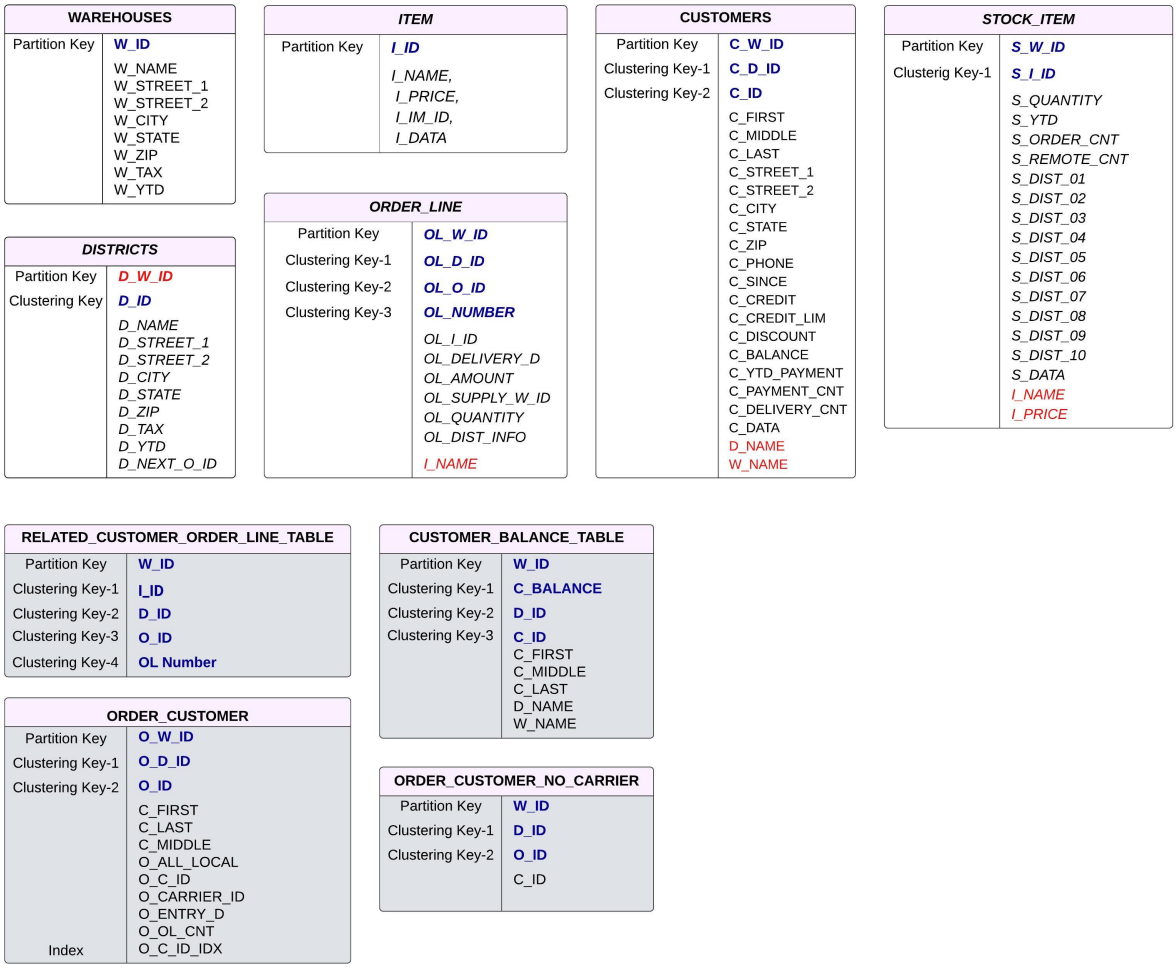
4. Additional filtering and sorting steps were required due to the lack of streamlined data relationships, causing delays in data retrieval and analysis. We had initially used "ALLOW FILTERING" for filtering operations on non-indexed columns during query execution.

5. Absence of predefined relationships and indexes between tables increased data scanning and searching, resulting in slower query performance and higher resource utilization.

3.1.3 Observations on results

The system's query processing times were extended due to the need to traverse multiple tables, which involved performing numerous lookups and retrievals across distributed nodes. This complexity was a result of the absence of optimized pathways for quickly locating data, forcing exhaustive searches and thus leading to increased retrieval times. Among all the transactions, Delivery Transactions and Related Customer Transactions were notably the most time-consuming. In light of this, a detailed examination was undertaken. We realized that the inefficiency stemmed from the way we had organized and partitioned our data. Each query had to go through many data partitions, exacerbating the latency and resource consumption issues. By reevaluating and subsequently modifying the data organization and partitioning strategy, we were able to streamline the data traversal process. These optimizations to our data model significantly reduced the latency of transactions and led to much faster processing times, demonstrating a successful optimization of our system's performance.

3.2 Optimized Data Model



3.3 Description of the data models with justifications for the design decisions

Our journey in data modelling with Cassandra led us to adopt a query-driven approach. Unlike traditional relational databases, Cassandra's non-relational nature meant that standard joins were not supported. To address this limitation, we focused on the specific transactions we needed to implement and tailored our table designs accordingly. In this query-centric approach, we leveraged data denormalization techniques to split tables, merge data, create duplicate columns, and structure the data in a way that would allow for efficient transaction execution. These strategies were instrumental in eliminating the need for complex join operations, which are commonplace in traditional relational databases.

The result of this approach is the set of tables depicted in the ER diagram. Each table was designed with a clear understanding of the queries it needed to support, making data retrieval and manipulation efficient and tailored to our specific use cases.

3.3.1 Orderline Table

The inclusion of the 'i_name' (item name) column in the 'order_line' table enables the direct retrieval of the name of the item associated with each order line. This simplifies the process of identifying popular items within specific orders, as well as calculating the percentage of orders that contain each popular item. The availability of the item name within the 'order_line' table eliminates the need for additional table joins or complex operations, facilitating a more efficient query.

3.3.2 Customers Table

This architecture simplifies the top balance customer transaction by consolidating all necessary information in one table. Without the need for multiple tables joins, the process becomes more efficient, eliminating resource-intensive tasks. Previously, querying three separate tables—warehouses, districts, and customers—was required, whereas now, only one table needs to be queried.

3.3.3 Stock_item Table

This design, tailored for the New Order transaction, eases the process of updating and managing stock information for a specific item. By integrating item name and price into the stock table, data maintenance, querying, and modifications become more straightforward and efficient. Storing item-related data, such as the item name and price, within the same table as stock information ensures data consistency.

3.3.4 Related Customer Orderline Table

Consider the scenario where we need to determine the customers related to a specific customer based on common items:

Without the 'related_customer_order_line_table': The process would involve querying the 'Order' table to find orders placed by each customer using O_C_ID. Next, we would need to query the 'Order Line' table to retrieve the items associated with each order using the O_ID in Order Line table. Subsequently, complex join operations would be necessary to identify orders that have at least two common items, which could be computationally expensive.

With the 'related_customer_order_line_table': The Related Customer Order Line Table was meticulously designed with performance in mind. It employs warehouse_id as the partition key and item_id as the first clustering key. This strategic choice of table structure enables efficient data retrieval, particularly for queries involving the item_id column.

By placing item_id as the first clustering column, the table is optimized for queries utilizing 'IN' conditions on the item_id. This deliberate design choice significantly accelerates the data retrieval process, enhancing the overall efficiency and responsiveness of the system. This table is updated whenever a new entry is made to the order_line table.

3.3.5 Customer Balance Table

The Customer Balance Table simplifies data retrieval by consolidating essential attributes from the Warehouses, Customers, and Districts tables. This design minimizes the need for complex joins and streamlines query processing. The Customer Balance Table is structured with a partition key based on Warehouse ID (warehouse_id). It includes multiple clustering keys, namely Customer Balance (c_balance), District ID (district_id), and Customer ID (customer_id).
The c_balance was selected as the first clustering key so that the values in the table could be sorted based on this column after being partitioned. The choice of having c_balance as the first clustering key enables efficient retrieval of customers with the highest balances. This setup optimizes queries, particularly when searching for top customers based on payments. This design allows for efficient data organization and optimized querying, especially when searching for top customers based on balance.

3.3.6 Order Customer Table

The 'order_customer' table brings together information from the 'order' and 'customer' tables, creating a unified table that simplifies data retrieval and analysis for the 'Popular-Item Transaction.

3.3.7 Order_Customer_No_Carrier

The primary objective of the Order_Customer_No_Carrier Table is to efficiently capture all orders that have yet to be assigned a carrier. This table proves to be a cornerstone for optimizing delivery transactions. Instead of scanning through all rows within a partition to identify orders lacking a carrier ID for a specific warehouse, the new table enables precise queries based on the warehouse ID for each district. This streamlined approach allows for the direct retrieval of the minimum order ID associated with unassigned orders.

Once a carrier is successfully assigned to a specific order, the corresponding entry is promptly removed from the Order_Customer_No_Carrier table. This strategic design greatly enhances the efficiency of the system, ensuring that carrier assignments and order management proceed seamlessly, minimizing unnecessary queries, and facilitating timely and accurate deliveries.

3.3.8 Warehouses, Districts, Items

- Warehouses - As per original schema.
- Districts - As per original schema.
- Items - As per original schema.

3.4 Overview of Implementation of the Transaction Functions

3.4.1 New Order Transaction

We designed our data model such that we were able to perform UPDATE and SELECT queries based on query constraints of Cassandra. We have religiously followed the instructions mentioned for the transaction and made changes to optimize the transaction and also cater other transactions. In this transaction when we query a table we get all the needful data from that table all at once that we need in the transaction. Example, when getting the 'D_NEXT_O_ID' from the Districts, we also fetch the 'D_TAX' that is needed later in the transaction. We have merged needed information (item name and item price) from item table into Stock_Item table. This optimization enables us to avoid querying item table separately. These are some enhancements we made in our data model and transaction that help us run this transaction efficiently.

3.4.2 Payment Transaction

This is a simple, straight forward transaction. We religiously follow the instructions mentioned for the transaction and make changes to optimize the transaction and also cater the other transactions. In this transaction as well, when we query a table we get all the needful data from that table all at once that we need in the transaction rather than querying the table again.

3.4.3 Delivery Transaction

This is a transaction that required optimization. Cassandra does not allow checking for null values, as it actually does not store it anywhere. So initially we decided to put carrier ID as -1 instead of null in order_customer table and created an index on carrier ID that would help us to query for minimum order ID from the order_customer table. But this was also taking a lot of time to execute. So we decided to create a new table Order_Customer_No_Carrier that additionally will store the orders that have null carrier ID. So we can query this table based on warehouse ID, district ID and order by order ID in ascending. This helped us optimize the query. So each time a carrier ID set in order_customer table, we delete an entry from the table and also create a new entry each time a new order is created. This is the major optimization we made in this transaction. Rest, we religiously followed the instructions mentioned for the transaction and modified the transaction, keeping in mind other transactions as well.

3.4.4 Order-Status Transaction

The code fetches and displays customer details, retrieves the customer's last order information including order number, entry date, and carrier identifier, and retrieves and displays specific details for each item in the customer's last order. We have used MAX(O_ID) to retrieve the highest order number associated with the provided customer identifiers (C_W_ID, C_D_ID, C_ID) from the order_customer table. This is stored in the variable last_order_oid. The.ONE( ) method is employed to obtain only the last order number from the query result, which is utilized in the subsequent queries to fetch additional details related to the last order.

3.4.5 Stock Item Transaction

We have a function named check_stock_threshold. It's straightforward and uses three simple queries to get the information we need. Since it only looks up data using the main keys in our database, we didn't have to create extra indexes or make any changes to how our data is set up. The function gives us a number showing how many items are low in stock from the latest orders for a specific warehouse area.

3.4.6 Popular Item Transaction

To obtain insights into the most recent 'L' orders within a specific district, we initiated the process by retrieving these orders from the 'Orders' table. To enhance the comprehensiveness of our data, we extended our focus to the customers associated with these orders, capturing essential details such as "c_first", "c_middle" and "c_last". These customer details were thoughtfully integrated into the "Order_Customer" table, thereby simplifying future queries for "o_entry_day" and "o_id" columns.

Each of the 'L' orders was subject to further examination as we delved into the "Order_Line" table to extract item-specific details. From these items the popular items of the order were determined by comparing the quantities of each order. Our approach included augmenting this table with the "item_name" information, streamlining the process of directly accessing and displaying these item names. Our analysis extended to quantifying the frequency of each item's occurrence within these orders, ultimately allowing us to showcase the corresponding percentages.

This meticulous process of data aggregation and analysis equipped us in determining the popular items.

3.4.7 Top Balance Customers Transaction

In our quest to identify the top customers based on their "c_balance" field, we encountered a challenge within the Cassandra database. The "c_balance" field, not being a part of the partition or clustering key, posed a limitation on our ability to execute direct queries on this criterion. To circumvent this limitation, we engineered a novel solution by introducing a new table. This table incorporated the "c_balance" field within the clustering column, enabling us to effectively sort and order the values in

descending order. As a result, we could efficiently pinpoint customers with the highest balances. The second table was updated as a part of all transactions that update the customer's balance.

The "w_id" was thoughtfully selected as the partition key for this table, facilitating data organization. Our process then entailed selecting the top ten customers from each warehouse, with an initial list created to store these customers. Subsequently, the list was sorted in descending order, ranking the customers based on their balance, and unveiling the ten customers with the highest balances.

This innovative approach didn't merely enhance our ability to query the "c_balance" field but also enriched our table by incorporating essential information such as district names and warehouse names linked to each customer. As a result, we were well-equipped to display the necessary fields for each of the top ten customers, thereby achieving our goal of identifying and presenting the most valuable customers with precision and efficiency.

3.4.8 Related customers Transaction

The transaction retrieved a comprehensive list of all orders associated with a specified customer from the "Order_Customer" table. For each of these orders, the corresponding items were extracted from the "Order_Line" table, and this information was meticulously organized in a hash map. Simultaneously, a set was maintained, encompassing all the unique item IDs found across these orders.

To identify orders that deviated from the same warehouse as the customer and contained items of interest, the transaction introduced a key challenge. The "Order_Line_Item_ID" column, a pivotal criterion for this query, was not part of the partition key or clustering key in the "Order_Line" table. To ensure swift data analysis, a specialized table known as the "Related_Customer_Order_Line_Table" was introduced. In this table, 'W_ID' was designated as the partition key, while the clustering key incorporated "I_ID", "D_ID", "O_ID" and "OL_NUMBER". This design allowed for direct querying based on the "I_ID" column.

Orders containing items relevant to the customer were identified through queries on the "I_ID" column, using the customer's list of items as reference. This information was further utilized to construct a hash map linking the order identifier (comprising "W_ID", "D_ID" and "O_ID") to the list of items for each order.

The final phase of the transaction involved a comprehensive analysis, traversing both hash maps to identify orders sharing more than two common items. As a result, the customer's identity was established based on the corresponding order. This intricate process of data extraction and analysis demonstrated the transaction's effectiveness in efficiently linking customers to the related customers.

3.5 Tests

We have developed individual tests for all transactions, allowing us to efficiently assess the accuracy of each transaction implementation across various scenarios.

3.6 Configuration Used

We set up Apache Cassandra version 4.1.3 and Java Development Kit (JDK) 11 on all our servers. All the Cassandra files are placed in a common shared directory that the whole team can access. Each server has its own temporary directory (named /temp) where we make server-specific changes to the configuration files. This way, the settings for one server won't affect the others.[1].
We created a Keyspace in Cassandra called a "teambgoing". This keyspace is designed to store copies of data in all servers for safety and reliability. We made sure this keyspace was set up on all the servers. Inside the keyspace, we created tables to hold transaction data. We then filled these tables with the transaction data on all the servers. We ran client applications to process transactions. These applications then generated performance data, which we saved into separate CSV files for analysis.

3.7 Benchmarking Results

3.7.1 Transaction Workload

The tables provided below represent the allocation of workloads derived from actual transaction files that were utilized in benchmarking. The distribution is categorized by both transaction type and client.

3.7.2 Performance Testing

Following the approach outlined in Cassandra documentation, we conducted performance testing on a five-server cluster. The results from these tests are compiled in the tables
Note: Due to the modified slurm configuration, we were restricted to using only 32G of memory for script execution.

client	A	B	C	D	E	F
0.txt	8132	5452.73	1.49	670.52	2198.42	2455.67
1.txt	25239	7746.98	3.26	306.93	1061.76	2659.61
2.txt	16038	9553.76	1.68	595.68	2429.68	2960.36
3.txt	10224	7413.71	1.38	725.11	2379.67	2890.43
4.txt	13083	6567.17	1.99	501.95	1899.73	2336.75
5.txt	5253	3968.81	1.32	755.52	2322.3	2547.58
6.txt	13121	9061.86	1.45	690.63	2489.43	2852.07
7.txt	11070	7536.59	1.47	680.8	2510.03	2943.5
8.txt	6605	6091.13	1.08	922.19	2589.24	2998.08
9.txt	10862	6502.98	1.67	598.68	1912.87	2354.32
10.txt	42208	7811.57	5.4	185.06	667.56	1990.49
11.txt	46437	8320.39	5.58	179.16	721.6	1131.5
12.txt	8762	6585.53	1.33	751.59	2419.23	2801.81
13.txt	14307	8090.58	1.77	565.49	1665.33	2579.39
14.txt	7085	5695.16	1.24	803.81	2306.49	2527.77
15.txt	11643	6587.87	1.77	565.81	1571.87	2373.86
16.txt	70682	10497.67	6.73	148.51	346.8	1056.22
17.txt	67372	9287.68	7.25	137.84	319.84	1128.88
18.txt	11034	7496.23	1.47	679.36	2357.41	2997.76
19.txt	33805	7102.43	4.76	210.08	819.63	2270.18

Table 6: Performance Measurements

The columns and what they represent

A: Number of executed transactions

B: Total transaction execution time (in seconds)

C: Transaction throughput (number of executed transactions per second)

D: Average transaction latency (in ms)

E: 95th percentile transaction latency (in ms)

F: 99th percentile transaction latency (in ms)

Minimum Throughput	Maximum Throughput	Average Throughput
1.08	7.25	2.7045

Table 7: Throughput summary statistics (in transactions per second)

Statistics	Value
Sum of W_YTD from Warehouse	588584516.58
Sum of D_YTD from District	655651880.87
Sum of D_NEXT_O_ID from District	416819
Sum of C_BALANCE from Customer	5358326854.63
Sum of C_YTD_PAYMENT from Customer	656728128.0
Sum of C_PAYMENT_CNT from Customer	561248
Sum of C_DELIVERY_CNT from Customer	203397
Max of O_ID from Order	5165
Sum of O_OL_CNT from Order	5206067
Sum of OL_AMOUNT from Order-Line	6285703245.69
Sum of OL_QUANTITY from Order-Line	26772507
Sum of S_QUANTITY from Stock	55572316
Sum of S_YTD from Stock	8023014
Sum of S_ORDER_CNT from Stock	1456170
Sum of S_REMOTE_CNT from Stock	14573

Table 8: Final database state

3.7.3 Discussion of Benchmarking Results

In the analysis of the Cassandra benchmarking we analysed some significant results. As the number of transactions increases, generally the throughput, that is number of transaction per second is also increasing. It is evident from execution of transactions in 16.txt that has 70682 transactions with 6.73 throughput and 17.txt. and 17.txt that has 67372 transactions with 7.25 throughput. It kind of indicates that the longer-time taking transactions are offset by the larger number of fast-running transactions such as Payment transaction

Another thing we noticed that 1.txt with 25239 number of transactions took lesser time to execute as compared to 2.txt with 16038 number of transactions. This is because 2.txt has significantly more number of New Order transactions that is a longer-running transaction as compared to transactions in 1.txt. And 1.txt has significantly more number of Payment transactions that is a shorter-running transaction as compared to transactions in 2.txt.

If we see the files 13.txt and 11.txt, number of transactions in 11.txt(46437) are more than 3 times as compared to number of transactions in 13.txt(14307). However, the total time taken to execute all the transactions in almost round about same. This is because 13.txt is dominated by Delivery type transactions that is a longer-running transaction and 11.txt is dominated by Payment type transactions that is a shorter-running transaction.

On the other hand, the files '10.txt' and '11.txt' also display a significant number of transactions. However, they exhibit somewhat elevated latencies when compared to '16.txt' and '17.txt'. This suggests that while they are capable of handling a large workload, each transaction consumed a slightly longer duration to complete.

The files '0.txt', '1.txt', and '2.txt' are characterized by a lesser workload, as reflected in the smaller number of executed transactions. Correspondingly, these files demonstrate a reduced pace in processing transactions, as seen in their lower throughput values.

The following table compares the performance of the initial and optimized implementations of Cassandra for one transaction of each type. There is a bit of increase in New-Order transaction as we are inserting into newly created tables as well. But there was significant performance in the other transactions.

Transaction Type	Initial Implementation (in seconds)	Optimized Implementation (in seconds)
New-Order	0.2	0.77
Payment	1.2	0.035
Delivery	50	0.6322
Order-Status	0.2	0.0199
Stock-Level	0.6	0.23
Popular-Item	2	0.089
Top-Balance	7	0.11
Related-Customer	55	0.08

Table 9: Comparison of 2 implementations of Cassandra

Note : The above table represents the time taken in SECONDS for a single transaction to run with 1 set of inputs in different cassandra models

4 Challenges & Lesson Learnt

4.1 General

In the initial phases of our project, we encountered some challenges. One of the primary issues was figuring out how to set up a database cluster on shared servers without having administrative permissions. This presented a significant hurdle that we needed to address.

4.2 Citus

As we began working with Citus, there was a learning curve in data modelling, as it played a vital part. We aimed to ensure that we took into account the workload and transactions when designing the

model. As we started implementing transactions, there were times when we couldn't access the nodes we needed. This caused delays in development. However, we later devised a solution by replicating the database on a single node. This allowed us to develop and test the transaction code without waiting for node access. Once we regained access, we smoothly implemented the transactions

When we integrated all the transactions into the main driver, we encountered deadlocks, which are situations where processes couldn't proceed. Resolving these deadlocks was a challenging task, but we eventually succeeded.

We also had complications with related customer transaction as initially, it took around 30 minutes to process a single file. To improve this, we did some research and made changes like using indexing and optimizing how we queried tables, reducing the need for complex JOIN operations and subqueries. These changes significantly improved performance.

Working with Citus taught us valuable problem-solving skills and gave us a better understanding of how relational and distributed databases function and interact. This project helped us see things from a relational database perspective and enhanced our knowledge of distributed databases.

4.3 Cassandra

Furthermore when it came to adapting to Cassandra, a NoSQL database, to accommodate the specific requirements of our transaction workload, a critical data modelling challenge emerged. Due to Cassandra's non-support for operations like joins, a fundamental shift was necessitated in our approach to data organization. The tables were deliberately de-normalized, leading to the duplication of various entities across the database.

This restructuring process was far from straightforward. Each transaction-driven modification prompted careful consideration of data replication, aiming to ensure the integrity and consistency of the duplicated records. In essence, whenever an update was applied to a record, corresponding changes were meticulously propagated to its duplicate entries, maintaining data coherence throughout the system.

In our study, we explored the concept of utilizing multiple clustering keys for a given table. An essential takeaway from our investigation was the significance of carefully determining the order in which these clustering keys are defined, as it profoundly influences how data is organized within partitions. Moreover, the chosen order of clustering keys plays a pivotal role in shaping our query capabilities. For instance, if a table's clustering keys are arranged as A followed by B, the table's data will be sorted primarily based on A and, subsequently, on B. Consequently, this configuration restricts our ability to query on B independently without also specifying A in the query.

There are some query constraints we figured out while studying documentation and executing queries. These are important to be put into consideration before we do our data modeling and write queries. Some key constraints we figured were:

- Partition key is must whenever using the where clause to query a table.
- Partition key is must whenever using the where clause to query a table.
- Order by clause can only be used on clustering keys. It cannot be used on the partition key.
- If a table has a secondary index on a column of a table, then order by cannot be used while querying that table.
- We cannot check for null values in queries in Cassandra, as Cassandra does not store null values.

One additional challenge was, the maximum memory that could be allocated to a process was reduced to 32G, this In conclusion, the journey to adapt Cassandra for our transaction workload was a transformative endeavour. It required us to think creatively and design our data model with an emphasis on denormalization. The meticulous handling of duplicated data and maintenance of consistent records proved to be crucial in overcoming the challenges posed by the absence of traditional SQL operations in Cassandra.

5 References

1. Getting started with Citus - Westermann
   https://www.dbi-services.com/blog/getting-started-with-citus-setting-up-a-four-node-cluster
2. Postgres Parallel indexing in Citus
   https://www.citusdata.com/blog/2017/01/17/parallel-indexing-with-citus
3. Multi-tenant Applications
   https://docs.citusdata.com/en/v9.3/use_cases/multi_tenant.html
4. Cassandra, “Cassandra Data Modeling Introduction,”
   https://cassandra.apache.org/doc/latest/cassandra/data_modeling/intro.html
5. DataStax, “Basic Rules of Cassandra Data Modeling,”
   https://www.datastax.com/blog/basicrulescassandradatamodeling