Group 1: Datastore project - A performance comparison of PostgreSQL and Apache Cassandra

Matthaeus Hilpold (13298933), Elisabeth Kräman (13314998),
Marta Turek (13076698), Will Chien (13236490)

ABSTRACT

When speaking in "Big Data" terms NoSQL databases are claimed to outperform SQL databases. In this paper we aim to investigate the performance of a NoSQL database, Apache Cassandra, against a SQL database, PostgreSQL. We compare database performance on increasing loads of read operations on a public retail dataset. Database read query latency and CPU, disk and memory usage is monitored and the trade-offs required to achieve respective performance are discussed.

Our results show that Cassandra read queries are faster than the equivalent read queries performed on PostgreSQL, but Cassandra is unable to efficiently perform aggregations or joins on the data. Cassandra data needs to be modelled exactly according to the queries that are going to use it.PostgreSQL is able to cache the results of complex queries and optimise resource usage for frequently executed queries. It can perform complex joins and arithmetic operations on data more efficiently than Cassandra.

1 INTRODUCTION

Traditional database systems for storage have been based on the relational model. However, NoSQL databases, which are based on storing simple key-value pairs, have risen in popularity on the premise that simplicity leads to speed. Key-value databases can scale to handle large amounts of data and extremely high volumes of state changes while being accessed simultaneously by millions of users through distributed processing and storage [1]. The choice of database system is crucial for most services. There is however no clear winner between NoSQL and SQL systems, as the performance of each depends highly on context.

This datastore project aims to investigate relative database performance for querying a retail dataset by conducting a comparison between the PostgreSQL relational database and the Apache Cassandra non-relational database. Ten different queries were run on the databases and were designed to represent increasing level of execution complexity, from simple select statements that query a single record in a table to more complex queries with multiple join statements that retrieve all queried records. It was interesting to observe how the databases performed with different amounts of data stored, increased hardware resources and varying surges of reads per second. The Python programming language was used to create the scripts which generate the data and insert it [2, 3] into the database tables. Google's Go programming language was used to execute the queries [4, 5] owing to its seamless asynchronous execution.

The main challenges revolved around creating a similar environment for both databases, when comparing query performance and guaranteeing the meaningfulness of the results. On the server side, we had to account for matching deployment environments, same amount and type of data, and manage timeouts caused by overload due to flooding of requests. On the client side, a major challenge was that the script performing read queries needed to execute all the queries simultaneously and thus be asynchronous. Network latency, which impacts the quality of time measurements was addressed by deploying the databases to the same region and running all queries from a single location.

Detailed Contributions.

- Matthaeus: Researched Cassandra data modeling logic and indexing, and modelled data for Cassandra. Wrote SQL and CQL queries. Set up a PostgreSQL database VM, wrote a program that inserts data into PostgreSQL database.
- Elisabeth: Researched Cassandra data modeling logic and indexing, and wrote Cassandra queries. Set up a Cassandra database VM, wrote a program that executes queries X number of times. Executed all queries. Recorded video.
- Marta: Researched PostgreSQL specific features, normalised data and visualised with entity-relationship diagram. Wrote a program that inserts data into Cassandra table. Wrote SQL queries. Coordinated report writing. Acted as second reader for research paper.
- Will: Researched PostgreSQL specific features, normalised data and visualised with entity-relationship diagram. Wrote CQL queries. Set up Prometheus. Visualised experiment results. Summarised research paper.

Achievements.

- We established, that when modelled appropriately, Cassandra performs better with respect to latency and CPU usage.
- If the data is not modelled for queries specifically, Cassandra can struggle - especially when calculations and aggregations have to be performed. PostgreSQL therefore offers more flexibility and support for more complex queries and joins.

2 PROBLEM & APPROACH

2.1 Problem Statement

To compare Cassandra and PostgreSQL performance, both need to be monitored under the same conditions. The amount of data in the database, underlying hardware and received load have to be exactly the same. Network latency has to be taken into account when measuring read query latency. Database resource utilisation needs to be compared, to be able to gain insights on database performance. We aim to assess which type of query scenarios are more suitable for which database by measuring latency, CPU, memory and disk utilisation.

2.2 Approach

2.2.1 PostgreSQL Model. The data for the PostgreSQL relational database was modelled according to the Relational Database Model [6] and is represented by the Entity Relationship Diagram (ERD) in Figure 1. The Entity Relationship (ER) Modelling process was applied to develop clean table structures, after which the process of normalization was applied to minimise data redundancies and to ensure that each table conforms to the concept of well-formed relations [7].

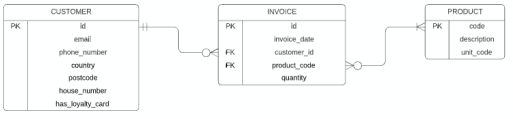

Figure 1: PostgreSQL Entity Relationship Diagram

2.2.2 Cassandra Model. Cassandra is a wide-column store system, which can be described as a hybrid between a tabular system and a key-value pair based one [8]. In our setup it keeps most of the data in one table on one node. Customer, invoice and product information as well as their keys are stored in the table *customer_invoice*. Since disk space is a cheap resource [9], three additional tables with aggregate values were created to optimise for queries 5,6,7 and 9, namely: *daily_revenue*, *product_revenue* and *product_sale_counts*. While this would break the atomicity criterion of normalised databases, in Cassandra, aggregates and duplication are accepted and even encouraged [9]. In fact, aggregation and arithmetic computation on Cassandra are computationally resource heavy and is only supported in newer versions.

2.2.3 Conditions. To compare Cassandra and PostgreSQL, we are attempting to evaluate performance under following conditions. Scaling the volume of data that each database contains, from 20% to 100% of the dataset with 20% intervals. Increasing the database cores and memory from 2CPU and 4GB of memory to 4CPU and 16GB of memory. Running the experiment with 4 distinct loads: 1 query per second, 100 queries per second, 1000 queries per second, 10000 queries per second. Running both database loads at the same time, from same physical location to make sure that the network latency is same for both databases and queries.

Figure 2 describes the ten queries that were translated into Structured Query Language (SQL) and Common Query Language (CQL) respectively, and run on both databases.

2.3 Implementation

Databases were deployed to Google Cloud Platform Virtual Machines, first with 2CPU and 4GB of memory and then with 4CPU and 16GB of memory. Tables were created and data inserted into the databases with Python scripts.

To monitor Cassandra and PostgreSQL we set up a Prometheus time-series database to collect CPU, disk and memory usage. Prometheus is a pull based database and queries data from specified endpoints every 15 seconds. To provide metrics for

Query 1) Get customer X's email address by invoice ID.
Pseudocode: SELECT email WHERE invoice_data.customer_id = customer_data.customer_id AND invoice_id = given invoice_id;

Query 2) Get customer X's last purchase date.
Pseudocode: SELECT invoice_date WHERE customer_id = given customer_id ORDER BY invoice_date DESC LIMIT 1;

Query 3) Get customer X's last purchases between specific time ranges.
Pseudocode: SELECT customer_id, invoice_id, invoice_date, product_code, product_description, quantity_purchased, unit_price WHERE customer_id = given customer_id AND invoice_date IS BETWEEN given start_date AND given end_date ORDER BY invoice_date DESC;

Query 4) Get customer X's whole purchase history by customer ID.
Pseudocode: SELECT customer_id, invoice_id, invoice_date, product_code, product_description, quantity_purchased, unit_price WHERE customer_id = given customer_id ORDER BY invoice_date DESC;

Query 5) Get the email addresses of the top 100 customers ordered by expenditure.
Pseudocode: SELECT email, SUM(quantity_purchased x unit_price) AS total_expenditure GROUP BY customer_id, email ORDER BY total_expenditure DESC LIMIT 100;

Query 6) Get all customer expenditure ratio per order.
Pseudocode: SELECT customer_id, (total_expenditure/total_order_count) AS average_expenditure ORDER BY average_expenditure DESC;

Query 7) Get the total sales per day ordered by date.
Pseudocode: SELECT SUM(quantity_purchased x unit_price) GROUP BY invoice_date ORDER_BY invoice_date;

Query 8) Get all products, ordered by sales count, highest to lowest.
Pseudocode: SELECT product_code, product_description, SUM(quantity_purchased) AS total_quantity_sold GROUP BY product_code, product_description ORDER_BY total_quantity_sold DESC;

Query 9) Get all products, ordered by generated sale revenue, in descending order.
Pseudocode: SELECT product_code, product_description, SUM(quantity_purchased x unit_price) AS total_revenue GROUP BY product_code, product_description ORDER_BY total_revenue DESC;

Query 10) Get all orders from customers in a given postcode.
Pseudocode: SELECT invoice_id WHERE postcode = given postcode AND invoice_data.customer_id = customer_data.customer_id;

Figure 2: Queries run on both databases

Prometheus to scrape, we installed on each database server a Prometheus Node Exporter, which exposes a wide variety of hardware- and kernel-related metrics. Prometheus provides a user interface where all collected metrics can be queried and observed. To evaluate latency, the experiments were first conducted with a Python script using the psycopg2 adapter [2] executing all queries for three different loads automatically. This experiment revealed, however, that asynchronicity is essential to test the limitations of the databases properly. Thus, we switched to the Go language, which is better suited for parallelisation. We implemented a script for each database type using the respective Go adapters, PostgreSQL client and ORM for Go [4] and gocql [5] to establish a connection to the databases in the cloud. The query string and the load parameters have to be set for the experiment before running the script. A WaitGroup suitable to the load is deployed and the Go routines are launched from a loop to asynchronously query the database as per the specified load and time the execution. Before aggregating (summing) the result, the WaitGroup forces the routines to synchronise. The Go scripts only differ for Cassandra and PostgreSQL in how they connect to the database.

3 EVALUATION

3.1 Experimental Setup

Prometheus metric monitoring system, two Cassandra and two PostgreSQL databases are hosted in separate Virtual Machines (VM) on Google Cloud. As time is measured on our clients, the VM instances are located in the same zone (us-central1-a) to ensure that the difference in network latency for requests via the internet would be negligible. One of each database's VMs is a e2-standard-4 (4 vCPUs, 16 GB memory) and the other is a e2-medium (2 vCPUs, 4 GB memory) machine. The experiment load generator ran on a personal laptop in Tallinn, Estonia. Request loads were all generated by one person, to keep the network latency consistent. The VM's run Ubuntu 20.04 LTS operating system.

The load generator, written in Go, used Cassandra and PostgreSQL database adapter libraries [4, 5].

3.2 Datasets

A public online retail dataset from the UC Irvine Machine Learning repository [10] is used in this project. The dataset contains 1,067,371 transactions for a UK-based retailer between

01/12/2009 and 09/12/2011. A second dataset, with 18,356 rows is automatically generated from the core dataset to facilitate the execution of more complex join queries.

3.3 Results

3.3.1 Query Response Latency. In general a database with a smaller dataset experiences lower latency, as it needs to process less data. Figure 3 shows that both databases with 20% of the data experience less latency compared to the databases with 100% of the data. It can be observed that PostgreSQL latency increases noticeably when there is more data in the database compared to Cassandra latency, which does not change as much. In addition, no matter how many queries per second are requested, response latency in Cassandra is remarkably lower than the figures for PostgreSQL.

PostgreSQL 100 (reads per second) latency is sometimes longer than for larger loads, for complex queries 6, 7, 8, and 9. This occurs when query results are cached and used when load increases. The longest latency (2917.62 milliseconds) in Cassandra is noted in query 6 with 4 CPUs when serving 10,000 reads simultaneously. This query contains arithmetic operations, for which Cassandra is not designed.

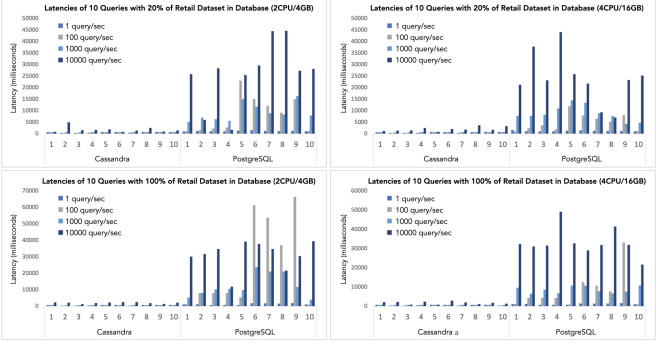

Figure 3: Query response latency in different scenarios

3.3.2 Disk I/O. The spikes in the top graph of Figure 4 show that PostgreSQL utilises its disk when it is queried, because of its limited memory (4GB). While both Cassandra databases and the larger PostgreSQL (16GB) are able to operate in memory.

3.3.3 CPU utilisation. The bottom graph in Figure 4 demonstrates the CPU utilisation for queries 6 and 7. When query 6 is executed 10,000 times simultaneously, both Cassandra CPUs run at full capacity and take seven minutes to recover. The same execution takes PostgreSQL less than a minute. This was caused by a query containing arithmetic operations, for which Cassandra is not meant. For query 7, Cassandra uses less CPU than PostgreSQL, because the data was preprocessed and no complex operations had to be performed on it by the database.

3.3.4 Memory utilisation. Figure 5 describes the database memory usage for query 7 execution, where Cassandra has preprocessed tables and PostgreSQL needs to aggregate data. Cassandra always keeps its data in memory, depicted as a constant line on the graph. While PostgreSQL needs to take several

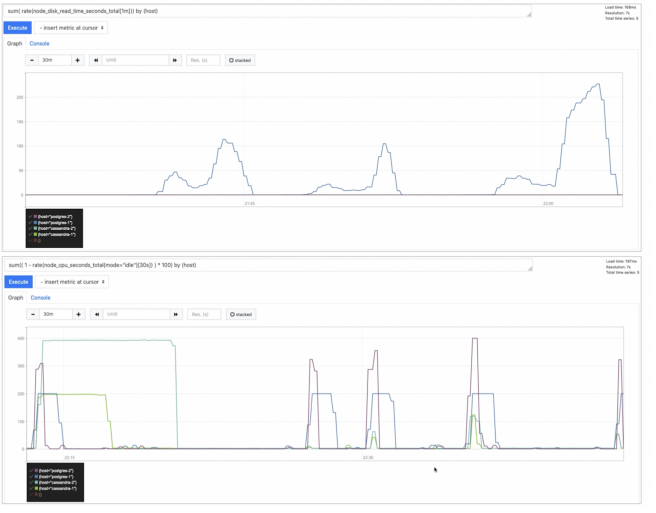

Figure 4: Disk read time seconds PostgreSQL-1 (top), CPU % utilised (bottom)

steps, including loading data in memory, processing and saving the result. The spikes indicate loading reads for 100, 1000, and 10,000 times respectively. The spike for 100 reads is most memory intensive because it is the first time the query is computed. The results are stored and following time it does not need to load all the data into memory.

Figure 5: Memory utilisation for query 7

3.4 Discussion and conclusion

PostgreSQL represents a shared-memory architecture, where all nodes have shared access to both memory and disk. PostgreSQL loads required data from the disk to a shared buffer pool, and it works directly within memory to reduce disk access. On the other hand, Cassandra represents a shared-nothing architecture, where data stored in independent nodes communicates.

As previously discussed, the main challenges related to achieving a realistic setup were with respect to server architecture, queries and experiment design.

Further research is needed to test how partitioning across nodes affects Cassandra's performance. Nodes, and therefore support for horizontal scalability, represent one major selling point for Cassandra and it stands to reason that collecting data across nodes would impact latency and memory usage metrics of Cassandra.

4 RESEARCH PAPER SUMMARY

Paper choice. We summarised *"Spanner: Google's Globally Distributed Database"*. During the Big Data course, the trad-eoff between managing data volume and maintaining high availability and consistency in a distributed system was frequently accentuated. *Spanner*, a temporal multi-version database, is the first system to distribute data globally and offer strong externally-consistent distributed transactions.

Summary of the paper. Horizontal scaling enables users to add additional servers to the cluster and increase performance linearly. However, in a distributed system this can create issues of inconsistency. In master-slave architectures, some latency might occur when data is propagating. There might exist a window of time where master and slaves may have different states. This is unfavourable in applications that require strong consistency.

To address this, Google introduced the Spanner architecture, a globally distributed database, in 2012. Not only does Spanner scale horizontally, but it also offers externally consistent reads and writes across databases. These features enable Spanner to support consistent backups, consistent MapReduce executions and atomic schema updates, all at a global scale, and even in the presence of ongoing transactions.

A Spanner deployment is called a *universe*. A handful of universes are deployed on a global scale. Spanner is organised as a set of *zones*. Each zone has one *zonemaster* and up to thousands of *spanservers*. Every spanserver is responsible for between 100 and 1000 instances of a data structure called a *tablet*. To support replication, each spanserver implements a single Paxos state machine, which stores its own metadata and log, on top of each tablet. The state machines are used to implement a consistently replicated bag of mappings. The set of replicas is collectively called a *Paxos group*. In a Paxos group, as long as a majority of the voting replicas for the split are up, one of those replicas can become the leader to process writes and allow any other replica, that is sufficiently up-to-date, to serve reads.

The key enabler of external consistency at a global scale is TrueTime, an API that leverages hardware-assisted clock synchronisation with GPS and atomic clocks. With TrueTime, despite distributed transactions, Spanner can assign a timestamp around the world, from a globally agreed-upon source to every transaction upon commit. The implementation consists of a set of time master machines in every datacenter and a time-slave daemon per machine. In a conventional setting, standard time interfaces do not provide information regarding uncertainty to users. Conversely, TrueTime introduces an interval with bounded time uncertainty where time is represented as an internal. TrueTime can therefore be used to guarantee the correctness properties around concurrency control, and those properties are used to implement features such as externally consistent transactions, lock-free snapshot transactions, and non-blocking reads in the past.

Replication, transactions and availability were evaluated. The latency experiments showed that as the number of replicas increases, the latency increases slightly. Snapshot transaction throughput stays roughly constant with the number of replicas. Write throughput decreases with the number of replicas, since the amount of work that Paxos does increases linearly with the number of replicas. The availability tests showed the benefit of running Spanner in multiple datacenters. The tests highlighted that doing a leader-hard kill on servers severely impacted throughput with rate of completion dropping almost to 0. Regarding TrueTime performance, the results show that clock issues are extremely infrequent, relative to much more serious hardware problems. This points to TrueTime's implementation being as trustworthy as other pieces of software upon which Spanner depends.

The paper further features a short case study on the successful transition of Google's advertising backend, called F1, to Spanner.

Strong points. A strong point of the paper is Spanner's key feature, TrueTime. The paper clearly details how reifying clock uncertainty in the time API makes it possible to build distributed systems with much stronger time semantics. Additionally, the paper shows in the experiments that as the underlying system enforces tighter bounds on clock uncertainty, the overhead of the stronger semantics decreases.

Another strong point is the description of Spanner's data model, its ease of use, the semirelational interface, transactions and SQL-based query language. From a system perspective, the paper details well the scalability, automatic sharding, fault tolerence, consistent replication, external consistency, and wide-area distributions.

Weak points. Given the realisation and iteration of Spanner was in part based on database features that the Bigtable system was missing, a more comprehensive treatment of these improvements and differences would allow the informed reader to better appreciate how Spanner has evolved from Bigtable.

A comparison of other relational databases, such as Amazon Aurora, PostgreSQL, and MySQL would further serve to enhance reader's understanding of Spanner's key standout features. Relevant metric comparison including consistency, availability, scalability, and latency might serve as referable indicators in database selection.

REFERENCES

[1] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings*, pages 15–19, 2013.

[2] psycopg2 · PyPI.

[3] cqlengine documentation — cqlengine 0.21.0 documentation.

[4] lib/pq: Pure Go Postgres driver for database/sql.

[5] gocql/gocql: Package gocql implements a fast and robust Cassandra client for the Go programming language.

[6] The Structure of the Relational Database Model - Jan Paredaens, Paul De Bra, Marc Gyssens, Dirk van Gucht - Google Books.

[7] Database Systems: Design, Implementation, & Management - Carlos Coronel, Steven Morris - Google Boeken.

[8] Wide Column Stores - DB-Engines Encyclopedia.

[9] Basic Rules of Cassandra Data Modeling | Datastax.

[10] UCI Machine Learning Repository: Online Retail II Data Set.