Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Assessing the performance of distributed PostgreSQL

Author: C.C. Felius (12368032)

1st supervisor:	Prof. Dr. Peter Boncz
daily supervisor:	Dr. Marco Slot (Microsoft)
2nd reader:	Dr. Gábor Szárnyas

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computational Science

December 23, 2022

Abstract

Citus is an extension on PostgreSQL that enables to scale distributed tables and shards across many nodes. This extension provides a mode that enables to 'query from any node' to increase throughput. In this thesis, we aim to investigate the scalability limits of this feature by means of measuring the performance of Citus when executing 100% INSERT and 100% SELECT workloads. Quantifying performance however is challenging since Citus is a distributed system where queries are distributed across multiple worker nodes. To map the performance and bottlenecks of Citus, we compose a benchmarking infrastructure that enables to automate benchmarks without introducing significant overhead. We found that for INSERTS the system seems memory limited and I/O bound, but for SELECT queries the system behaves as CPU bound. Future research should mainly focus on improving the efficiency of simple SELECT queries.

Contents

List of Figures **iii**

List of Tables **vii**

1 Introduction **1**

1.1 Goals . . . 2

1.2 Outline . . . 3

2 Background **5**

2.1 Terminology . . . 5

2.2 PostgreSQL . . . 14

2.3 Citus . . . 19

2.4 Amazon Aurora . . . 24

2.5 Google Spanner . . . 26

2.6 Statistics . . . 27

3 Related Work **29**

3.1 Benchmarking . . . 29

3.1.1 Tooling . . . 29

3.1.2 Frameworks . . . 31

3.2 Profiling . . . 33

3.2.1 Tooling . . . 33

3.2.2 Distributed Tracing . . . 34

3.3 Bottlenecks . . . 36

3.3.1 Identification . . . 36

3.3.2 Understanding . . . 38

3.3.3 Applicability . . . 39

3.4 Scalability . . . 39

3.5 Queueing 40

4 Benchmarking **43**

4.1 Methodology 43

4.2 Experiments and Results 47

4.3 Performance Comparison 53

5 Profiling **59**

5.1 Benchmarking Infrastructure 59

5.2 Experiments 70

5.3 Results 76

5.4 Bottlenecks 94

6 Discussion **97**

7 Conclusion **101**

References **105**

7.1 Example YCSB Insert 113

7.2 PostgreSQL logs 113

List of Figures

11figure.2.1
2.2 PostgreSQL query parser 15
2.3 Citus Architecture (source: citusdata.com) 20
2.4 Citus 11 (or Citus 11.0) architecture (source: citusdata.com) 21
2.5 Citus Global Process ID parsed (source: Citus Blog) 23

3.1 Flow Chart Diagram of USE-Method (1) 37
3.2 High-level overview of Cassandra YCSB architecture (source: (2)) 41

4.1 Transactions per Second (TPS) measured by YCSB for Citus in production, varying thread counts and cluster sizes. Driver VM: `Standard_E16ds_v3`, worker nodes with 16 vCores, 2 million operations. INSERTS (left), READS (right) 47
4.2 INSERTS (left) and SELECTS (right). TPS for Citus cluster of varying worker nodes, shard count of 2 × worker nodes, 100M reads on 100M records, 800 and 990 threads respectively and a single Driver VM of 64 vCores . . . 49
4.3 Performance of Managed Service (production) vs Performance of Marlin. Both consists of Citus clusters of 16 worker nodes with each 16 cores 49
4.4 Performance in TPS on V3 and V5 hardware for read-only workload for 8-node and 16-node Citus clusters (left), Performance on V3 and V5 hardware for insert-only workload for both 8-node and 16-node Citus clusters (right). 50
4.5 Y-axis is Transactions per Second (TPS). Graph shows TPS for SELECT workload when different amount of connections are used. Machine used: `Standard_E16ds_v5`, 2TB SSD, 128GB RAM, 100M records. Amount of connections on x-axis, throughput in transactions per second on y-axis. Amount of connections are doubled every iteration: 400, 800, 1600 and 3200 51

4.6 Y-axis is transactions per second. TPS for INSERT workload. Machine used: (`Standard_E16ds_v5`), 2TB Disk, 128GB RAM, 100M records. Amount of connections on x-axis, throughput in transactions per second on y-axis. Amount of connections are 800, 1600, 1800, 3200 and 3600 52

4.7 Y-axis shows Transactions Per Second (TPS). TPS for SELECT workload (800 connections, right) and TPS for INSERT workload (3600 connections, left) for different amount of workers. Machines used: (`Standard_E16ds_v5`), 2TB Disk, 128GB RAM, 100M inserts. 53

4.8 Results of Aurora vs Citus for the same amount of Vcores. For Aurora around 700 connections are used, for Citus 3200. All YCSB workloads (table 3.1). 54

4.9 Benchmark results of Spanner vs Citus. All YCSB workloads (table 3.1) . . 57

5.1 JDBC connections with unequal load balancing of direct connections. 2 Worker nodes, 16 vCPUs, 128GB RAM, 2TB disk, 3600 connections. 78

5.2 Left: INSERTS, Right: READS. Distribution of average query execution times from the Postgres client until its forwarded back to the YCSB client. Execution times are derived for a workload of 100 million `INSERT` and `SELECT` queries with 3200 and 800 threads respectively. 79

5.3 Profile of time of `INSERT` query spent in different monotasks, binary protocol enabled. *fsync()* not captured. 79

5.4 Profile of `INSERT` query without binary protocol, *fsync()* captured 82

5.5 Y-axis represents throughput. Scalability of Citus for 100 Million INSERTS (YCSB workload a), varying connections, 16 worker nodes with 32vCPUs (`V5_E16_ds`), 256GB RAM, 2TB disk. Data is fitted with Numpy `Polyfit` function. 84

5.6 CPU utilization for Citus cluster of 32 worker nodes, shard count of 64, 100M inserts, 800 and 3200 threads respectively and a single Driver VM of 64 vCores. Only inserts. Threads in this case means connections to the entire Citus cluster. 87

5.7 Await reported by iostat during an Insert workload run for Citus cluster of 32 worker nodes, shard count of 64, 100M reads on 100M records, 3200 threads and a single Driver VM of 64 vCores. 1b is a zoomed version of 1a, truncated at 250 ms waiting time. 88

5.8 Profile of query execution times for 100 million `SELECT` queries without driver-to-cluster latency, cluster-to-driver latency and YCSB overhead, 800 threads, 16 vCPUs 89
5.9 CPU utilization for Citus cluster of 32 worker nodes, shard count of 64, 100M reads on 100M records, 800 and 3200 threads respectively and a single Driver VM of 64 vCores. YCSB workload c. 91
5.10 Trashing explained in a graph. Image derived from `https://www.studytonight.com/operating-system/thrashing-in-operating-system` 92

LIST OF FIGURES

List of Tables

3.1 YCSB Core Workloads (Source: (3)) 31

4.1 Specifications for V3 and V5 hardware used for VMs in Azure 50
4.2 Throughput, mean and 99th percentiles query execution times for 100 million `INSERT` queries without driver-to-cluster latency, cluster-to-driver latency and YCSB overhead, 3600 threads, 32 workers, 16 vCPUs 53

5.1 Input parameters for benchmarking in Marlin 60
5.2 Coordinator and Worker nodes cluster configuration 75
5.3 Monitoring overhead for 5-node Citus clusters (`Standard_E32ds_v5`) with 100 million `SELECT` queries and 800 connections based on four iterations . . 76
5.4 Percentiles query execution times for 100 million `INSERT` queries without driver-to-cluster latency, cluster-to-driver latency and YCSB overhead, 3600 threads, 16 vCPUs 82
5.5 10 percent sampling frequency for a cluster with 32 worker nodes, 16 vCPUs, 128GB RAM and 2TB disk and 100 Million inserts and 3600 connections . . 83
5.6 Benchmark results of Citus clusters with `Standard_E32ds_v5` (256 GB RAM) hardware. Coordinator and worker nodes contain 32 vCPUs, 256GB Ram, 2TB disk. Benchmark workload is 100 million inserts and 16 worker nodes . 84
5.7 Benchmark results of Citus clusters with `Standard_E16ds_v5` hardware. Coordinator and worker nodes contain 16 vCPUs, 128GB Ram, 2TB disk. Benchmark workload is 100 million inserts and 3600 connections 86
5.8 Percentiles query execution times for 100 million `SELECT` queries without driver-to-cluster latency, cluster-to-driver latency and YCSB overhead, 800 threads, 16 vCPUs 89
5.9 Most CPU-consuming methods during a `SELECT` query across eight samples 91

LIST OF TABLES

1

Introduction

Database management systems (DBMSs) are continuously evolving to leverage horizontal scaling and elasticity enabled by the cloud. Nowadays PostgreSQL is a popular choice for developers, although this database was yet coined in the 1980s (4). The popularity of this DBMS however makes cloud providers such as Google, Amazon and Microsoft attempting to modernize PostgreSQL by optimizing and scaling different parts of the system. Examples of modern database systems that offer scalable PostgreSQL are `Aurora` (5), `Spanner` (6) and `Citus` (7). In this thesis we focus on Citus, which was acquired by Microsoft in 2018. Citus allows to scale PostgreSQL by creating distributed tables and shards across different nodes in a cluster. In addition, Citus enables elasticity by allowing users to scale up or down by adding or removing nodes (7) to deal with in- or decreasing workload intensities. Moreover, Citus is purposefully built as an extension of PostgreSQL such that it is compatible with the open source ecosystem of libraries and tools that PostgreSQL offers (7).

Citus has a specific mode where a user can query from any node in the cluster. Although this 'query from any node' mode of Citus existed for several years, its limitations are unknown. With the amplifying availability of data however more use-cases have arisen that demand databases that remain performant under intensive write- and read workloads. Moreover, a frequent requirement for users is that the database should be consistent when multiple concurrent writes happen. While NoSQL databases often guarantee high performance in terms of throughput, they fail in ensuring data consistency. Since PostgreSQL and thus Citus is based on the relational model, it provides a stronger level of consistency which makes PostgreSQL-based systems a popular choice.

In this research we microbenchmark the 'query from any node' feature, which is the default mode as of Citus 11. By exploring the behavior of Citus 11 under different conditions we evaluate which bottlenecks occur for large Citus clusters. To explore optimal configurations and fundamental limitations of Citus, we compose a novel infrastructure that enables benchmarking with Yahoo! Cloud Serving Benchmark (YCSB) (8) while monitoring the system under test (SUT). This infrastructure allows us to obtain an accurate profile of the internal behavior of large Citus clusters during intensive workloads. In addition, we compare the results of the benchmarks to the performance of `Aurora` and `Spanner`. As a result, there will be more clarity on the performance of different PostgreSQL offerings.

Benchmarking is frequently used to show the performance of the database SUT under different conditions (9, 10, 11). However when monitoring a SUT the adopted mechanisms should avoid significant overhead. This is important to avoid 1) a decrease in the overall performance and 2) a wrong indication of bottlenecks. For example, if a monitoring process is CPU intensive, one should avoid to measure the performance of the monitoring itself. In addition, due to the distributed nature of Citus other challenges arise in monitoring performance during benchmarks. Citus clusters often consists of a single coordinator and multiple worker nodes that are distinct machines. These machines communicate through the network and some nodes may fail during execution. Moreover, the execution of distributed queries involve operations on multiple worker nodes and also require a transaction to perform multiple network hops. Monitoring *all* these different resources that communicate and interfere during the execution of concurrent, distributed queries is challenging.

1.1 Goals

In this research we aim to benchmark Citus 11 and explore the behavior of Citus 11 under different conditions to evaluate which bottlenecks occur if at large Citus clusters under intensive workloads. We achieve this by first constructing an end-to-end benchmarking framework and subsequently model the performance to predict the performance of Citus under different system configurations. In addition, to assess the competitive landscape of Citus we aim to compare its performance to Amazon Aurora and Google Spanner. More specifically, the goals are:

1) Understanding the performance and exploring limitations of Citus 11 through benchmarks and analysis of the system. This leads to RQ1.

RQ 1. What are the bottlenecks of Citus 11 when a Citus cluster size or workload increases?

- **RQ 1.1 How can we instrument and analyze metrics in a distributed system under test (SUT), without causing significant overhead?**
- **RQ 1.2 Which bottlenecks occur when the Citus cluster size increases?**

2) Assessing the performance of Citus and compare it to its two main competitors; AWS Aurora and Google Cloud Spanner.

RQ 2. How does Citus perform in comparison with AWS Aurora and Google Spanner?

If we analyse the behavior of the system, we will get more insight into distributed database systems, which will help future research as these systems are continuously evolving. For Citus and PostgreSQL specifically, insights will be gathered to assess what some of their current limits are and where they can improve their system. In addition, with a comparison of different PostgreSQL offerings we will get more insight in when to use which system. This ultimately makes it easier for developers to choose the most appropriate RDBMS for their workloads. In summary, we aim for the following contributions:

- Gather insights about limitations of Citus 11 by exposing performance bottlenecks
- Find (sub-)optimal configurations for the performance of Citus (by e.g. managing client connections)
- Insight in different scalable PostgreSQL offerings by three large cloud providers

1.2 Outline

In this research we first explain some basic terminology and background concepts that are fundamental for understanding the subsequent chapters. Thereafter, we discuss related literature that discuss different benchmark- and profiling tooling along with some end-to-end frameworks for benchmarking and profiling distributed systems. Subsequently, we discuss

the initial benchmarks on Citus clusters along with AWS Aurora and Google Spanner. Then, we will discuss the profiling of Citus to expose its bottlenecks. Lasly, we have our discussion and future work.

2

Background

2.1 Terminology

To profoundly understand this research, we go over some required DBMS terminology that occurs in this thesis.

Relational Database Management Systems
Ted Codd proposed the relational model (12) to avoid having to constantly rewrite database management systems. The fundamental principle of this model, which describes a database abstraction, is to establish relations based on the values of data objects. He based this abstraction on three different principles.

1. Simple data structures (i.e. relations) are used to store the database
2. The data must be accessible using a high level language, but the database itself chooses the most effective execution method
3. The DBMS implementation is in charge of handling the physical storage

A relation itself is defined as "an unordered set that contains the relationship of attributes that represent entities. Since the relationships are unordered, the DBMS can store them in any way it wants, allowing for optimization" (13, pp.2). Data is changed by use of Data Manipulation Language `(DML)`. But schema changes are done through Data Defenition Language `(DDL)` commands.

OLAP and OLTP
Most databases are optimized for either Online Analytical Processing (OLAP) or Online

Transactional Processing (OLTP) workloads. This is, different applications have different kind of underlying query patterns which can roughly be distinguished in Online Analytical processing (OLAP) and Online Transactional Processing (OLTP) workloads. The former focuses on an Analytical workloads, i.e. complex queries involving summations or aggregations on large parts of the data. Examples of modern OLAP systems are DuckDB (14), ClickHouse (15) and Snowflake (16). OLTP however entails transactional workloads, which typically consist of simple queries that include modifications of records and simple reads (17). PostgreSQL, Spanner and Aurora focus on a transactional workload, where the challenge is not that individual queries are heavy, but that a continuous, high volume of transactions is executed at high throughput.

Transaction

In a database system, many things can go wrong. The network, application, hardware or software involved can fail at any time. In addition, it may occur that multiple clients write to the system concurrently. This could cause race conditions and leads to inconsistent data. A *race condition* occurs when an application or system attempts to perform multiple writes to an object at the same time. This could lead to undesirable behavior since the operation needs to be completed in a particular order to be correct. Transactions simplify these issues. That is, multiple read and write operations are condensed into one logical unit and carried out as one operation. The transaction can either succeed or fail as a whole. If a transaction succeeds, this is referred to as committed. If a transaction fails, the transaction will be aborted or rolled-back (18).

ACID

Realiable transactions should ensure *Atomicity, Consistency, Isolation and Durability* (ACID).

1. Atomicity ensures that a transaction is either completed or rolled back in case of failure, such that half-completed transactions in the database are avoided

2. Consistency is about seeing the right data at the right time. For example, a transaction only needs to access one version of the data during its execution.

3. Isolation covers concurrency, meaning that concurrent transactions cannot interfere with each other. The challenge is to maintain a good performance while providing serializability, i.e. the outcome of a transaction has to be equal to when all transactions were executed serially instead of concurrently (18)

4. Durability is important as changes in the database caused by transactions need to persist after the transaction is committed.

The **CAP theorem** can be considered as a rule of thumb for distributed systems. This theorem implies that when a distributed system is partitioned, i.e. two or more nodes cannot interact with each other due to a network failure, the system can be either optimized for availability or consistency. Hence, this theorem implies that distributed systems can only provide two of the three guarantees. The guarantees are defined as follows:

- Consistency: data on all nodes are synchronized and see the latest version of the data
- Availability: If some nodes are down due to e.g. a network problem not all requests can be processed. In this case, availability ensures that every request either waits until the problem is fixed or receives a response that the request failed.
- Partition tolerance: the system continues to operate despite arbitrary partitioning due to network failures

The CAP theorem is rather indicative, it can happen that systems can offer all guarantees. For example, when partitioned, consistency and availability could both be ensured (18).

Sharding

Sharding refers to partitioning a dataset into multiple parts. These parts of data are distributed among multiple machines and the process of sharding intends to improve the throughput of a system.

Hashing

Hashing is an irreversible cryptographic function (hash function) that maps data such that it produces a unique string for a given piece of data.

Hash-based sharding

Hash-based sharding is a method to shard data across different nodes. Sharding keys are hashed into a hash ring, and by applying a modulo this is turned into a partition-id. This way, the data is split in equal size partitions that represent virtual shards (19). Once the size of the data grows, or when a new node is added to a cluster, this may lead to the

necessity to re-partition the data.

Load balancer

A load balancer distributes incoming traffic among available servers so that request can be handled at a fast rate (20). Examples of load balancers are the `JDBC` loadbalancer for database load, or `nginx` (21) for web traffic.

Connections

A client can open multiple connections to a database to perform multiple operations in a parallel fashion. However, handling these connections require CPU and thus can be CPU bound. Establishing connections is an expensive task and keeping idle connections open is sometimes more beneficial, but this depends on the specific case. PostgreSQL only allows one operation at a time per connection, which could be a bottleneck while a large workload is executed against the database. If a query is executed through a connection the client awaits the response through the same connection. Only after the client receives a response, a new query can be posed.

Connection Pooling

PgBouncer is a lightweight connection pooler that sits between your application and the PostgreSQL database. When a new connection to the database is established, PgBouncer 'grabs' it and keeps the connection open so that it can be reused. This could ultimately increase response time as establishing new connections come with a performance penalty, and keeping the connections open will overcome this. The downside however is PgBouncer itself has overhead and is single-threaded.

Write Ahead Log (WAL)

Write ahead logging is the basic technique of persisting changes to disk before they have been persisted in all affected data pages. A WAL is an append-only file making writing to a WAL file reasonably fast and is mainly intended to make databases resilient to crashes (18). In the event of a crash recovery is able by reading out the log and redo the logged changes that have not been applied to the data pages.

Paging

Paging is a mechanism by the operating system (OS) that loads processes from storage into main memory. Main memory is often partitioned in fixed-size blocks, referred to as

frames, which should be the same size as pages in a database. In PostgreSQL, pages are by default 8kb. This mechanism is particularly used to ensure fast access to frequently used data. The size of pages could be increased. In PostgreSQL there is also an option that enables `huge pages`[1], which could increase performance by reducing time spent on CPU and memory management.

Checkpointing

A checkpoint is essentially a forcing of the dirty pages to disk. This means that the WAL is no longer required, as they are then already persisted. Checkpointing thus guarantees that all files have been updated with the information that is written to the database before the point in time in which the checkpoint occurs. This is beneficial for when a PostgreSQL server is restarted, since a checkpoint functions as a point in time from which all data will be recovered to reflect the state at that current time. Checkpoints are often an intensive process that could cause significant I/O as it flushes all dirty pages to disk, and therefore its execution is often spread out by e.g. usage of background writers.

Snapshots

A snapshot is a current state of a particular data point. Namely, pages could contain different versions of the same row. However, transactions should be able to see only one version of each row to get a consistent picture of the data within the same execution of the transaction.

Locking

To handle the concurrency of different transactions, most databases adopted some kind of locking mechanism to ensure data consistency. PostgreSQL uses locking in numerous occasions, for example when data pages are modified concurrently.

Concurrency control

PostgreSQL manages concurrent access to data by `Multiversion Concurrency Control` (MVCC), providing *transaction isolation*. This is, reading locks do not conflict with write locks, aiming for consistent transactions. The main goal of this concurrency control is to show the right version of the data.

[1] `https://postgresqlco.nf/doc/en/param/huge_pages/`

Two Phase Commit (2PC)

To enforce consistency across worker nodes, for example when some metadata is changed that needs to be propagated to all worker nodes, 2PC is a commonly used technique that achieves atomic transaction commit (18). In 2PC, the commit or aborting of a transaction is split into two distinct phases: the 'prepare' and 'commit' phase. It requires a coordinator or transaction manager which initiates the 'prepare' phase by sending a request to all participating database nodes once a transaction is ready to commit. If not all nodes respond, the coordinator will abort the transaction. Only if *all* nodes respond with a 'yes', the coordinator initiates the second phase by sending 'commit' to all nodes and the commit actually takes place.

Thread vs Processes

A thread is a set of instructions that is executed which can be managed by a scheduler. Contrary to a process, threads can access memory and states of other threads belonging to the same process, allowing for more dynamic share of resources. A process is thus more CPU-intensive then a thread and has its own memory which is not directly accessible by other processes.

Deamon Processes

A deamon process is a background process. This process only runs when the main process is running. Thus, if the main process terminates the deamon process also terminates.

Stall

A transaction can be stalled e.g. if it needs to wait for a response. To avoid idle CPU compute, the CPU will resume with another process in the meantime by *context switching*. If a transaction is stalled, this could cause a cascading effect when other transactions are dependent on the stalled transaction.

Context switching

Context Switching is a software event where a CPU is switching from one process to another by saving the state of a current process in order to resume another execution later. For every switch, the costs are saving the CPU state of the current process and loading the context of a new process. A lot of context switches in a short period of time could reduce performance as processors are fighting for available CPU rather than doing actual

compute work.

Interrupt Requests (IRQs)

An interrupt request is a signal that is sent to stop a running program, while an interrupt handler resumes as a running program.

System call

System calls are functions from the OS that call processes. For example, once a query which requires a reading a specific page is executed, the kernel could be asked to read a page which will place it in the *OS page cache*. Such a call to a kernel function is then called a system call. A system call most often requires a context switch to a kernel process, which therefore comes with a minor performance penalty since registers needs to be flushed and new data required for this context have to be loaded into the registers.

Database Storage

Figure 2.1 gives an overview of volatile vs non-volatile storage. Volatile storage refers to storage that is flushed every time a machine loses its power whereas non-volatile storage is persistent after such an event. CPU registers are the most fast and small storage and entail the storage of e.g. a single process. The CPU caches are fast accessible storage of the CPU and consist of different levels. DRAM is also referred to as main memory. It is the slowest volatile storage mechanism but way faster then any non-volatile storage. Non-volatile storage is persistent storage, and the most fast non-volatile storage

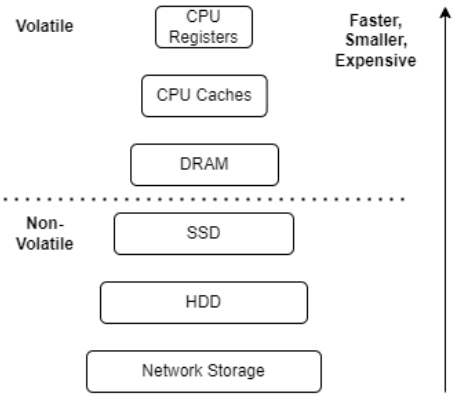

Figure 2.1: Volatile vs Non-Volatile database storage. Image in based on lecture 'Database Storage I' of Andy Pavlo at CMU[1]

Cache

Caching is storing data in fast accessible memory. Usually a cache is relatively small and fetching data from cache is supposed to be way faster then retrieving data from main memory or disk. Each modern CPU core has its own memory cache and CPUs usually have three different layers of caching. These layers are L1, L2 and L3 where $L1 \leq L2 \leq L3$ in terms of storage, but $L1 \geq L2 \geq L3$ in terms of accessing speed (18). A `cache hit` c_h refers to the event when a requested page resides in cache. In contrary, a `cache miss` c_m refers to the event when a requeste page does not reside in cache. For read-heavy workloads, the `cache hit ratio` *chr* (2.1) is a good indication if the required data pages are loaded into memory. If the cache hit rate is low, one could argue that the database does not perform well or that the cache size could be decreased.

$$chr = \frac{c_h}{(c_h + c_m)} \tag{2.1}$$

Scalability

It is not trivial to determine the scalability of a distributed database. The performance of a database system can be assessed if the load or system capacity changes. The latter can be be scaled by e.g. adding or removing nodes from a cluster, commonly referred to as *horizontal scaling* (11). Common metrics to measure the performance of a DBMS are throughput, response time and latency. Scalability can be defined as “the ability to cope with increased load" (18, pp. 10). As the amount of available data we process is doubled every year we need to cope with increasing loads. This means that there is a need to improve the performance of database systems to cope with these loads.

Load

According to Kleppmann (18) load can be described by different parameters which depend on the architecture of your DBMS. Examples of these parameters are request per seconds to a server, the amount of client connections, the ratio of reads and writes and data volume (11, 18). Since we are interested to explore the behavior of Citus 11 specifically if the client connections with the cluster increase, we define load here as a combination between the *number of client connections* and the *workload*. The **workload** in this research is defined

as the amount and complexity of transactions.

Throughput and Response time

Throughput is often defined as the number of records we can process per second. Instead of the number of records however, we define throughput as *the amount of transactions per second* as this is more appealing for OLTP use cases (22). Response time is related to troughput, but is defined as the time between a client request and receiving a response. Response time will vary across different runs of the same query and is thus rather a distribution of values. Since the *arithmetic mean* of all the responses does not conveniently capture outliers, we will use *percentiles* and *median* (50th percentile) to reflect what percentage of the requests are answered within a specific amount of time (18). Lastly, latency is the time a request is waiting to be handled. Latency can occur due to numerous reasons, such as package loss on WAN networks or unresponsive nodes. Since Citus is a distributed system, latency of communication between nodes within a cluster ought to be minimized to obtain a desirable throughput.

2.2 PostgreSQL

PostgreSQL is part of the Postgres project, coined in 1990 by Michael Stonebraker (23). Historically, PostgreSQL is composed of multiple processes that share memory among them. PostgreSQL is purposefully split into several processes since threads did not exist yet in 1990. By splitting PostgreSQL into multiple processes, it was possible to make use of more memory facilitated by its `shared memory`.

Interaction with a PostgreSQL database happens through a *client-server* system architecture. A `client` can be described as a program that requests activity from a system or program, called a `server`, to accomplish a certain task. A server thus is a program that *receives* requests from one or multiple clients and performs certain operations to allow the client to accomplish specific tasks (24). The client side of the PostgreSQL application consists of a program that allows to communicate with the PostgreSQL database. This could be a grapical user interface (GUI) such as pgAdmin or a terminal based interface such as `psql`. The server side consists of the actual database server that coordinates and manages connections and operations from which the Postmaster is the parent process.

Communication with PostgreSQL

The Postmaster process is a Deamon (background) process which initially handles the connection from a client. This is a parent process which acts as a 'supervisor' and coordinates the connection with the client. It does so by first checking if the client has required permissions and, if permission granted, it starts several background processes that enable the communication between the client and the database. From this point the client only communicates with these back-end processes. Note that these processes are able to handle a single query at the time submitted by the client. This makes that PostgreSQL is single-threaded, i.e. *process based* (25). However, multiple clients can connect to the system such that multiple queries can be executed concurrently. Still, while handling a large workload scaling issues will arise since a process consumes more memory and requires more state switches compared to a thread.

Shared Memory

Instead of reading and writing data directly from and to disk files, data is buffered in a shared memory area. This is a slightly old-fashioned way to share memory among processes. The shared memory area entails the shared buffer and shared WAL. Thus, once a

new process is created by the postmaster and connected with the client application, this process attempts to minimize disk I/O by only manipulating the shared buffer and shared WAL rather then reading directly from disk.

Background processes

The postmaster executes several programs in the background at runtime. First of all, the `logger` writes error messages to the log file. The `checkpointer` writes the dirty buffer to the file at a certain point in time, the *WAL writer* writes the dirty WAL buffer to the file, the `Logger` writes (error) messages to the log files and the Writer flushes all dirty pages to disk after the WAL is written. The `AUTOVACUUM Launcher` is a background process that regularly cleans up redundant data that entails e.g. deleted rows or old versions of updated rows. The `archiver` is optional and writes WAL logs to an archive. Lastly, the `Stats Collector` collects statistics about sessions and tables which can be queried by `pg_stat_activity` and `pg_stat_all_tables` and forwards this collected data to other processes by means of temporary files.

Query Processing in PostgreSQL

When a connected client issues queries against a PostgreSQL database, the queries are handled by back-end processes. These processes consist of roughly five subsystems (figure 2.2): the parser, analyzer, re-writer, planner and executor. The **parser** splits the query up in sub-tasks described in plain text and generates a parse tree. Subsequently, the **analyzer** performs semantic analysis and generates a query tree which is transformed by the **rewriter** according to rules stored in the 'rule system'. Following, the aim of the **planner**, or optimizer, is to generate the most optimal execution plan. To do so, it evaluates statistics and features of previous workloads, the costs of different query trees and ultimately chooses the most effective tree to be executed by the **executor**. The executor operates by means of the buffer manager. It uses some memory areas, such as `temp_buffers` and `work_mem` which are allocated in advance to create temporary files if necessary.

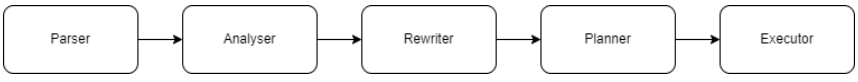

Figure 2.2: PostgreSQL query parser

Concurrency Control in PostgreSQL

PostgreSQL uses a specific version of MVCC. The default and most commonly used isolation level is *read committed*. When a data item is updated or inserted to the relevant page, PostgreSQL will select the right version of the data belonging to an individual transaction. It determines the right version by visibility check rules. For read committed, problems may occur if two transactions concurrently read a value from the same row and then both add 1 to this value in distinct transactions. It may happen that this value only increases by 1 instead of 2. *Serializable Snapshot Isolation* prevents this issue in rigorous way by cancelling one of the two transactions. Read committed however reduces this type of errors which makes it more commonly used. Moreover, as long as the value is read and updated within the same transaction, problems like the above will not occur.

Buffer Manager

The buffer manager takes care of the process of transferring data between main memory (volatile) and persistent (non-volatile) storage. In PostgreSQL, the buffer manager consists of a 3-layer structure: 1) the buffer pool, 2) buffer table and 3) buffer descriptors. The buffer pool is an array in which each slot contains one `buffer_id` and stores exactly one page. Buffer descriptors are stored in an array, where each descriptor points to exactly one slot in the buffer pool and holds metadata of the page stored in this slot. The buffer table maps the buffer tags to buffer ids in the buffer descriptor layer. Hence, when the buffer manager receives a request to access a particular page corresponding to a given `buffer_tag`, it calls the buffer table who maps the `buffer_tag` to a `buffer_id`, which corresponds to a `buffer_descriptor` that knows the exact location of the required page. On its turn, the buffer manager uses locks for various purposes.

How PostgreSQL relies on the OS

In PostgreSQL, a database thread is mapped directly to an operating system (OS) process. The OS handles the coordination of these processes, meaning that the usage of shared memory requires support from the OS. As a consequence, the shared buffer is historically advised to set to 25% and maximum 40% of memory. The OS maintains its own separate file cache, commonly referred to as the *OS page cache*, for the file system. This is, if the DBMS reads a page from disk the OS is going to keep it in its page cache and there will be another copy of the page in the buffer pool which comes with a minor performance penalty. Values over 25% for the `shared_buffers` can be useful if a large part of the workload fits in cache. This is particularly beneficial for read-heavy workloads such that

disk I/O is avoided. A larger value could be detrimental if the workload is write-heavy since the contents of `shared_buffers` are processed during writes.

PostgreSQL Logging

PostgreSQL provides several message levels to write certain events to the server log from which the default is WARNING. Other values include, DEBUG1 ... DEBUG5, FATAL, LOG, INFO, NOTICE and ERROR. Each event implies a certain severity of logging. **DEBUG1 .. DEBUG5** is intended for debugging and provides successively-more-detailed information for use by developers. **INFO** provides information implicitly requested by the user, e.g., output from VACUUM VERBOSE. **NOTICE** provides information that might be helpful to users, e.g., notice of truncation of long identifiers. **WARNING** provides warnings of likely problems, e.g., COMMIT outside a transaction block. **ERROR** functions as a warning and reports an error that caused the current command to abort. **FATAL** implies that a current session is aborted and reports an error. Lastly, **PANIC** implies an error that led to aborting all database sessions (26).

User-Defined Functions (UDFs) and Stored Procedures

Stored Procedures and User-Defined Functions `(UDFs)` are SQL functions consisting of procedural (i.e. declarative) statements. These functions are stored in the database and can be invoked by the SQL interface. In PostgreSQL both stored procedures and UDFs can be created by `CREATE FUNCTION`. However, stored procedures and UDFs have a couple of differences. For example, stored procedures include functions such as `UPDATE` or `DELETE`, while UDFs returns output.

`GROUP COMMIT`

A group commit is a feature in PostgreSQL that enables to flush a batch of transactions logged in the WAL in one go. This means that if multiple writes happen concurrently they will be appended to the WAL file and be flushed as a single batch as opposed to be flushed individually. This is amortizing for the WAL costs and can improve the throughput simply by adding more clients as shown in the benchmarks performed by Greg Smith in 2012[1]. He noticed that by using pgbench the throughput increased from 10 TPS for one connection to 2000 TPS for 300 connections by using this feature.

[1] `https://www.postgresql.org/message-id/CAEYLb_V5Q8Zdjnkb4+30_dpD3NrgfoXhEurney3HsrCQsyDLWw@mail.gmail.com`

`PREPARE` statements

Clients are able to compile `SQL` queries in PostgreSQL by means of `PREPARE` statements to optimize performance. When a `PREPARE` statement is executed, PostgreSQL parses, rewrites and analyses the query. If subsequently an `EXECUTE` statement is executed, the query is only planned and executed. This reduces the amount of work for subsequent queries of the exact same format, since its values are passed on as parameters, leading to an increase in throughput.

PostgreSQL extensions

As mentioned by the author of (27), one of Postgres' main design choices enabled its extensibility. Citus is purposefully build an extension on PostgreSQL. By being an extension, Citus is directly compatible with PostgreSQL when e.g. a new PostgreSQL version is released. Moreover, in this way Citus leverages the open source ecosystem of PostgreSQL.

2.3 Citus

Citus is based on a shared-nothing architecture. This is, each (virtual) machine runs database software. The machines are called *nodes* and contain their own independent CPUs, RAM and disks (18). Citus supports shared-nothing parallelism, meaning that a Citus cluster is composed by different independent machines (figure 2.4) that communicate with each other through the network. By no means one system can directly access memory or disk of another system. The coordination of the communication of these independent machines is done at the software level. In this case, the communication is entirely managed by the DBMS itself. Frequently, each node in a shared-nothing cluster is a complete DBMS in and of itself. This is, each machine is able to handle requests from the client, compiling queries and managing access to the data on the disk. Data is distributed (*sharded*) across nodes of the database cluster by *horizontal partitioning* such that each node contains a subset of the data. Examples of systems that support Shared-Nothing are Teradata, Tandem and IBM DB2 (25). A Citus cluster consist of exactly one coordinator and multiple worker nodes, where each node hosts a seperate PostgreSQL server. The coordinator in a Citus cluster initially handles queries from the client and routes or parallelizes them to one or multiple worker nodes. Since the coordinator particularly keeps metadata and does not contain actual shards, it can be a relatively small machine. There are three different types of tables in Citus.

(1) Distributed tables - A *distributed table* is a larger table in Citus from which its data resides on multiple worker nodes. Such a table is divided in *shards* and is partitioned by a *shard key*.

(2) Reference tables - A *reference table* is a smaller table that resides on all nodes in the Citus cluster. These tables are the same across all users.

(3) Standard Postgres tables - The *standard Postgres tables* are regular Postgres tables that typically are used for administrative sections of an application and do not join with a distributed table.

Citus allows to distribute tables across the cluster. Sharding a table across workers is not fully automated; a client is required to manually choose a *shard key* or *distribution key* and distribute their tables, such the data is sharded across different worker nodes in the

cluster. Choosing the right shard key is of key importance to obtain a good performance while using Citus. In Citus, a `reference table` is thus a table that resides on every node in a Citus cluster containing information for most workloads whereas a distributed table is a table that is sharded across many worker nodes in a Citus cluster distributed by a *shard key*. Citus provides some Citus-specific commands that allow to distribute these tables accross multiple worker nodes, e.g.:

```
SELECT create_distributed_table('table_name', 'shard_key');
```

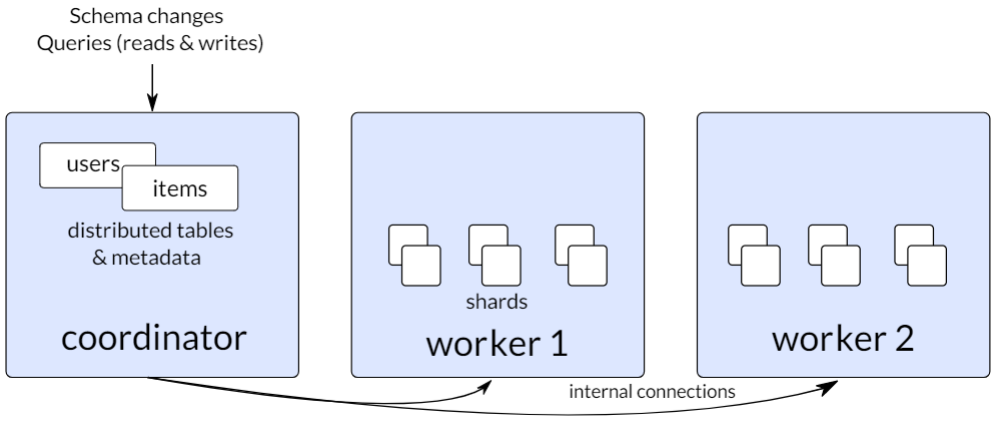

Figure 2.3: Citus Architecture (source: citusdata.com)

`DDL` are operations that usually modify metadata, such as creating or altering a new user and are currently only supported via the coordinator node.

Citus 11

Citus 11 offers a specific mode, formerly referred to as Citus MX, that enables the client to directly connect with worker nodes. This feature is standardized in Citus 11.0 and is mainly intended to improve performance of simple create, read, update and delete (`CRUD`) workloads. This means that for CRUD operations, a client can either connect with the coordinator or any worker node. Worker nodes in a Citus cluster hold shards as opposed to the coordinator node. More complex operations that change the metadata, such as adding a new column, can only be executed through the coordinator.

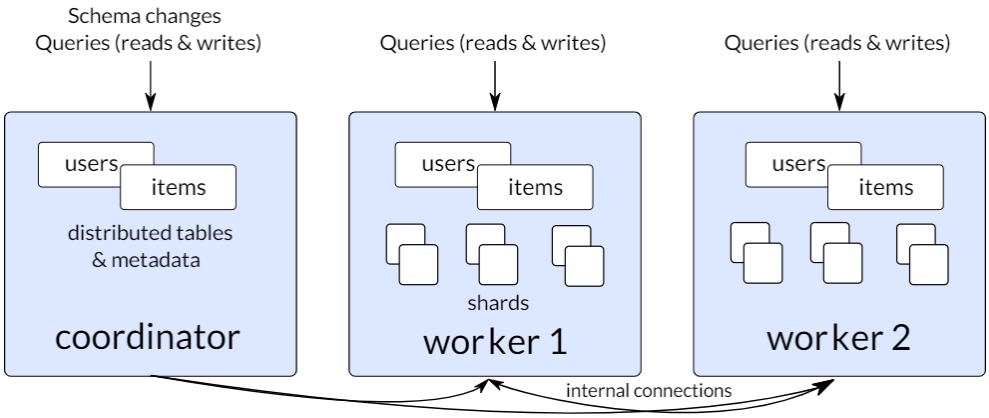

Figure 2.4: Citus 11 (or Citus 11.0) architecture (source: citusdata.com)

Query processing in Citus

Citus extends the plan and execute phase of PostgreSQL. The processing pipeline of Citus involves two distinct components; a *Distributed Query Planner and Executor* and a *PostgreSQL Planner and Executor*. The Distributed Query Planner creates a plan tree in the planning phase and translates the query into a format that enables parallelization of the query. Moreover, the Distributed Query Planner aims to minimize network I/O and performs several optimisations to ensure scalability. After this process, the query is split up in two distinct parts: a part that runs on the coordinator node and a part consisting of query fragments that run on individual shards on worker nodes. Once these query fragments have been established, the distributed query plan is forwarded to the *Distributed Query Executor*.

The *Distributed Query Executor* executes the distributed query plans while also handling failures. The executor opens a connection per shard on a worker and forwards query fragments directly to the shards. Since establishing a connection can be expensive, Citus caches by default one connection per node. Thereafter the executor awaits until the partial results have been computed, collects them and merges them to finally send the end-results back to the client. Once the fragmented queries arrive on the child node, the regular PostgreSQL Planner and Executor take care of the query.

Note that for `DML` operations or queries that involve a lookup based on the distribution column the process is slightly different. These operations typically involve only one shard. For these type of operations, the planner decides the correct shard by calculating its hash

by looking at the metadata of the distribution column. Following, the query-plan is rewritten such that it includes the reference to the shard and is subsequently forwarded to the distributed executor.

Life of a write Query in Citus 11

If a write query is executed on a Citus cluster, the query is parsed, analyzed and rewritten similarly as a 'regular' PostgreSQL query. However, as briefly touched upon in the previous subsection, the Distributed Query Planner looks at the Citus-specific `distribution column`. To determine to which worker and shard respectively the query will be to be forwarded, it calculates the hash of the shard by means of the value in the distribution column. For simplicity, we refer to the worker node that coordinates a query as a *parent*, and the worker node that a query is forwarded to is referred to as a *child*. Once the planner determined the route by calculating the hash of the required shard, the executor further handles the query and forwards it to the right child node. Once the query reached the child, the planner on the child node generates a new plan and lets this plan be executed by the executor on the same node. The executor makes sure that the insert is written to the WAL and modifies the index along with a page which, if it was not before, is now *dirty*. In PostgreSQL the WAL is continuously flushed to disk while the dirty pages itself are written to disk at least before a checkpoint occurs. After this checkpoint, the WAL files are deleted since the data is now located on persistent storage. Once the WAL is updated, the executor of the child worker node notifies the executor on the parent worker node, which on its turn notifies the client that the query has succeeded.

Life of a Read Query in Citus 11

Similarly as the write query, the read query is forwarded from the executor on a parent node to a child node where the actual shard that holds the tuple requested by the select query is located. Once the query is forwarded to the child node, the planner and executor on the child node now take care of the query. The planner determines the route by calculating the hash of the shard, and the executor executes this route. As opposed to a write query, no write to WAL and modification of indices as well as pages are required. Instead, a read query only needs to access the page containing the required data. If this page is already loaded in the shared buffer cache, the page can be read immediately from cache. If the page is not stored in the shared buffer, a system call to the kernel is made to check whether the page is stored in the OS page cache. This requires a context switch and therefore comes with a minor performance penalty compared to the shared buffer cache.

However, if the page also does not reside in the OS page cache the data has to be read from disk and the transaction will be stalled until the required page is loaded into memory from disk. When the required page is accessed and the tuple is read, the child executor notifies the parent executor which subsequently ensures that client receives the results of its query.

Transaction logging in Citus

Citus stores a global process id (`GPID`) for every distributed query. This GPID functions as a local process id (`PID`) to keep track of the worker and connection that initiated and executes a particular query. The GPID reflects a single connection and *not* a distinct query.

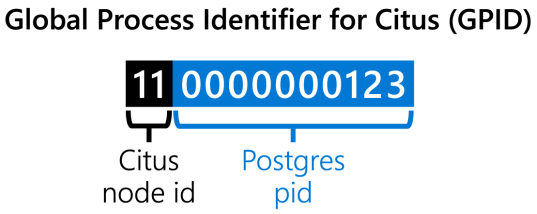

Figure 2.5: Citus Global Process ID parsed (source: Citus Blog)

The Citus global process ID (GPID, figure 5.4) is used to uniquely identify a query by its process ID. This global ID enables the user to cancel a specific query using

```
pg_cancel_backend(120000019449);
```

This can be useful e.g. if a slow query is blocking other operations. The first two integers from the left represent the ID from the node that initiated this particular query, while the resulting integers represent the Postgres process id. This Postgres PID refers to the (internal) connection from the client with Citus, which handles a query.

2.4 Amazon Aurora

In the shared disk model all different compute components are separated from the disk. The disk is accessible through a network layer that gives access for multiple devices to a centralized storage unit. The compute components, or *nodes*, still have their own memory but access the storage layer through the network. This is particularly beneficial to reduce disk I/O and efficiently coordinate disk access. Moreover, several operations that are usually performed within the database engine, such as writing dirty pages to disk, are pushed onto the disk layer. According to (28), shared disk database systems have less communication overhead between parallel query processing compared to shared-nothing and can therefore be beneficial for this matter.

AWS leverages this architecture as Aurora (5) decouples `compute` and `storage`, which allows them to optimize compute for flexibility and storage for durability. Aurora offers both `MySQL` and `PostgreSQL` compatibility. Within Aurora, compute represents the database engine and is mainly optimized for flexibility by taking care of e.g. client connection management, query processing, transaction management, locking and buffer cache. Storage takes care of backups, restore and replication while compute is able to scale if the workload changes. Their main philosophy is to reduce bottlenecks, which traditionally is known to be disk I/O, but with the cloud era moved to network I/O. Namely, the speed in which a modern DBMS issues writes could lead to an amplification of network traffic (5).

AWS Aurora provides a database cluster that consists of a primary instance that handles writes and reads. All modifications of the data are done through this primary database node. An AWS Aurora cluster can scale out by adding up to 15 replicas, i.e. read-only instances. These replicas can greatly improve read-scalability, but write-scalability remains limited. To reduce network I/O they write redo log records to storage instead of letting the database engine issue the actual writes. In addition, only changes in writes are logged and appended to the WAL file. Hence for Aurora the metaphor "*the log is the database*" is commonly used to describe their main internal workings.

Aurora uses a *boxcar* technique to optimize I/O by transferring sets of ordered logs records. The log records are ordered by their log sequence number (LSN), are shuffled, and are shipped to the storage layer in a partial ordered state. The storage layer actually writes the WAL files continuously to update the data pages. To improve response time, Aurora

makes use of a six-way quorum which is spread across three availability zones (AZ). This is, if 4 out of 6 writes succeed the write is considered as successful and an acknowledgement is sent. The main benefit is that all steps are asynchronous from the database engine and there are no events such as checkpointing that block writes. Moreover, continuous backups are streamed to Amazon S3 storage.

This approach works quite well for workloads that entail only writes or reads, but it has been criticized [1] as it is expected to pay a performance penalty if writes are read immediately after. Namely, all writes have to be replayed by the redo-logs in order to be read by the client, which is a time consuming process. Moreover, compared to a monolitic architecture, Aurora writes six times as much records but the network traffic is less.

In the compute layer Aurora implemented several optimizations that increase throughput. The first one is that the DBMS relies less on the OS compared to traditional RDBMSs that offer PostgreSQL. Aurora partly replaced the shared buffer 75% of the memory, whilst in traditional PostgreSQL this is often 25% to 40% (5). Another optimization is that the cache in Aurora is independent of any database processes. This means that in case of a restart or crash of the database, the cache is still in tact.

[1] MariaDB blog: `https://mariadb.com/resources/blog/dissecting-the-architecture-of-google-alloydb-amazon-`

2.5 Google Spanner

Google Spanner with a PostgreSQL interface is Googles globally distributed database which offers a PostgreSQL interface. Spanner shards data across multiple Paxos state machines which are located worldwide (6). If the workload intensity or the amount of (span)servers changes, Spanner takes care of automatic resharding this data to balance the load. Spanner comprises multiple zones, where each zone has its own zonemaster that assigns data to *spanservers*. In addition, each zone contains a placement driver that orchestrates the movement of data between spanservers, e.g. for loadbalancing. Each spanserver is in charge of 100 to 1000 tablets, which is similar to BigTables tablet (29). The state of a tablet is stored in Write-Ahead-Logs and B-tree file structures that are stored on Colossus (30). According to Corbett et al. (6), Spanner is rather a multi-version database since it assigns timestamps to data.

In summary, a Spanner cluster consist of one or multiple spanner nodes that can be added or removed any time. The exact definition of a node however in Spanner remains unclear, which makes it somewhat harder to compare Spanner to other distributed databases. Moreover, the PostgreSQL interface of Spanner makes use of an open-sourced API called PGadapter [1] that translates the PostgreSQL wire-protocol into a format readable by Spanner databases. This API, or proxy, thus needs to run in a separate process to enable communication from the client to the Spanner database instance with a PostgreSQL interface.

[1] `https://github.com/GoogleCloudPlatform/pgadapter`

2.6 Statistics

In order to get significant results from our benchmarks, we apply statistical formulas that are discussed in this section.

Arithmetic Mean

The mean is denoted as μ and the sample mean is calculated as denoted in equation 2.2.

$$\overline{X} = \frac{1}{N} \sum_{i=1}^{n} X_i \tag{2.2}$$

Variance and Standard Deviation

The sample variance is denoted as S and calculated as 2.3

$$S^2 = \sum_{i=1}^{n} \frac{(x_i - \overline{X})^2}{N-1} \tag{2.3}$$

Within the sample variance the divisor yields an unbiased estimator. The standard deviation is calculated by taking the root of the variance $\sqrt{S^2} = S$.

Confidence Interval

A confidence interval shows accuracy of an estimation by giving bounds (31). It is calculated as denoted below in equations, 2.4, 2.5 and 2.6. For the λ in equation 2.4, we commonly use 1.96 to denote a probability of 0.95.

$$a = \frac{\lambda\sigma}{\sqrt{N}} \tag{2.4}$$

And the confidence interval is then calculated by

$$-a + \overline{X} \leq X \leq a + \overline{X} \tag{2.5}$$

This results in

$$-\frac{\lambda\sigma}{\sqrt{N}} + \overline{X} \leq X \leq \overline{X} + \frac{\lambda\sigma}{\sqrt{N}} \tag{2.6}$$

σ is replaced by the mean standard deviation of the benchmark simulations, N. In Python, we use the Scipy package (32) to generate the λ value, given a certain percentage (usually 95).

Percentiles

To obtain a percentile, every measurement is ordered in ascending order. Then the 25th, 50th, 75th, 95th and 99th percentile can be identified. This is particularly convenient for latency, as the distribution of response time tend to have a very long tail.

3

Related Work

To assess the performance of Citus we comprise four different area's of research; database benchmarking, profiling distributed systems, database architectures and queuing theory. There exists some research that entail most of these research area's, but we first look into every area individually to extract useful techniques that we can use for the performance monitoring and the prediction of the behavior of Citus.

3.1 Benchmarking

Benchmarking databases has been done extensively to assess the performance and reliability of database systems. However, the rise of the cloud and adaptable databases slightly changed the way of benchmarking and its corresponding metrics. For example, the authors in (9) conducted experiments to assess the performance of several cloud-based data management systems such as Cassandra (33) and HBase (34). However, they specifically focus on the differences between several storage architectures and data models rather then performance (bottlenecks) of a single system.

3.1.1 Tooling

Cubukcu *et al.* (7) divided the most common workloads patterns that occur amongst Citus users into roughly four different types: *Multi-tenant* (Software-as-a-Service), *Real-time Analytics* (Customer-facing dahsboards), *High Performance CRUD* (Microservices) and *Data Warehousing* (Analytical Reports). From these four types, the High Performance CRUD workloads are targeted as they are expected to comprise most of the future workloads of Citus 11 users. Thus, for this research we will particularly focus on CRUD workloads as directly querying from worker nodes in Citus 11 is specifically beneficial for multiple small

reads and writes. Zhan *et al.* (35) list different tooling for benchmarking DBMS's and the type of use cases for which the tooling is suitable. For Citus 11 specifically, simple queries such as inserting, selecting, update or deleting are targeted workloads. These type of workloads, commonly referred to as CRUD workloads, fall under the OLTP-paradigm. According to Zhan *et al.* (35) an appropriate benchmark tooling suite for OLTP workloads are TPC-C and YCSB (10).

Pgbench

PostgreSQL itself offers `pgbench` as a standard benchmark suite, which is build-in tooling to run simple benchmarks in Postgres. With `pgbench` the user is able to specify the amount of queries executed against the PostgreSQL db, among with some other configurations. A standard pgbench benchmark consists of inserts, updates and selects. In the case of CRUD workloads, pgbench lacks specific functionalities such as loadbalancing which other benchmark tooling does offer. In addition, for extensive benchmarks pgbench is not widely used in research.

TPC

The Transaction Processing Performance Council (TPC) is the industry standard for transactions processing benchmarks. TPC states seven principles that creators of benchmarks should strive for. The benchmarks need to be (1) relevant, (2) understandable, (3) consist of good metrics, (4) scalable, (5) the coverage has to be sufficient, (6) there ought to be a common acceptance towards the benchmark and (7) portability (35). Following their own principles, they constructed `TPC-H`, a benchmark simulating a Decision Support System (DSS). DSSs often are read-intensive and involve complex queries with myriad concurrent modifications, while consistent results are expected for accurate business analysis (8). Some researches that extensively cover TPC-H are (36, 37, 38). Another TPC benchmark that arised is `TPC-C`. According to (39), TPC-C is the most widely used and intended for `OLTP` workloads. It consists of workloads that contain simple, short queries with frequent updates. TPC-C is used in (17) amongst others.

Yahoo! Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) (10) is a modular benchmarking tool that is often used to benchmark NoSQL databases. It enables to load (insert) data into a DBMS and on top of that it provides 6 different core workloads which are summarized in table 3.1. In their standard workloads, each row is roughly 1KB and consists of 10 columns,

each filled with 100 bytes. Moreover, YSCB provides options to create your own simulated workloads by adjusting the parameters accordingly. YCSB keys are randomly generated and thus are not correlated in some sort with each other.

Table 3.1: YCSB Core Workloads (Source: (3))

Workloads	Description
A: Update heavy workload	Mix of 50/50 reads and writes.
B: Read mostly workload	95/5% Mix of Reads/Writes
C: Read only	100% reads
D: Read latest workload	Traffic on recent inserts
E: Short ranges	Short ranges of records are queried
F: Read-modify-write	Client reads, modifies and writes a record

A workload could have a *zipfian* distribution (3.1), which is a discrete distribution often used to model rare events. It is a special case of the Power Law distribution, and in this case it means that some records in the DBMS are very popular and others not at all.

$$p(x) = \frac{x^{-(\rho+1)}}{\zeta(\rho+1)`} \tag{3.1}$$

Other possible distributions are *uniform*, where each record has an equal probability to be 'hot'. Lastly there is *latest*, where the latest records are 'hot'. YSCB provides load balancing through JDBC by setting `loadBalanceHosts=true`. This enables us to load balance all connections across multiple worker nodes in Citus to distribute the load. Note that when YCSB opens a connections to the database, this connection usually is kept open. This means whenever a large amount of connections is open, the penalty of context switching is significant in the case of a stalled transaction.

3.1.2 Frameworks

Benchmarking frameworks go a step beyond executing simple benchmarking workloads against a Database. There exists several frameworks that discuss effective ways to map the performance of a cloud-based or distributed systems (40, 41, 42, 43, 44). Most of these frameworks however focus on a specific part of benchmarking e.g. the benchmarking of query engine (42) or understanding query execution (44). Nevertheless, we can draw best practises from these frameworks and adapt them in a way such that they will suit to profile

Citus.

DIAMetrics

Although intended to benchmark query engines, DIAMetrics (42) is an end-to-end framework for performance monitoring at scale. It consists of several components that act as entry points and consist of the workload extractor, data scrambler, data mover, workload runner and system monitoring. The workload extractor mines query logs and generates a representative workload for testing. The data and query scrambler anonymizes data, the data mover moves data between different formats, the workload runner is the execution component that takes care of the actual execution of the workloads as well as the profiling of each query. It subsequently takes care of the storage of generated metrics for later analysis. The last component presents a report of the results to users and is able to alert users if an issue is detected.

PEEL

Peel (43) is an Open-Source [1] end-to-end framework that enables transparent benchmarking of distributed systems. It automatically orchestrates experiments, handles the set-up and the deployment of systems. In their paper, the authors benchmark a supervised machine learning workload to exhibit the use of PEEL and evaluate a system under test. PEEL organizes experiments in experiment suites, which can easily be started by using e.g. the PEEL CLI. The execution lifecycle consist of the (1) setup of the experiments, (2) execution of experiments and (3) tearing down the experiments. Once a workload execution is started, they make sure that only for specific parts of the benchmark the system performance is recorded on the involved nodes by setting up an additional monitoring system that is running in the background, e.g. `dstat`. When execution is finished, logs are gathered from all compute nodes and all temporary file systems on these nodes are cleaned and removed or replaced by new ones for a different configuration. To make sense of the gathered log data PEEL provides an extensible ETL pipeline which contributes in filtering relevant data form log files. Moreover, it enables to transform the logs and loads it into a database.

[1] `https://github.com/peelframework/peel`

3.2 Profiling

To get an accurate profile of a distributed database, we explore different profiling tools for distributed systems that may contribute to obtain an accurate profile of the behavior of Citus when executing benchmarks on a Citus cluster. There are several profiling tools from which some are specifically intended for distributed systems. The larger part of current available distributed profiling tools aim to sample threads that follow asynchronous paths to ultimately construct traces of these threads or processes. According to (43, p.2), a trace itself can be defined as a "set of spans that form a tree-like structure, typically arising from one request". Examples of tools solely intended to profile distributed systems are e.g. Dapper (45), Dprof (46), and Tprof (47) which are briefly discussed in 3.2.2.

3.2.1 Tooling

There is a large number of available tools for performance monitoring and analysis. To obtain an overview of a number of profiling tooling, Gregg (1) listed different (hardware) resources and corresponding tooling to measure utilization, saturation and errors for these resources [1]. We discuss some of these tools that are of interest to profile the behavior of Citus.

Vmstat

One of the most traditional tools to monitor system-wide resource utilization is `Vmstat` (48). This tool is capable of showing e.g. processes, memory, CPU and I/O utilization.

Htop

Htop is used for resource monitoring and offers a comprehensive UI to check the utilization per process. The general usage of htop is simple due to its convenient UI, but when using for monitoring this tool presents data in a way that makes it harder to parse for further analysis.

Iostat

Iostat is another tool used by (17) that monitors system performance by reading from the `\proc` filesystem in linux (49). It shows output of CPU utilization, disk I/O request and I/O waits. Moreover, on device level it is able to show e.g. reads and writes per second, MB written per second, average queue length, average waiting time and service time per

[1] `https://www.brendangregg.com/USEmethod/use-linux.html`

request.

Dstat

Dstat is a vmstat-like tool (50) which is used by (43) to monitor performance of a distributed system. Dstat is a basic tool that offers a wide range of settings that enable to e.g. show CPU, disk, network, process, swap and memory stats.

Perf

Perf is a popular Linux-based profiling tool which monitors the system through incrementation of counters and is particularly useful for tracing errors.

Bpftrace

Bpftrace (51) is another tool useful for investigating performance problems. Bpf stands for Berkeley Packet Filter, but despite its name Bpftrace is often used for tracing rather than for packet filtering. It attaches actions to kernel and userspace tracepoints and dynamic function probes. Tracepoints are specific points defined in the source code, while dynamic function probes perform an action when some code is executed which is not necessarily intended for tracing.

3.2.2 Distributed Tracing

Profiling tools often collect samples of events by making use of Performance Monitoring Units (PMUs) (52, 53) . These tools could be divided in roughly two types; intrusive profilers (45, 54, 55) and non-intrusive profilers. The former modifies source code of the application to gather data at runtime, which more likely leads to overhead and is often system specific. The latter often samples a thread or a process to extract useful information from this event, without modifying the source code of the program (56).

Dapper

In 2010 Google introduced Dapper (45), an intrusive profiler which functions as a tracing infrastructure with low overhead and is specifically intended for large-scale distributed systems. Dapper relies on two fundamental requirements of which (1) is ubiquitous deployment and (2) continuous monitoring. Moreover, they ensured that the tooling guarantees *low overhead*, *application-level transparancy*, *scalability* and fast availability of the gathered data (i.e. data should be available for analysis within a minute). They found that

to ensure low overhead, sampling is necessary and a single sample out of thousands of request most often provides sufficient information. Dapper essentially constructs 'Trace trees' where the tree nodes are referred to as *spans*, i.e. "basic units of work" (45, p. 3). The edges between different spans are indicating a causal relationship, and each span refers to its *parent span*. In Dapper, spans contain their *span id*, *parent id*, *trace id*, *span name* and different annotations that can be specified by the user through the Dapper API. When a sampled thread is traced, Dapper ensures to attach the above indicated *trace context* to thread-local storage which is (albeit partly) passed on during callbacks. This is essential for reconstructing traces that follow asynchronous paths. To ensure fast availability of data, Dapper first writes data to local log-files, subsequently pulls the data from the different sources and finally writes the data to Dapper Bigtable repositories to ultimately enable analysis.

Dprof

Dprof (46) arised as a new tool to construct profiles of distributed performance. In its core, it is a new *timestamp synchronization algorithm* that deals with the inaccuracy of timestamps of distributed systems. Dprof particularly focuses on synchronizing timestamps to ultimately construct accurate traces, which makes it a bit more challenging to adopt this tool for assessing the performance of Citus.

Iprof

Furthermore, Iprof (57) is a non-intrusive tool that reconstructs execution flows of requests by inferring from system logs. The authors evaluated their tool on four distributed systems, from which they tested Cassandra (33) and HBase (34) using YCSB benchmarks. They concluded that Iprof succeeded in inferring useful statistics from the logs produced by YCSB, with an accuracy of 95% for Cassandra and 90% on average for all four systems.

Tprof

More recently, Tprof was invented by Huang *et al.* (47) and takes profiling one step further by enabling *fine-grained aggregation* by focusing on similar trace structures. The authors argue that a trace or *span* is a timespan in which a task executes. Such a span is split into *subspans*, which each perform a specific task or access a specific resource. Its counterpart is *coarse-grained aggregation* which provides a more broad system view by aggregating many diverse traces and calculating the average or percentiles. This however means that it cannot provide a detailed view of traces which tprof does enable, since the structure

for traces differ (47). Tprof's workflow is (1) grouping all traces, (2) partition traces by request type, (3) grouped by span structures and ordering these structures.

LRTrace

Another non-intrusive approach of grouping traces is shown by Pi *et al.* (58), who propose keyed messages to reconstruct traces and called their tool LRTrace. This is, a uniform structure of log messages to address the challenge of extracting and reproducing logs in distributed systems. They essentially develop two major components for leveraging their keyed messages; a Tracing Worker and Tracing Master. A Tracing Worker runs on all machines of the distributed system and is responsible for locally collecting both logs and resource metrics. After collection of the data, the Tracing Master pulls the gathered information by all Tracing Workers and transforms this data into keyed-messages. Lastly, it stores the keyed-messages in a database for further analysis.

3.3 Bottlenecks

A bottleneck exposes a fundamental limitation in the performance of an application. The key to improving performance is knowing which elements to speed up. As a result, it is important to identify the limiting resource and determine if it is being used to its full potential. We distinguish two main resources as (1) *instruction execution* and (2) *data transfer bandwidth*. Some examples of common performance bottlenecks are load imbalance or bandwidth saturation. Other performance patterns that might be of interest are bandwidth limitations, non-cached memory access and synchronization overhead. The latter can occur due to e.g. locks protecting shared resources or speedup going down as more cores are added. Determining bottlenecks is rather difficult in Citus due to its distributed nature and we therefore aim for a methodological approach to identify these bottlenecks in the first place. For the distributed databases, some common bottlenecks are known to be latency (59) or memory bandwidth (59, 60).

3.3.1 Identification

Gregg (1) proposes the `USE method` (figure 3.3) to enforce to methodically think about performance and for identifying systemic bottlenecks. The author argues that for every resource, one needs to check utilization, saturation, and errors to assess the performance. With resource, they mean all physical server components which include, *inter alia*, CPU,

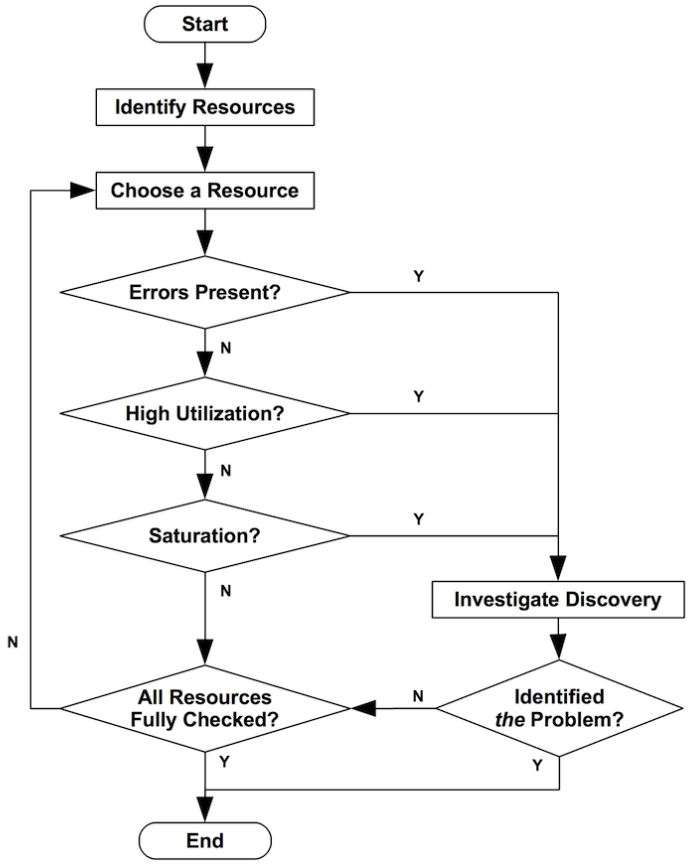

Figure 3.1: Flow Chart Diagram of USE-Method (1)

network and disk. With utilization of a resource they imply the percentage of time a resource is busy during a specific time interval, during which a resource often is able to accept more work. If a resource is saturated, the remaining work is most often waiting in a queue.

Gregg defines key metrics of the USE method as follows (1, p. 5):

- Utilization is defined as a percent over a time interval (e.g., one CPU is running at 90-percent utilization), or the percentage of the total capacity of a resource (e.g. main memory) that is used
- Saturation can be identified when measuring the run queue length
- Errors simply as the count of all errors in a specific resource

Another resource pointed out by Gregg that is of interest in this research is process/thread capacity. The system can be limited by the amount of processes, which is defined as utilization. The wait on the allocation of these processes (synchronization overhead) might indicate saturation and when the allocation fails errors may occur. In addition, when utilization is above a certain percentage, queuing will become more prevalent as the queues will be longer. This process can be modeled by queuing theory. In summary, the USE method requires to first compose a list of resources, then check utilization, saturation and errors of all of these resources (by using the Flow Chart in 3.3) and then, if a bottleneck is detected, analyze this bottleneck in-depth.

3.3.2 Understanding

Ousterhout (61) devoted her research on clarifying how observed bottlenecks can be understood for parallel jobs and parallel resource utilization by means of `blocked time analysis`. She argues that tasks usually consists of multiple subtasks, which she calls *monotasks*, that each utilize one specific resource, e.g. CPU, network or disk. The blocked-time in this case implies the time spent waiting for a monotask to access a particular resource. By focusing on blocked-time for each monotask, the optimal performance of a single task can be approximated by subtracting the blocked time from the total execution time of one task. This way, the theoretically optimal runtime could be estimated if a specific resource was infinitely fast. Ousterhout also focuses on straggler tasks which she defines as “a task that takes much longer to complete than other tasks in the stage" (61, pp.32). A straggler task is always part of another task, meaning that this ultimately slows down the overall execution of a total task or query. By blocked-time analysis one can expose the maximum gain (in terms of speed) for a task, if a resource would be optimized maximally. Moreover, if straggler tasks are identified one could possibly understand identified bottlenecks that slow down execution. Ousterhout states that measuring actual blocked-time is challenging, but that we are able to measure how many data is read and how long it took. This way one is able to estimate the bandwidth or disk I/O. More specifically, she argues that one is able to obtain complete metrics about the time a monotask spends on a resource if we combine existing per task I/O counters (e.g. shuffle byte read) with machine level utilization metrics.

3.3.3 Applicability

A single query executed against a Citus cluster could be considered as a thread or a job that consists of one or multiple *monotasks*. Namely, in order to complete, a query ought to access different resources such as network, CPU and disk on or between one or multiple nodes in a Citus cluster. If we list all resources that are accessed by a single Citus query and subsequently apply the USE method (1), we can identify potential resource bottlenecks in a system under test. Thus, we split the execution of a single Citus query up in different monotasks that each require a specific resource. To take this one step further, by measuring the blocked time of each monotask belonging to a single Citus query, we could identify straggler causes within the execution of a query that potentially decreases the ultimate throughput on a large Citus cluster. With an indication of the blocked time we could estimate the theoretically maximal throughput on Citus 11 if no resource bottlenecks occur.

3.4 Scalability

The exact definition as well as the evaluation of scalability of distributed systems is frequently discussed. According to (62), scalability could be defined as the throughput and response time divided by the cost factor. Kuhlenkamp *et al.* (11, p. 1220) classified three different types of scalability benchmarks; (1) Change load between subsequent workload runs without changing system capacity, (2) Change system capacity between subsequent workload runs without changing load, (3) Change system capacity and change load proportionally between subsequent workload runs. In addition, Kleppman (18) argues that assessing scalability can be viewed in two ways: 1) increasing load and keeping resources constant and 2) increasing load and increase resources accordingly to keep the performance unchanged. For this research, we will construct experiments that are based on the variations of load and system capacity (cluster size) as described in the 3 different types of scalability benchmarks by Kuhlenkamp *et al.* (11). The authors in (63) take the cost factor into account and present a new framework to assess scalability of a distributed systems. However, for this research, the cost factor is out of scope although it is an important factor. We instead focus on some older commonly used metrics:

- Speedup S: $S(k) = k$, where k is the number of workers
- Efficiency E measures the work rate per worker $E(k) = \frac{S(k)}{k}$

- The scalability from scale k_1 to scale k_2 is the ratio of efficiency: $\psi(k_1, k_2) = \frac{E(k_2)}{E(k_1)}$

However, in their paper (63) they state that these metrics are not suitable for distributed systems, because these systems bring new dimensions of complexities. Therefore, they propose a new framework to numerically assess the scalability of different configurations of distributed systems. Although it is not our aim to numerically explore different configurations, we will use some of these metrics to assess the scalability of Citus 11.

3.5 Queueing

Queuing models are often used to assess performance bottlenecks in complex systems such as a DBMS. Delis & Roussopoulos (64) describe queuing network models for different DBMS architectures; standard client-server, RAD-Unify and enhanced client-server architecture to identify performance related issues for these architectures under different types of workloads. Qiang et al. (65) present an implementation of a performance forecasting framework for a DBMS by means of constructing a Layered Queuing Network (LQN). With this queuing network, they focus on performance forecasting and aim to understand the impact of hardware selection and software structures for a DBMS. In addition, they verify whether the behavior of their implementation resembles real systems.

Another researched is performed by Dipietro et al. (2), who proposed a novel queueing network model for Cassandra to map resource provisioning. Their model explicitly defines configuration parameters which makes it possible to compare multiple setups. Due to its distributed nature, Cassandra resembles the architecture of Citus 11 in myriad aspects, such as the fact that clients can connect directly to a particular node. A difference however is that Cassandra ultimately is a NoSQL database as opposed to Citus.

The most appealing from their research is that the authors modelled YCSB workloads to assess the performance of their queueing network. In their case, they modelled the YCSB workload generator as two separate queues that represent network and CPU, where the YCSB workload CPU generates new requests which are sent through the network queue (figure 3.2). Once an executed query returns at the YCSB generator queue, a new query is generated. To distinguish different types of request, Dipietro et al (2) modelled two different types of requests: a read-data from disk request (local request) or retrieve data from another node (remote request).

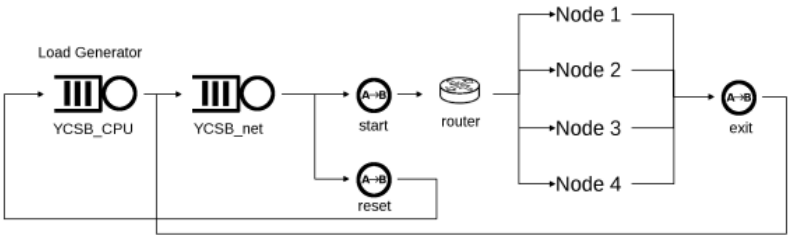

Figure 3.2: High-level overview of Cassandra YCSB architecture (source: (2))

Osman et al. (66) designed a clear framework to evaluate the performance of different database designs. The most appealing about their research is that they attempted to model queues for accessing tables and have clear definitions of parts of the queueing system. As extensively described in Alomari et al. (67), queuing networks are classified in two types: open and closed. The difference is that in a closed network the same amount of jobs always reside in the system. Alomari et al. examine fork-join (FJ) queuing networks and argue that for a client-server system with a known number of clients and a heavy workload a closed queuing network is suitable, which means that a finite number of jobs is cycling through the system.

In the case of a RDBMS the workload intensity is expressed as the amount of transactions (txn), which can be be mapped to the number of customers N in a queuing network that demand one or more *jobs* (67). To account for different types of workloads in a real RDBMS, we could compose a multi class queuing network where we model different classes of customers $N \in \{\overrightarrow{N_R}, \overrightarrow{N_W}\}$ where $\overrightarrow{N_R}$ represents a vector of read transactions and $\overrightarrow{N_W}$ represents a vector of write transactions. In a multiclass model each txn can be routed differently, depending on the resources available and needed for the txn to commit. Moreover, in the case of Citus we could model the system as a homogeneous FJ where the distribution of service times at parallel queues is similar.

4

Benchmarking

To benchmark and compare the performance of Citus 11, AWS Aurora and Google Spanner, we will make use of YCSB benchmark tooling. First, we benchmark Citus to get a general picture of its performance on Azure Managed Service, followed by benchmarking Citus when we directly use the control plane software abstraction (Marlin) that manages and provisions Citus cluster. At last we do a performance comparison between Citus, Aurora and Spanner.

4.1 Methodology

Citus has been extensively benchmarked by HammerDB [1] and a TPC-C workload. However, HammerDB simulates a workload that resembles queries used in a warehouse-like system and measures their throughput in New Orders Per Second (NOPS). Citus 11 however is expected to increase the throughput of simple writes and reads. This type of workloads can be categorized as create, read, update & delete (CRUD) workloads. Hence, for benchmarking Citus, AWS Aurora and Google Spanner we are using YCSB (see 3.1.1).

Execution

To minimize the time-consuming process of manually executing numerous benchmarks, we automate the initialization of a new infrastructure based on the Citus benchmark-suite [2]. While using the Citus Benchmark `Azure` module, we initiate a new Citus cluster and

[1] https://www.citusdata.com/blog/2022/03/12/how-to-benchmark-performance-of-citus-and-postgres-with-hammerdb/

[2] `https://github.com/citusdata/citus-benchmark/`

a Driver VM respectively for testing and coordinating the benchmarks. This VM automatically executes multiple iterations of different YCSB workload configurations using straightforward scripts for initialization 4.1. The Azure module itself consists of a few bash scripts and `.bicep` files which enable the automatic configuration of Citus clusters and Azure VM's. To run YCSB, a few parameters (e.g. iterations, shard count and thread counts) are added to these files to pass on to the benchmark script that runs on the Driver VM. Once the new cluster and driver are allocated in the Azure cloud, a few Linux and YCBS specific packages as well as a JDBC driver are installed through a `cloud-init` script upon initialization of the Driver VM. To connect with the database, we make use of the Java Database Connectivity (JDBC) driver. This is an API that enables to connect with and execute queries against a database. JDBC is supported by YCSB and enables the load balancing of connections across multiple database instances, herewith supporting concurrent query executions. By distributing connections across multiple worker nodes, we are able to distribute the load of concurrent queries among workers. The load balancing provided by JDBC however is not necessarily in a round-robin fashion. Instead, the JDBC driver lists suitable worker nodes and then randomly chooses one of the candidates. This most often leads to an unequal distribution of direct connections from the driver across worker nodes.

```
python3 benchmark.py --records=1000 --threads=100,200 run_all_workloads
```

By making use of the Python package `Python Fire` [1], we provide scripts that automates the execution of different YCSB runs. The runs are performed in the order as described in the YCSB Wiki page for core workloads [2]. The algorithm used for running the workloads in the required order is shown in algorithm 1. We iterate multiple times across different workload configurations to obtain statistical significant results of multiple benchmarks runs.

To run the actual benchmarks, we make use of `tmux` such that the execution takes place in a distinct process. `Tmux` is realiable in the sense that it resumes with the benchmark if the `ssh` connection with the driver suddenly drops. The loading and running of the workload itself by YCSB is called by the benchmark script (`run_workload(`w_n`, r, o, `c_k`)`), but

[1] `https://github.com/google/python-fire`
[2] `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads`

Algorithm 1 `run_all_workloads()` function

Require: *Ordered* list of YCSB workloads W
Require: List of different thread counts (connections) C
Require: Records r
Require: Operations o
  for iteration i **do**
    for threads $c_k \in C$ **do**
      `truncate_table()`
      `load_workloada(`w_a`,` r`,` c_k`)`
      for workloadn $w_n \in W$ **do**
        if $w_n == w_e$ **then**
          `truncate_table()`
          `load_workloade(`w_e`,` r`,` c_k`)`
        end if
        `run_workloadn(`w_n`,` r`,` o`,` c_k`)`
      end for
    end for
    `generate_output()`
  end for

executed in a bash script (4.1).

```
bin/ycsb load jdbc \
    -P workloads/$WORKLOAD \
    -p db.driver=org.postgresql.Driver \
    -p recordcount=$RECORDS \
    -p threadcount=$THREADS \
    -cp ./postgresql-42.2.14.jar \
    -p db.user=$PGUSER \
    -p db.passwd=$PGPASSWORD \
    -p db.url="jdbc:postgresql://$CITUS_HOST/$PGDATABASE?loadBalanceHosts=true" \
    | tee ${HOMEDIR}/${OUTDIR}/load_${WORKLOAD}_${THREAD}_${RECORDS}_${ITERATION}
    _${WORKERS}_${RESOURCE}_${DRIVERS}_${PART}.log
```

When the benchmarks are finished, a `csv` containing the throughput and total execution time per workload configuration is generated on the driver VM from the raw YCSB logs. This CSV is subsequently transferred to the local machine using `scp`.

Benchmarking on Marlin

To have the possibility to tune more parameters and exceed standard Azure settings such as the`max_connections` which is set to 1000 by default, we enabled benchmarking on Marlin. Marlin is the control plane software to provision Citus clusters and hosts Azure Managed Service. By modifying the utilities (`utils.rb`) and config file (`config.rb`) we incorporate YCSB specific parameters in Marlin. The YCSB specification file (`ycsb_spec.rb`) orchestrates the end-to-end execution of the benchmarks. This process initiates the Citus cluster and builds a `config.yml` file in which information about the cluster is stored. Moreover, the process calls the `.bicep` files that spins up the driver VM and takes care of pre- and postprocessing tasks for the benchmarks.

Expected bottlenecks for Citus

A practical implication of Citus 11 is that when the client opens many connections to the Citus cluster, the amount of incoming connections for every worker equals the amount of connections made by the client to the entire cluster. Since the the CPU only can handle a limited amount of concurrent processes (i.e. connections), and since every connection consumes memory, this will cause CPU saturation and Out of Memory (OOM) errors. CPU Saturation and OOM errors ultimately negatively influence the response times of an executed query due to the introduced overhead.

Evaluation

To evaluate scalability can make of different evaluation formulas (63).

- Speedup S: $S(k) = k$, where k is the number of workers
- Efficiency E measures the work rate per worker $E(k) = \frac{S(k)}{k}$
- The scalability from scale k_1 to scale k_2 is the ratio of efficiency: $\psi(k_1, k_2) = \frac{E(k_2)}{E(k_1)}$

Speedup is most commonly used and a straightforward indicator of the performance, so we will mainly use this metric. Since we are merely interested in OLTP workloads from which the performance often is measured by the transaction throughput, we measure the amount of transactions per second, but it can also be expressed in a time-interval $T \epsilon \{t_0, ..., t_n\}$ which is calculated according to the formula in 4.1.

$$TP_{t_1 ... t_n} = \frac{\sum_{i=1}^{j} Tr_i}{t_n - t_0}. \tag{4.1}$$

4.2 Experiments and Results

Experiment set-up

The very first experiments are executed on *Azure Managed Service* which is referred to as Citus in production. The Driver VM used consist of 64 vCores and is a `Standard_D64s_v3` machine, while the coordinator and worker nodes all contain 16 vCores and consist of a `Standard_E16ds_v3` machine. In these first experiments the default shard count is set to 48 since this value is divisible by many numbers, the Citus version used is **10.2**, PostgreSQL version is **14** and we benchmark Citus using YCSB `workload a` for inserts and `workload c` for reads.

Citus in Production

The most accessible way to use Citus is through Azure Managed Service that manages the Citus cluster for the customer. There are a few limitations for the Managed Service, such as a maximum amount of `max_connections` and the creation of more PostgreSQL users which disabled by default. We perform initial experiments on Citus in production to investigate the current limitations for customers. More specifically, we explore the behavior of the cluster for different thread counts and cluster sizes. Since Azure limits the amount of connections with the Citus cluster to 1000, we vary the amount of connections between `200, 400, 600, 800 and 990` in the initial experiments. The amount of nodes we vary is `4, 8, 16, 24`. The shard count is set to 48. We test the Citus cluster under YCSB workload A (100% `INSERT`) and workload C (100% `SELECT`), by 2 million inserts and 20 million operations (i.e. reads) respectively.

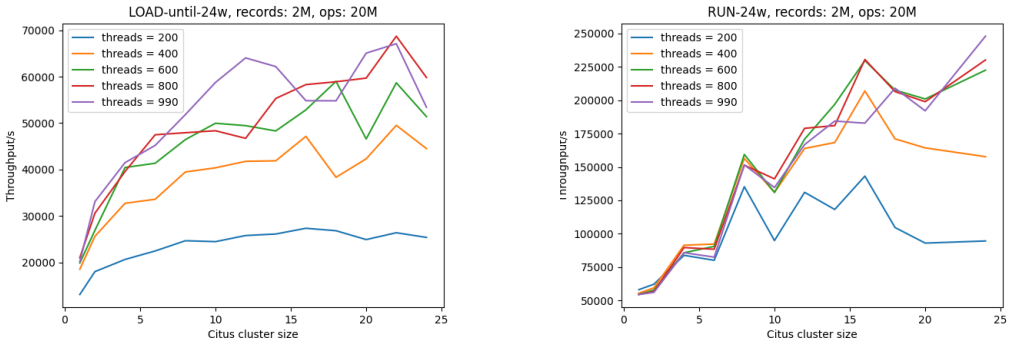

Figure 4.1: Transactions per Second (TPS) measured by YCSB for Citus in production, varying thread counts and cluster sizes. Driver VM: `Standard_E16ds_v3`, worker nodes with 16 vCores, 2 million operations. INSERTS (left), READS (right)

The graphs in figure 4.1 exhibit that the number of connections are becoming more important when the cluster size grows. This is particularly evident in the 100% `SELECT` workload.

Optimal amount of threads

The graphs in figure 4.1 show that for both `INSERT` and `SELECT` queries the optimal amount of threads is 990 and 800 respectively for a single iteration. Therefore, we do the resulting experiments in this chapter with 990 connections for 100% `INSERT` workloads and 800 connections for 100% `SELECT` workloads. The throughput for inserts is lower then for reads, which is expected. For each insert a new tuple containing information about the record to be inserted is appended to the WAL file. This file is subsequently flushed to disk, which is an extensive operation. Only after the WAL file is written to disk, the transaction is committed and a new query is fired through the same connection. Waiting for the WAL being written to disk often stalls the transaction for a few milliseconds, which slows down the total execution time of the transaction. For reads however, no write to WAL and thus waiting for disk I/O is required. For a `SELECT` transaction however, fetching data from disk might be necessary if not all data fits in RAM. Note that in these experiments we test a workload 2M inserts on a cluster where each *individual* worker node contains 128GB of RAM. Every insert on YCSB is around 1KB. This means that for this experiment we have 2 Million KB records, which translates into 2GB. 2GB fits easily in RAM, so for these experiments no data pages need to be fetched from disk. Note that a workload of 2 Million operations is not large, thus when the workload increases the behavior of the Citus cluster will differ.

Scalability

To assess the scalability of Citus in production, we perform benchmark runs with a thread count of **800** for reads and **990** for inserts. Moreover, we vary the amount of workers between `5, 10, 15, 20, 25` and `30` to examine the scalability. The results are depicted in figure 4.2. The graphs exhibit that the throughput for 100% `INSERT` does not really scale well. The difference between 5 and 20 worker nodes is about 50.000 transactions per second. This means that for a scale factor of 4, the speedup is only around $\frac{125000}{75000} = \frac{5}{3}$. For workload c, the 100% `SELECT` workload, the throughput increases significantly from 5 to 20 worker nodes. However, after the peak at 20 worker nodes of around 500.000 transactions per second, the TPS decreases and then seems to stagnate. This means that the peak for Azure Managed Service for reads is around 20 worker nodes with 800 connections. For

inserts, the scalability in production is bound by `max_connections`.

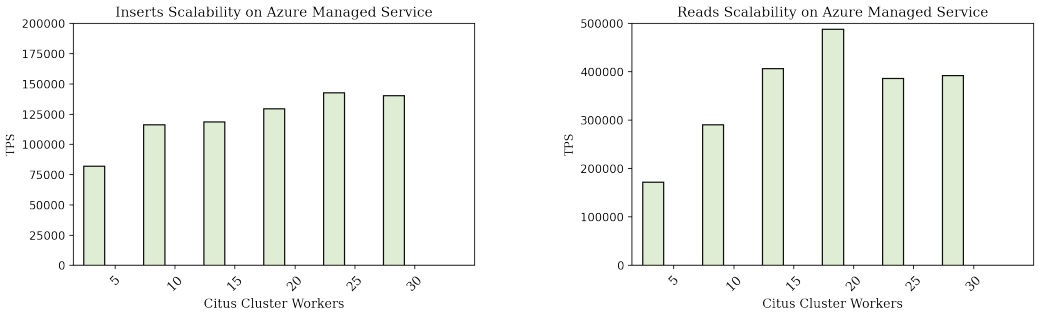

Figure 4.2: INSERTS (left) and SELECTS (right). TPS for Citus cluster of varying worker nodes, shard count of 2 × worker nodes, 100M reads on 100M records, 800 and 990 threads respectively and a single Driver VM of 64 vCores

Marlin

A Citus cluster deployed on Marlin enables more fine-grained tuning of different PostgreSQL and Citus settings. This entails that with deploying from Marlin directly enables to scale the amount of connections up to 10000 as opposed to the managed service, who limits the amount of connections to 1000. Therefore, the performance of Citus when directly deployed from Marlin is significantly better as shown in figure 4.3.

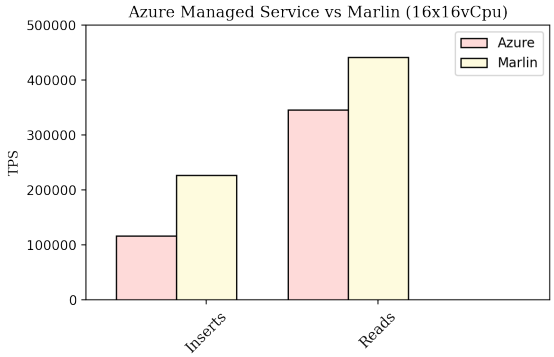

Figure 4.3: Performance of Managed Service (production) vs Performance of Marlin. Both consists of Citus clusters of 16 worker nodes with each 16 cores

V3 vs. V5 hardware

Azure provides VMs that run on different kind of hardware. Citus in production by default used V3 hardware until it completely transferred to V5 table 4.1. In June 2022 all V3 hardware has been updated to V5. This too results in a significant difference in the

Table 4.1: Specifications for V3 and V5 hardware used for VMs in Azure

Configuration	Standard_E16ds_v3	Standard_E16ds_v5
vCPU	16	16
Memory (GB)	128	128
SSD GiB	256	600
Max data disks	32	32
Uncached disk throughput (IOPS/MBps)	25600/384	25600/600
Max NICs	8	8
Max network bandwidth (Mbps)	8000	12500

performance of Citus for YCSB workload a and workload c. To investigate this difference we performed multiple runs to compare V3 and V5 hardware. The results are shown in figure 4.4 and exhibit a performance increase from $> 50\%$ for `SELECT` workloads. In addition, the performance increase for `INSERT` workloads is also significant, although the difference in performance is around 30%.

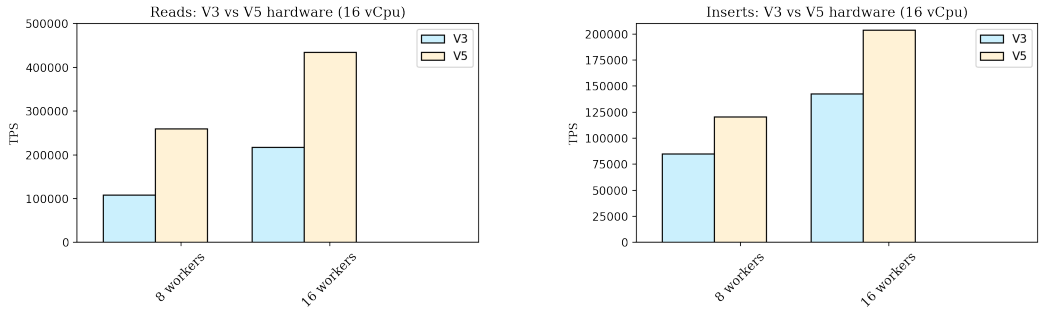

Figure 4.4: Performance in TPS on V3 and V5 hardware for read-only workload for 8-node and 16-node Citus clusters (left), Performance on V3 and V5 hardware for insert-only workload for both 8-node and 16-node Citus clusters (right).

Optimal amount of threads

Due to PostgreSQL and Citus being single threaded, the only way to execute multiple queries concurrently is to increase the number of connections to the database cluster. However, at some point the overhead introduced by maintaining these connections will exceed the benefits of having multiple connections. To assess these limits for Citus on Marlin, we first performed some experiments to get an indication when the performance drops if the amount of connections increases beyond that point. The results show that the performance significantly drops for a 100% `SELECT` workload when there are $>$ 800 connections to the entire cluster (figure 4.5).

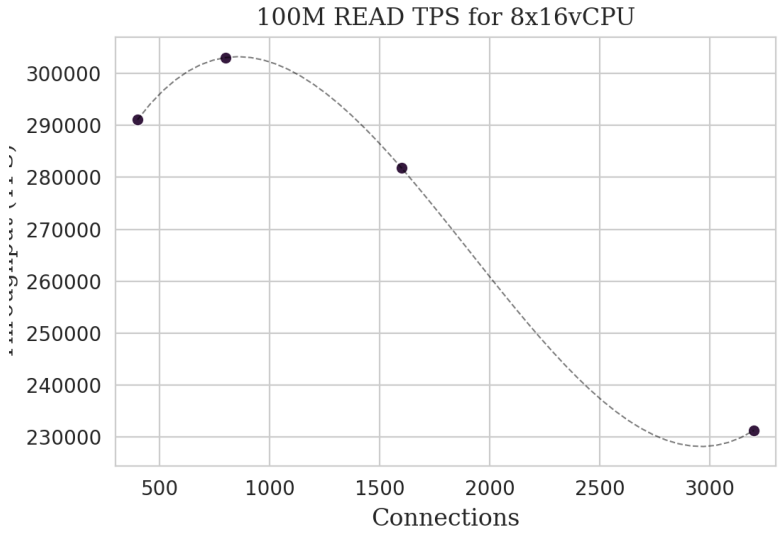

Figure 4.5: Y-axis is Transactions per Second (TPS). Graph shows TPS for SELECT workload when different amount of connections are used. Machine used: `Standard_E16ds_v5`, 2TB SSD, 128GB RAM, 100M records. Amount of connections on x-axis, throughput in transactions per second on y-axis. Amount of connections are doubled every iteration: 400, 800, 1600 and 3200

For the 100% `INSERT` workload there is no drop in performance (figure 4.2). The issue that arises with this workload is that beyond 3600 connections out-of-memory errors start to occur frequently. This could indicate that the performance of a 100% `INSERT` workload is limited by the available memory as we cannot upscale the amount of connections to the Citus cluster.

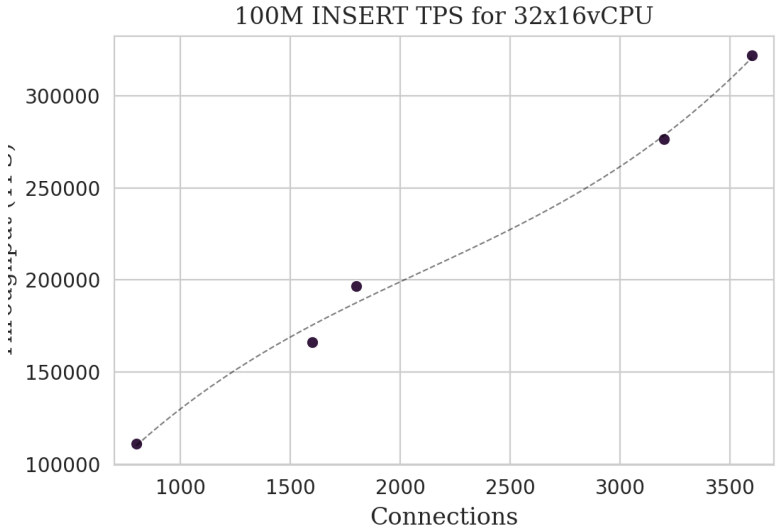

Figure 4.6: Y-axis is transactions per second. TPS for INSERT workload. Machine used: (`Standard_E16ds_v5`), 2TB Disk, 128GB RAM, 100M records. Amount of connections on x-axis, throughput in transactions per second on y-axis. Amount of connections are 800, 1600, 1800, 3200 and 3600

Increasing workload intensities

To test whether the workload influences the TPS, we tested different workload intensities on 32x16vCpu clusters. The results are depicted in table 4.2. Increasing workloads increase the TPS overall. This could mean that lighter workloads are more expensive, due to the overhead introduced by the first few executed queries that will take longer. If the workload becomes larger the relative fraction of queries necessary to 'warm-up' the cluster is smaller, leading to a better average TPS. Note that we did not test workloads greater then 500 Million transactions. We however expect that at some point the behavior of the Citus cluster will differ since not all data will fit in RAM leading to increasing disk I/O. Ultimately the cluster will become saturated if the workload intensity keeps increasing.

Scalability

Since the optimal amount of connections that does not result in out-of-memory errors for inserts and reads seems to be 3600 and 800 respectively, we use those numbers for our resulting experiments. Figure 4.7 shows that the performance of Citus clusters both increases for reads and writes, but the performance increase curve is flattening out beyond 16 workers. As a result, we conclude that for Citus clusters with 32 worker nodes the throughput does not scale well.

Table 4.2: Throughput, mean and 99th percentiles query execution times for 100 million `INSERT` queries without driver-to-cluster latency, cluster-to-driver latency and YCSB overhead, 3600 threads, 32 workers, 16 vCPUs

Workload	TPS	Mean (ms)	99th pctl. (ms)	YCSB latency (ms)
1 Million	41203	10.11	27.29	12.59
10 Million	165978	5.70	18.82	11.69
100 Million	294601	5.10	21.68	11.34
500 Million	311992	8.92	51.71	11.18

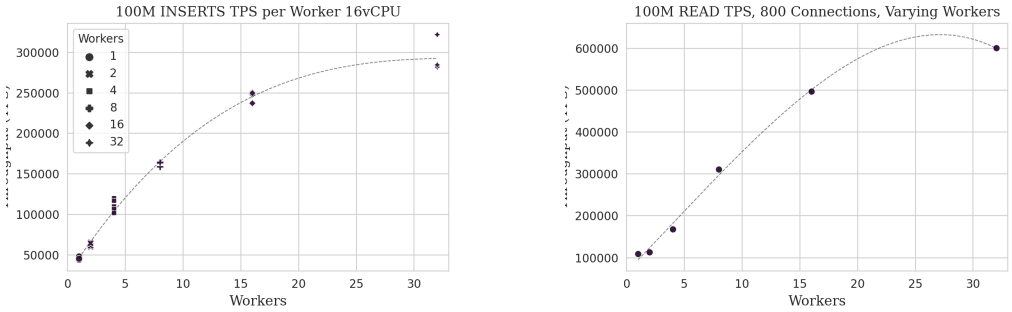

Figure 4.7: Y-axis shows Transactions Per Second (TPS). TPS for SELECT workload (800 connections, right) and TPS for INSERT workload (3600 connections, left) for different amount of workers. Machines used: (`Standard_E16ds_v5`), 2TB Disk, 128GB RAM, 100M inserts.

4.3 Performance Comparison

Aurora

To evaluate the performance of AWS Aurora we spin up an EC2 instance which functions as a driver VM. Moreover, we configure an EC2 instance by installing scripts for YCSB to work. We created simple bash scripts to automatically install and configure Aurora instances, wich are automatically created when a YCSB benchmark (batch)-run is called by the Python Fire scripts module.

Experiments and Results

Aurora does not enable to scale out by simply adding workers as opposed to Citus. Instead, Aurora offers large machines. This means however that when the amount of compute

required to handle a certain workload exceeds Amazons' largest machine, the client needs to extend its infrastructure by using more machines. This also requires that, if one wants to scale beyond Auroras largest machine, one has to *manually* shard the workload across different Aurora instances. For the benchmarks a C5.18x large EC2 instance (72 vCores) as driver is used. Furthermore, a db.r5.24xlarge RDS instance with PostgreSQL (96 vCores) is set up. The results of the benchmarks of Aurora are depicted in figure 4.8.

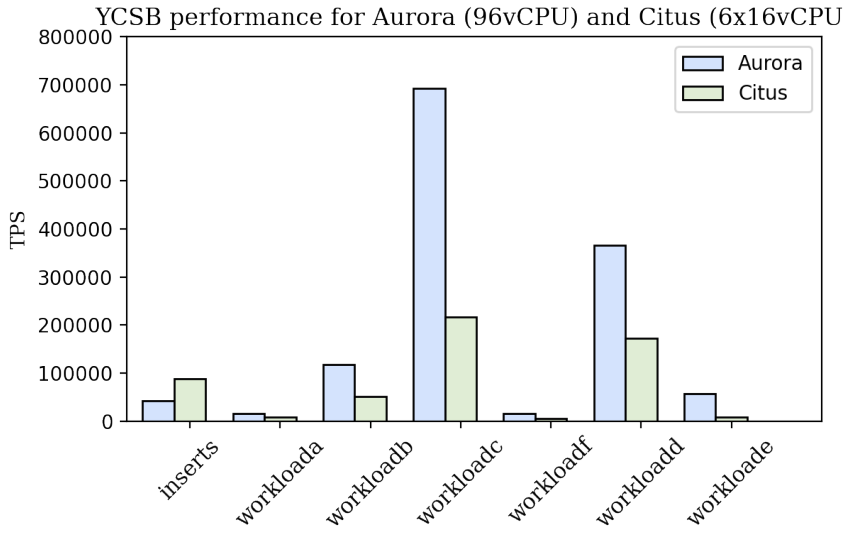

Figure 4.8: Results of Aurora vs Citus for the same amount of Vcores. For Aurora around 700 connections are used, for Citus 3200. All YCSB workloads (table 3.1).

The first thing we noticed while doing experiments that Auroras benchmarks results are very consistent, meaning that the throughput reported by YCSB does not vary significantly between different runs. In addition, the amount of connections does not influence Auroras throughput greatly, but the best performance is obtained around 700 connections with the cluster. For the 100% `INSERT` workload Auroras maximum throughput approaches 45000 TPS. The maximum TPS of the 100% `SELECT` workload however is around 700000 TPS *for one instance*. Aurora enables to add up to 16 read replicas, so the performance of reads is expected to reach around $16 * 700000 = 11.2$ million TPS.

Compiled Queries in Aurora

Aurora does not support the regular JDBC driver, but instead AWS provides their own Amazon Web Services JDBC Driver for PostgreSQL[1]. This could mean that AWS sends

[1] `https://github.com/awslabs/aws-advanced-jdbc-wrapper`

`PREPARED` statements which accelerate the execution speed of queries. With `PREPARED` statements the query is only parsed once and the specific queries of the values are passed in the form of parameters. To investigate this we eliminated `?loadbalancehost=true` and compared the throughput with and without this parameter. The throughput for reads significantly decreased with around 200 thousand transactions per second, meaning that the maximum `SELECT` throughput is now approaching 500000 TPS. 500k TPS however is still roughly twice as much throughput as Citus obtains with a similar amount of vCores.

Why is Aurora so much faster for a read-heavy workload?
Standard PostgreSQL evicts data pages from cache if they are dirty, subsequently write these pages to disk and then load a new page into cache. Aurora outperforms Citus in YCSB workload c (100% `SELECT`). This could be explained by the fact that *only* the storage layer takes care of keeping durable pages up-to date while network I/O is minimized, which enables to use a different replacement policy. Instead of writing a dirty page to disk Aurora only has to request a newer version of the page to load it into cache, which highly speeds up the process and results in a better read performance. However, in these experiments Citus does not have to access disk since all data fits in RAM, so the difference in performance is likely due to other bottlenecks in Citus. These bottlenecks are potentially the latency introduced by hopping the network to a different node, coordinating the distributed query, calculating the hash of a shard, context switching overhead due to dealing with many different connections or copying and swapping data pages from the OS cache to main memory and vice versa. Aurora might have introduced a smarter eviction policy for pages into cache, and relies less on the OS for swapping data pages.

To investigate why Aurora performs so much better compared to Citus with respect to a read-heavy workload, we will investigate what happens within Citus during a read-only workload by sampling and profiling distributed queries in a SUT. With profiling we can obtain an indication in which fragment a distributed query spends most of its time. By indicating these straggler fragments, we could optimize these fragments to ultimately improve the performance of Citus.

Spanner
Google offers a PostgreSQL interface for their Cloud Spanner instances. To communicate with a Spanner instance using psql, Google developed and open sourced PGadapter. This is a PostgreSQL API that enables the client to communicate with a Spanner instance.

Since the PostgreSQL interface is available as of april 2022, it still lacks support of some basic PG features and datatypes. This became apparent when executing YCSB workload E which involves `SCAN` operators. Similarly as for the benchmarks with Aurora, for Spanner we spin up a Compute that functions as a Driver VM for coordinating and executing the YCSB benchmarks against the Spanner cluster. To be able to use the PostgreSQL interface for Spanner, a PostgreSQL API (PGadapter [1]) is required which functions as a proxy between the psql client and the cluster. Thus, for automating the benchmarks we run PGadapter in a seperate tmux session when the YCSB benchmarks are initiated. We use similar scrips for the automation of YCSB benchmarks as we did for AWS and Azure.

Cloud Spanner can in theory scale indefinitely by adding nodes. One node consists of 1000 processing units; the exact definition of a processing unit, however, is unclear. Spanner is particularly highly scalable and easy to use. Spanner clusters are configured and available almost instantly.

Experiments and Results
To benchmark Spanner with a PostgreSQL interface, a Spanner instance containing 1, 8 and 16 nodes is created. We execute all different YCSB workloads and compare the results to Citus.

Since the exact definition of a compute node in Spanner is not clear, we compared the results of Spanner to a Citus cluster of 6 nodes with 16 vCpu's, 128GB RAM and 2TB disk (figure 4.9). Spanner reaches only a TPS around 3700 for workload A (100% `INSERT`) and around 6400 for Workload C (100% `SELECT`). In addition, as opposed to Aurora, the throughput reported by YCSB for Spanner is very inconsistent. Another thing we find that adding any new nodes does not increase the performance in terms of throughput for Spanner.

The performance of Spanner can be increased by inserting in batches. But apart from some 'best practices' reported by Google, there is not much guidance on how to improve the overall performance. Spanner does however have a warm-up time, which is not taken into account while running these benchmarks. In conclusion, Spanner with a PostgreSQL interface does not perform well on YCSB benchmarks with this set-up.

[1] `https://github.com/GoogleCloudPlatform/pgadapter`

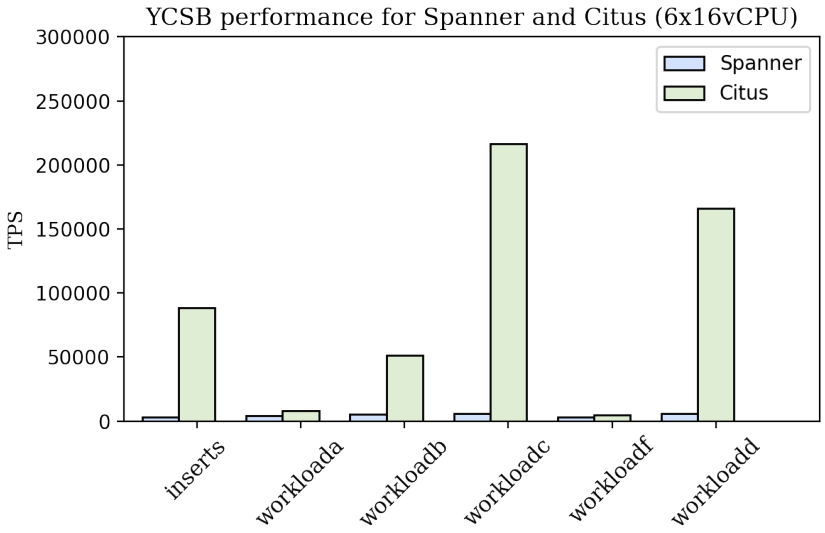

Figure 4.9: Benchmark results of Spanner vs Citus. All YCSB workloads (table 3.1)

Comparison with Citus

Aurora performs better on almost all YCSB workloads compared to Citus. The only exception is the 100% `INSERT` workload. For the 100% `SELECT` workload Aurora reaches a throughput from around 700k TPS. This value is not even approached by a Citus cluster containing in total $32 \times 16 = 512$ vCores. Aurora's largest machine contains 'only' 96 vCores and is thus highly efficient for YCSB workload c. Google Spanner, however, performs worse on all YCSB workloads. A possible explanation for this poor performance is due to *not* considering warm-up time for the experiments. Another aspect that could lead to a poor performance is the required PGadapter. The PG API translates all PostgreSQL queries to a format interpretable by Spanner. The translation of the queries comes with a performance penalty. In addition, while Spanner may not seem to perform well on YCSB workloads, it could perform well on different workloads. For example, workloads that are more analytical and consist of `JOIN`s.

5

Profiling

In the previous chapter, we conclude that Citus does not scale well at 32 worker nodes. With 32 worker nodes, only a speedup of 6.5x for `INSERTS` is achieved compared to a single-node Citus cluster. To get an indication what actually happens when running large Citus clusters, we examine the internal behavior of Citus during benchmark runs. To monitor and assess the behavior of Citus, we create an infrastructure that automatically executes YCSB workloads and collects the generated YCSB output and captures the internal behaviour. We build an infrastructure such that workloads can be executed automatically using this framework. This framework consist of roughly three components; the control plane `Marlin`, the `Citus-benchmark` repository which is executed on the driver and code from the Citus-benchmark repository that is executed locally. The local processes are used for monitoring the Citus cluster, parsing, storing and analyzing gathered data.

5.1 Benchmarking Infrastructure

For monitoring the SUT, we build a framework that captures and gathers data of Citus. The execution workflow or the pipeline structure of this monitoring framework is explained in this chapter.

Marlin

Marlin is the control plane in Microsoft that allows to spin up and customize a Citus cluster. This repository is mostly written in `ruby`. If one wants to enable particular PostgreSQL and Citus specific settings, it is most easiest to modify the source code of Marlin. Compared to Azure Managed Services, Marlin has the possibility to tune the cluster with more settings,

for example a higher amount of maximum connections. To enable to run benchmarks with YCSB on Citus clusters, a few input parameters and functions have been added to the source code. These additions allow to automate the benchmarks while defining specific YCSB settings and cluster configuration settings. These inputs are depicted in table 5.1.

Table 5.1: Input parameters for benchmarking in Marlin

Parameter	Type	Explanation	Example value
`name`	string	name of benchmark	32w32-s-citus-0-exp
`cores`	int	worker cores	16
`master_cores`	int	master cores	16
`workers`	int	amount of workers	32
`master_disk_quota_mb`	int	disk size (mb)	2048
`worker_disk_quota_mb`	int	disk size (mb)	2048
`operations`	int	operations YCSB	100 Million
`records`	int	records YCSB	100 Million
`shard_count`	int	shards in cluster	64
`thread_counts`	list<int>	connections	800,3600
`iterations`	int	iterations	5
`drivers`	int	driver VMs	1
`parallel`	bool	monitored benchmark run	true
`custom_configs`	bool	use custom configurations	true

Execution Pipeline

The execution pipeline of the benchmarking infrastructe is shortly explained below. We will dive deeper into the different modules used after explaining the pipeline.

1. **Configuration** Cluster and benchmark configuration settings (table 5.1) are set in a distinct `ruby` file (`ycsb_spec.rb`) in the Marlin repository which is cloned on a local machine. Another `ruby` script (`ycsb_bench.rb`) functions as a host and coordinates the whole benchmark process. If the `ycsb_bench.rb` script is executed, a Citus Cluster is being configured using the configurations defined in `ycsb_spec.rb` and a new Citus cluster is spawn up.

2. **Metadata** In the first phase of the process the script initiates the creation of a `config.yml` file by calling the `build.py` script. The config file is saved locally and stores all necessary metadata for the profiling process such as PostgreSQL user passwords and specific endpoints.

3. **Preparation** The database contains 3 users by default: *postgres*, *citus* and *monitor*, where the latter user is solely used for logging. Once a cluster is configured, the host alters permissions for the *monitor* user on every worker node in the Citus cluster. This is done by a separate script since the automatic propagation of `ALTER` statements to the worker nodes failed during the time the infrastructure was build.

4. **Driver VM** One or more driver VMs are created by `ARM` and `.bicep` templates, which are initiated by the host on the local machine. A `cloud-init` script specified in `driver-vm*.bicep` file initiates the `Citus-benchmark` repository on all drivers. In addition, it starts two distinct processes by using `tmux` from which 1) is the *benchmark process* that executes and coordinates the benchmark and 2) is a *server process* used to synchronize the benchmark process and monitoring among multiple machines. It thus is used for the communication with the host and other driver VMs, if any.

5. **Communication** The host attempts to connect with the server on the Driver VM until a connection is established. Once the driver is then ready to benchmark, the host waits until it receives a message with the current state of the benchmark from the driver that it can start the *Monitoring Preperation* process.

6. The **Benchmark Process** first checks whether YCSB and all necessary packages are installed. If this is not the case, it makes sure that YCSB gets installed. Once the installation is completed, it sends a message to the server process on the primary driver VM, who forwards the current benchmark state to the host. Now the monitoring processes can be started.

7. **Monitoring Preparation** The host connects with all distinct worker nodes in the cluster and truncates the `pg_logs` on these nodes. Subsequently it starts a separate process, again with `tmux`, to initiate the sampling with `iostat`. If these operations are succesfull it sends a message containing the current state of the benchmark to the server on the driver VM.

8. **Data Collection** The host awaits until it gets a sign from the driver that the benchmark run is finished. Upon this sign, the host stops all initiated processes on the workers and directly gathers and deletes the generated data once all raw data is transferred to the host.

9. **Finalizing** Once all `pg_logs` and telemetry data is transferred from the Citus workers to the host, the host sends a message to the primary driver VM. This VM can

then broadcast a message to all other VMs (if any) that a run is finished and initiates a new iteration, if scheduled.

10. **Background writer** Steps 6 up to and including 9 are repeated until all iterations are done. A separate thread attempts to transfer YCSB-specific data from the driver VMs to the host every 5 minutes and stores this data in a PostgreSQL RDS instance and S3 for later analysis.

Implementation Details and Considerations

The details of how each stage is implemented is discussed in the following paragraphs. The initial start of the profiling process is initiated by means of `Marlin`. Marlin creates and configures the Citus cluster. It enables to create clusters with specific PostgreSQL and Citus settings, such as setting the `max_connections`, `work_mem`, `enable_binary_protocol` and specific hardware configurations. If the cluster is running, Marlin will initialize processes that creates a configuration (`config.yml`) file with all data of the cluster and prepares the Citus cluster for the monitoring of the benchmarks. At the same time, Marlin initiates a Driver VM that is responsible for running the benchmarks. Once the VM is created and configured, Marlin will initiate a client socket locally. It then awaits until the host can connect to the primary driver VM.

Driver Script

Once the Citus cluster is configured and running, a Driver VM is spin up on Azure by means of the `.bicep` templates which are called in Marlin. In one of the bicep templates, there is a `cloud-init` script automatically executes and installs necessary dependencies such as psql, python, java, wget and tmux. Subsequently, YCSB packages are downloaded and thereafter the YCSB benchmarks starts. In the case of performance monitoring, a second YCSB client executes 0.01% of the total amount of queries, which are logged. The driver script for the YCSB benchmarks is `benchmark.py` and includes methods to automatically execute different YCSB workloads and enables input parameters such as the thread count, the amount of iterations, records, operations. CSV files are automatically generated on the driver VM after every iteration by the `generate-csv.py` script. Both scripts make use of the Python package `Fire`, created by Google, which enables the parsing of arguments in python while executing the python script in terminal. This enables a modular approach, i.e. if something breaks during the benchmark it is easy to manually resume the benchmarks by running a straightforward script.

Sampling

Attempting to avoid introducing significant overhead due to logging in PostgreSQL, we make use of adaptive sampling and sample $\frac{1}{1000}$ (i.e. 0.1%) of the executed queries. The 0.1% sampling frequency is based on the findings from the Dapper paper provided by Sigelman et al. (45), who found that a sampling frequency of one out of thousand provides sufficient information to obtain an accurate profile of a request. PostgreSQL provides a sampling option (`log_transaction_sample_rate`) that enables to specify a sampling rate. However, this option randomly samples *any* transaction on a worker node. Since queries in Citus are distributed across multiple nodes, this means that we cannot control whether we sample two transactions that correspond to each other. As a result, it is likely that we are not able to detect e.g. slow running distributed queries. To overcome this issue we run a parallel workload by means of a second YCSB client which utilizes a different user in PostgreSQL which we call `Monitor`. This user is initialized in Marlin before the configuring of the cluster by means of putting 5.1 in the YCSB benchmark specification (`ycsb_spec.rb`) file.

```
Mediators::FormationRole::Upserter.run(
  formation: f,
  name: "monitor",
  password:  "A*3a" + OpenSSL::Random.random_bytes(32).unpack1("H*"),
)

f.sem.incr(:role_sync)
```

The downside however is that the queries are executed sequentially in a relatively short time span. To run a second YCSB client in parallel that performs inserts, we specify the starting index (`insertstart`) and the amount of records to be inserted (`insertcount`) which are included in the YCSB benchmark tooling. For the `Monitor` user, the settings

```
alter user monitor set log_duration = on;
alter user monitor set log_min_messages = ‘debug1’;
alter user monitor set log_statement to ‘all’;
```

are enabled. A minor addition has been made into the citus source code to enable logging on `DEBUG1` level. These settings produce log messages stored in the `pg_log` folder on every

node in a Citus cluster (`/dat/14/data/pg_log`). The logs give insight in the time duration of work executed on the CPU at both the coordinator and worker nodes when a query is executed. When taking these times into account, we can calculate the total average compute time of a remote and internal query executed on a Citus cluster and herewith estimate the throughput and time spent on different resources (CPU, network and disk).

Metadata

The Host monitors the Citus cluster during benchmarks and orchestrates the benchmark process. After the Citus cluster is up and running, the `build` script generates a `config.yml` file which entails metadata about the cluster as well as cluster specific PostgreSQL data such as the Citus host DNS. In addition, it stores cluster-specific passwords necessary to connect and initiate psql. Non-sensitive metadata is directly pushed to a PostgreSQL database such that the metadata about this benchmark run is kept if the system crashes.

Preparation

The host prepares the cluster with the `prepare` script. This process gives the PostgreSQL user *Monitor* permissions on the coordinator and worker nodes and it creates a temporary file structure. It then waits until the Driver VM is ready to perform a specific benchmark workload that we want to monitor. If the Driver VM is ready, it sends a message to the host which then makes sure that the monitoring processes on the worker nodes are initiated. Once the host initiated these processes and thus is ready, it sends updates the state-array on its second index to 1, and lets the Driver VM know by sending the updated array. This way, the Driver knows it can start with the benchmark run. When the benchmark run is finished, the driver VM sends a message to the host which then makes sure that all gathered data is collected. Once the host collected and removed temporary data files on worker nodes, it notices the Driver VM and further processes and analyzes this gathered data asynchronously.

Communication

The driver and host communicate through a socket which is initiated in a separate process by `tmux` on the driver VM. This server process initiates at least two different threads. The first thread is monitoring for new connections and that want to connect and spawns a separate client thread for every new connection. This new client thread then handles the communication between the server and that particular client.

State Monitor

The host (client) attempts to communicate with the driver once the Driver VM is configured. Both the benchmark driver and host keep track of the current state of the benchmark. The *state monitor* runs as a separate thread on the host and driver(s) to not interfere with the benchmark runs. The current states of the benchmark is stored as an array. Initially the array contains four zeros (i.e. `[0, 0, 0, 0]`) which each represent a different state: (1) ready to benchmark, (2) monitoring prepared, (3) benchmark run finished, and (4) data collection finished. If a state is set to 1, it means that the state is true. Since we keep track of the states, both benchmarks know where to resume once a connection is re-established. Moreover, by means of communication trough a socket, the host can asynchronously process data while new benchmarks are running on the driver. Once all states are set (i.e. `[1, 1, 1, 1]`) the driver resets the states to all zero's and starts the preparation of a new benchmark run. Since the amount of states is dependent on the amount of driver VM's, we each update the current state of the benchmarks by a bitwise `OR` operator between two lists. Since Python lacks support for working with bits directly (i.e. without casting a list to a matrix using the numpy package), we calculate the next state of the benchmark by means of equation 5.1

$$(a|b)_i = a_i + b_i - (a_i \times b_i) \tag{5.1}$$

Heartbeat

The connection between the driver and host needs to remain open while waiting until a benchmark run is finished. However, during large benchmark runs no message is send for a relatively long time. Therefore, we attempted to set `socket.SO_KEEPALIVE` such that after 5 minutes of inactivity, every 3 minutes a keep alive is send to the server. This solution however did not work as expected. To account for this we implemented a heartbeat. This heartbeat keeps the connection alive and ensures that all connected clients contain the most up-to-date states. This heartbeat runs as a separate thread on all the drivers and hosts and sends its current benchmark state to the server.

Telemetry Data

There are different options for monitoring specific statistics, such as `dstat` (43), `htop`, or `iostat`. These profiling tools are extensively discussed in 3.2.1. Similarly as in (17), we make use of standard linux utility `iostat` to monitor CPU usage and disk I/O during

different runs. Iostat exposes all processes we are interested in. Specifically CPU usage (calculated by 100 - `%idle`) and `await` are interesting. Await represents the average waiting time for a request to be processed. This average waiting time includes time in the queue and processing time. However, there is a trade-off between accuracy and overhead (47) caused by iostat. A high sampling frequency, i.e. granularity, could introduce significant overhead while executing benchmarks, so we set the sampling frequency to 1 Hz. Since our benchmarks runs are overall quite long, this frequency should be high enough to capture the behavior of a Citus cluster.

Monitoring

Once the a specific workload run is initiated on the Driver VM, the host acts as a *Tracing Worker* as mentioned in Pi *et al.* (58). When the host acknowledges that the driver(s) are ready to benchmark, it spawns up multiple threads which truncate the `pg_log` files on each worker node to reduce the file size and to throw away unnecessary data. Furthermore, a background process is started on every worker node in a Citus cluster. This process makes snapshots of iostat statistics using nohup and stores the gathered data in `nohup.out`. Once the specific benchmark run has ended, the host will kill the tmux processes on all worker nodes and gathers the generated iostat and `pg_log` data for further analysis.

Benchmarking with multiple drivers

To make sure that the driver itself is not a bottleneck for `INSERTS`, we shard the YCSB workload by *range partitioning* across two different drivers and divide the amount of connections among them in a round-robin fashion. Since there are two drivers, we manually add the reported throughput by YCSB from both drivers to obtain the total throughput for a particular run. We shard the workload by range as depicted in algorithm 2 and 3 and set the `INSERTSTART` and `INSERTCOUNT` parameters from YCSB accordingly. The amount of connections is then calculated by means of algorithm 4

Communication with multiple drivers

The communication between the two drivers and the host differs slightly from the communication between the driver VM and host with a single driver. The second driver VM needs also to be taken into account when the benchmark states are updated. This is solved by using an array of length 6 to include to states of the second driver VM. The host now waits until both VM's are ready to benchmark until it starts the monitoring processes.

Algorithm 2 Pseudocode for calculating the Insertstart, Insertcount for Parallel runs

Require: $drivers \geq 1$
  $shardsize = records \ // \ drivers$
  $insertcount = driverid * shardsize$
  if driver VM is the last driver VM **then**
    $insertcount = shardsize + (records \mod shardsize)$
  else
    $insertcount = shardsize$
  end if

Algorithm 3 Calculating the Insertstart, Insertcount for Parallel runs for Monitor

$insertstart = (1 - sampling\_frequency) * shard\_size$
$insertcount\_monitor = shardsize - insertcount$
$insertstart\_monitor = insertstart + insertcount$

Algorithm 4 Calculating the amount of connections for Parallel runs

Require: $drivers \geq 1$
Require: $connections \geq 1$
  $connections = threads \ // \ drivers$
  if driver VM is the last driver VM **then**
    $connections = connections + (threads \mod connections)$
  end if

Data collector

The data collector is a separate thread on the host and functions as a background writer. It writes data from the benchmark drivers to the host continuously, such that the host can asynchronously monitor the executed benchmarks. This speeds up the benchmarking process as there is no need to wait until all data is transferred from the driver to the host. Since the drivers often maintain a large number of connections to the Citus cluster, connecting with the driver to gather data is sometimes leads to a `connection timeout`. As a consequence, the whole benchmark process is then stalled if we wait for the data on the driver to be transferred to the host. The data collected by the data collector is the raw data that is produced by YCSB along with the `csv` file containing the relevant results of the benchmark. The data that is produced on the Citus cluster itself are collected after every benchmark run is finished.

File Structure

In order to distinguish data from different workers, iterations and drivers, the structure of the gathered file is composed such that each worker and iteration could be identified in the filename. For example, for a postgresql log the structure is `$RESOURCE/pglogs/PGLOG-${WORKER_NUM}-${ITERATION}.log` and for resource metrics the output file path and structure is `$RESOURCE/general/worker-${WORKER_NUM}-${ITERATION}.out`. In addition, the gathered data is deleted on the worker nodes to avoid contention due to a full memory which could reduce the performance of a Citus worker node.

Data storage

After a benchmark run is finished, the host collects all produced PostgreSQL logs by connecting to every worker node and gathering all produced data. It stores data in a temporary file structure and from there parses and subsequently pushes the data into AWS S3 and a PostgreSQL database. The file structure of the S3 blob storage folder is depicted below.

```
resourcegroup/YCSH
    -- resourcegroup/YCSB/raw
    -- resourcegroup/YCSB/results
resourcegroup/pglog
resourcegroup/general
```

The *raw* folder contains all raw YCSB logs output generated on the driver, the *pglog* folder contains PostgreSQL specific log data produced by the user *Monitor*. Lastly, the *general* folder contains the monitoring data produced by `iostat`. The initial parsing of YCSB logs happens on the Driver VM. From every run, each filename is parsed as such that the current iteration, thread count (i.e. client connections), records, workload type, workload name operations and amount of worker nodes along with the measured throughput by YCSB are extracted. In addition, similarly as in the PEEL framework (43) we construct a temporary file system which stores all files gathered from the Driver VM and all nodes in a Citus cluster. Since the size of these files are often large, particularly `pg_log` files, the temporary files are deleted after every configuration once all data is securely stored in AWS S3 and PostgreSQL.

Storing parsed data

Before pushing the data into the PostgreSQL database, results are thus stored in a local temporary file structure. In order to derive meaningful data of the initially unstructured files, we parse the `pg_log`, YCSB output and `iostat` files. Part of extracting YCSB logs already happens on the Driver VM that executes the benchmark itself, where after every benchmark iteration a CSV file is updated with the results of that particular iteration. When the benchmarks crash for some reason, this way partial results are collected in a CSV. Moreover, if the benchmark resumes after a crash the CSV is updated accordingly. After all iterations are finished, the host gathers the results file from the Driver VM, parses it locally and pushes the results into a PostgreSQL database by using the python package `psycopg`. For runs that are monitored, the results are pushed into a table called 'parallel' to keep track of potential performance penalties caused by interference of the monitoring processes. By using a SQL database, results can be easily retrieved and analyzed by performing simple SQL queries.

5.2 Experiments

We perform various experiments with the constructed framework. At first, we investigate whether there is overhead introduced by the framework or the sampling mechanisms. We do this specifically by

- Evaluating whether the monitoring itself introduces any overhead
- Evaluating how much the logging rate influences the total execution time
- Investigating whether the unequal load balancing influences CPU usage

Subsequently, we construct profiles derived from information in the PostgreSQL logs. Since multiple queries use the same connection sequentially, it is hard to find the trace of a single query in the `pg_logs` while only using the `GPID`. If we would match queries on GPID only, the main challenge would become dealing with the short execution times in combination with asynchronous time in a distributed system. To account for this there are roughly three options.

1. Implementing an ID that distinguishes each query ever executed on the cluster and include this in the GPID
2. Aggregate all queries executed per connection path
3. Coarse-grained aggregation: aggregating all parent and child queries

```
2022-06-06 13:00:20.609 UTC [18366][17/74335] : [286616-1]
[app=Citus_internal gpid=120000019449] LOG:  duration: 0.028 ms
2022-06-06 13:00:20.609 UTC [18366][17/74335] : [286617-1]
[app=Citus_internal gpid=120000019449] LOG:  duration: 0.038 ms
2022-06-06 13:00:20.609 UTC [18366][17/74335] : [286618-1]
[app=Citus_internal gpid=120000019449] LOG:  ...
2022-06-06 13:00:20.609 UTC [18366][17/74335] : [286619-1]
[app=Citus_internal gpid=120000019449] LOG:  duration: 0.038
```

Implementing an exclusive GPID for monitoring to identify a distinct distributed query would involve numerous modifications in the Citus source code, which might be tricky. The latter two options are more easy to implement, but do not succeed in constructing an exact profile of a single Citus query. We assume that aggregating all parent and child queries give a rough indication of the overall behavior of a query on a specific node and therefore

choose this option. We can trace the execution time of internal queries (i.e. queries on a child node) in Citus by filtering log lines that include "`app=citus_internal`", used by an internal Citus connection. Fragments of queries that are executed on the parent node can be filtered with "`app=JDBC`". Moreover, it is important to note that by default Citus uses the binary protocol aiming to compress rows. As a consequence, three consecutive messages are spawn in the PostgreSQL logs as opposed to one single log containing the total execution time of that query.

The above chunk of logs represents the output of an 'internal' Citus query. We derive the following execution times from these logs, along with their log-line identifier (28661*). The exact explanation of the logging structure is explained in Appendix 7.2.

Parse: 286616-1 (0.028 ms)
Bind: 286617-1 (0.038 ms)
Execute: 286619-1 (0.038 ms)

In the *parser* stage, a query is parsed by the PostgreSQL parser. This entails checking whether the syntax of the query is valid and if so, a parse tree is build using fixed rules about the SQL syntax. In the *binding* stage a query is evaluated on whether the semantics are correct. This entails validating whether the objects, columns, tables exist and whether the user has the right permissions to access them. Lastly, in the *execution* phase the hash of the shard is calculated on which the relevant tuple for the transaction resides. The query is then forwarded to this node and shard respectively by the *distributed executor*. Once a query arrives at the child node, the PostgreSQL parser and executor finish the computation on the child node.

Profiling Experiments

We construct a profile of queries fired at two different workloads, 100% `READ` and 100% `INSERT` queries. A summary of the main experiments we performed:

- Initial experiments with binary protocol enabled
- Experiments for both `READ` and `INSERT` workload without binary protocol to obtain a profile of the execution of a CRUD transaction
- Experiments with a higher sampling frequency for `INSERT` queries

- Experiments to map the different bottlenecks for smaller clusters with larger nodes, and larger clusters with smaller nodes
- Experiments to determine the maximum amount of connections that a Citus cluster is able to handle
- Running perf while executing `READ` workloads

Calculation of spans

The calculation of the length *monotasks* or *spans* is done by taking the mean of values reported in the `pg_logs`. For every sampled query, we derive the total execution time from the parent (coordinator) node directly from the logs. Furthermore, the total time of the waiting for the remote transaction that is executed is also reported in the logs. Lastly, the compute on the child note itself is reported. Using these values, we aggregate the `pg_log` values into four different execution times:

- **Total avg execution time** T is the total average execution time of a completed transaction from the point a query is received from the Postgres client on the parent node until the moment the query is forwarded back to the client after its execution is completed. This thus entails all operations from PostgreSQL and Citus on the parent node and all operations on the child node including latency between the parent and child nodes.
- **Parent compute** P_c is the total time a query resides on the parent or coordinator node of a distributed transaction.
- **Child compute** C_c is the time that is used for compute by parsing, binding and executing the query on the child node. This excludes waiting for I/O on the child node when the binary protocol is enabled.
- **WaitForChild** W_c is time that is depicted in `pg_log` which reports how long the query on the shard took. This entails the Child compute and the latency from the parent to child node and vice versa.
- **Latency** L is calculated by `WaitForChild - Child compute`. It shows the average amount of time that the query is stalled and no compute on the parent nor child node is performed.

Since the total average execution time T is reported in the PostgreSQL logs, we can calculate the Parent Compute P_c by $P_c = T - W_c$. Similarly, Wait-for-child W_c is reported along with the Child Compute C_c. With these two values we can compute the approximate latency L involved in a transaction by $L = W_c - C_c$.

Calculating the average

For both the child and parent compute filtered rows, we calculate the amount of rows and divide this amount by 3 as the binary protocol spawns three different messages for each executed query. These messages entail *parse*, *bind* and *execute*. For analysis we are (1) aggregating the execution times derived from the `pg_logs` and (2) plotting the utilization of the CPU and disk I/O. For the aggregation from the `pg_logs` we skip the first k transactions in every log to exclude the transaction involve establishing connections and loading data into cache. This is done by means of algorithm 5

Algorithm 5 Skip K first transactions

Require: `pg_log` file
Require: lines to be skipped k
Require: filetype $f \in \{0, 1, 2\}$
  if not f **then**
    $k = 2k$
  else$f < 2$
    $k = 3k$
  end if
  skip_index = 0
  for index, line in `pg_log` **do**
    while $k > 0$ **do**
      try: float(line)
      k -= 1
    end while
    skip_index = index
  end for
  return: pg_log[index:]

By means of this algorithm, we succeed to exclude the first few long-executing queries that typically involve loading data pages in main memory and evaluating query plans. The multiplication factor of k is depending on whether a transaction consists of either 2 and 3

distinct execution times for parent-queries and queries on the shard respectively. Therefore, we multiply the k with the amount of `pg_log` messages corresponding to whether we are filtering on total execution times (parent queries) or execution times for queries on shards (child queries). After we skipped the first k transactions, we aggregate the values by iterating through the PostgreSQL logs of all different Citus nodes and storing all execution times corresponding to either parent or child queries in a python list. To obtain the mean execution time, we sum all values in this list and divide this value by the the amount of values in the list divided by 2 and 3 respectively. Lastly, we parse and aggregate all queries on the shard to get an indication of how long the parent had to wait to finish a computation.

Experiment set-up

If not stated otherwise, we experiment on 32-node clusters that each consist of 16 vCpu's. We specifically investigate the limits of Citus and thus choose this configuration since Citus clusters do not scale well beyond 32 worker nodes. In addition, provisioning clusters larger then 32 worker nodes consisting of 16 vCpus seemed challenging as the cluster becomes more prone to the failure of a single node. The regeneration or failure stalls the whole cluster configuring process. The bottlenecks at 32 worker nodes thus seem interesting to investigate since they might explain why Citus does not scale well. The worker nodes are `Standard_E16ds_v5`, `E_ds_v5` machines on Azure containing the 3rd Generation Intel Xeon Platinum 8370C processors which reach a clock speed of 3.5 GHz. Each node has 128GB of RAM and a 2TB disk. The driver VM consist of 64 vCores and is a `Standard_D64s_v3` machine. This machine contains 256GB RAM and 512GB storage. We use YCSB workload a for 100 Million `INSERT` queries and workload c for 1 Billion `SELECT` queries, if not stated otherwise. Note that all data fits in RAM. We insert 100 Million records and every record in YCSB corresponds to roughly 1 kb. This means that all records together comprise 100GB. 100GB $<$ 128GB RAM per worker node. Assuming that records will be sharded in a round-robin fashion across multiple worker nodes, the amount of GB per worker node that is filled by records is even less. Since all data fits in RAM, disk I/O is avoided for `SELECT` queries. The JDBC loadbalancer is used to loadbalance the connections across different workers in the Citus cluster. First, we investigate the profile of `INSERT` queries, followed by `SELECT` queries.

Cluster configuration

The Citus coordinator and worker nodes are configured with the values depicted in table

5.2.

Table 5.2: Coordinator and Worker nodes cluster configuration

Parameter	Value
work_mem	16MB
max_connections	10000
superuser_reserved_connections	3
citus.max_shared_pool_size	10000
max_prepared_transactions	11500
pg_stat_statements.track	none
citus.stat_statements_track	none

5.3 Results

In this subsection we depict and discuss the results of the experiments done with Citus cluster to expose underlying bottlenecks. First, we briefly test the overhead we introduce by logging and profiling on the Citus clusters. Then we perform various experiments where we test whether the disabling of binary protocol leads to a reduce in performance, what the current (hardware) limitations are and where in which fragment of a transaction the transaction can best be optimized.

Monitoring Overhead

At first we compare the overhead introduced by the monitoring framework to see how much the benchmarking infrastructure influences the throughput. We did 2 separate runs consisting of 4 iterations each on 5-node (`Standard_E32ds_v5`) Citus clusters with 100 million inserts. The results (table 5.3) show sthat the monitoring infrastructure does not influence the overall throughput. In contrary, the throughput in TPS is slightly higher with monitoring. The standard deviation (σ) however is high, meaning that the difference in TPS for monitoring is likely the result of randomness and that in general the benchmark runs are subject to high variance.

Table 5.3: Monitoring overhead for 5-node Citus clusters (`Standard_E32ds_v5`) with 100 million `SELECT` queries and 800 connections based on four iterations

Configuration	TPS (μ)	Std. Dev. (σ)
Non-monitored	311384	47181
Monitored	312909	46931

Sampling Overhead

The overhead introduced by the monitored sampling with increasing sampling rates is tested. The sampling rate we extensively use is 0.1% and based on the findings of Dapper (45). Since we are not using Dapper but our own constructed monitoring framework, we investigate the overhead to see whether this sampling rate is also applicable for our monitoring infrastructure.

However, if we set the sampling frequency high the `pg_logs` will increase in size. Since these logs need to be transported across the network and stored at a local machine and

S3 which costs money, we prefer the sampling rate to be relatively low. If we sample 0.1% of 100 Million queries, we still have data of 100000 samples. Although the sampling rate does not seem to play a significant role in workloads with 10 Million inserts, this could be different for 100 Million inserts since a tenfold amount of records will be logged at every sampling rate.

Unequal load balancing

One of the key advantages of Citus enabling the user to connect with any node in the cluster, is that one can balance their load across multiple nodes to increase throughput. However, when the connections are not equally distributed across nodes in the cluster the influence on throughput and other factors such as memory and CPU usage of a node under a heavy workload is unclear. We make use of the JDBC driver to loadbalance connections across multiple nodes. The JDBC driver selects suitable candidates and randomly load balances connections among these candidates. As a consequence, the distribution of direct connections from the JDBC client to the worker nodes is often unequal. For example, for a run with 2 worker nodes the direct connections from the JDBC client are all forwarded to a single worker node. This means that the resulting worker node only gets internal Citus connections. To figure whether this unequal distribution of direct connections leads to a difference in CPU usage we plotted the results. Figure 5.3 exhibits that there is very little variation between the CPU use of the two worker nodes. The worker node containing solely internal connections exhibits a slightly lower CPU usage during the benchmark run. This difference however is not significant and thus negligible. Since all workers in a Citus cluster ultimately end up with the same amount of connections, we conclude that direct connections from the YCSB client and internal Citus connections do not have a significant impact on difference in CPU intensity.

Distribution of execution time

Since the query execution times of a workload differ highly, we try to map these values and investigate the distribution of the values. This will enable us to identify outliers and understand how the average query execution time is constructed. For example, when the distribution is very skewed and we take the average, this could give a wrong indication. Also, if there are a number of extreme outliers this will also influence the average and we might need to investigate if there is a bottleneck that produces these outliers.

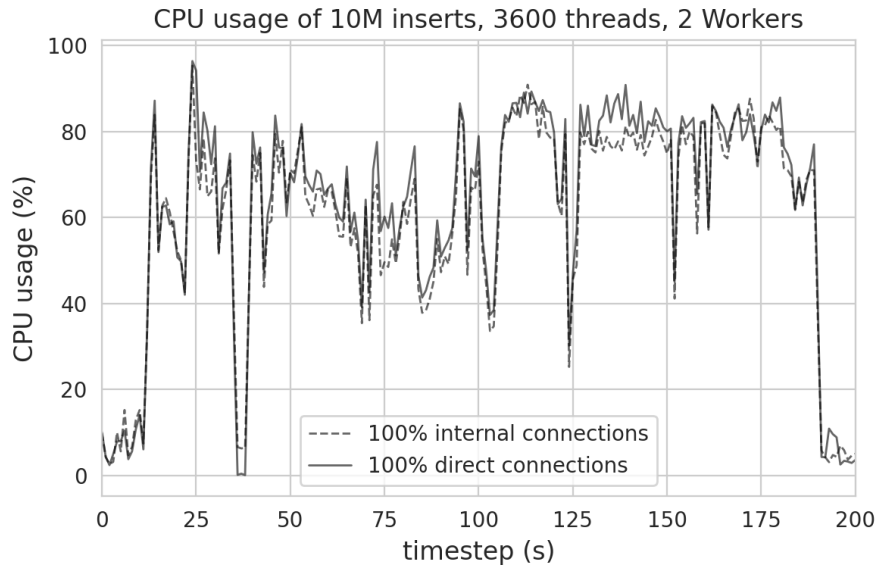

Figure 5.1: JDBC connections with unequal load balancing of direct connections. 2 Worker nodes, 16 vCPUs, 128GB RAM, 2TB disk, 3600 connections.

The left graph in figure 5.2 exhibits that the distribution of the `INSERT` queries exhibit one narrow peak around 4-5 ms. Remarkable is the distribution time of `SELECT` queries that exhibit a large peak followed by a smaller peak. Since all the data for this benchmark should fit in RAM, it is unlikely that data pages are fetched from disk. A modern CPU often has three different levels of cache, where L1, L2 and L3 respectively increase in storage capacity but decrease in accessing speed. The difference between the first and second peak of the read workload is around 1 ms, which is too high for accessing a different caching level of RAM. The longer running queries could include some form of extra computation or communication however. Both histograms are truncated at 14 and 2 ms respectively, but there are some influential outliers that reach 300 ms. Explanation for these outliers are broad. A possible explanation is that Azure hypervisor, the program that deploys and monitors the VM provided by Microsoft, influences these execution times periodically by some of its processes. Due to these outliers however taking the mean of the distribution will give skewed results. Therefore, we can do two things of which (1) is removing the outliers and (2) taking the 95th or 99th percentile (18) as opposed to the arithmetic mean. We choose to use the 95th percentile since outliers possibly expose useful information about requests that perform poorly.

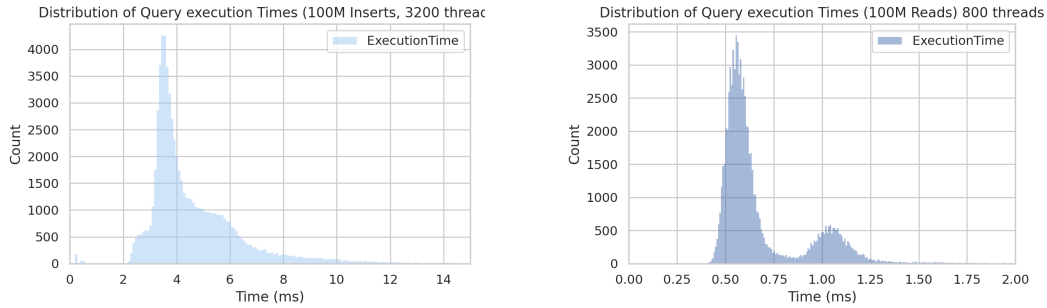

Figure 5.2: Left: INSERTS, Right: READS. Distribution of average query execution times from the Postgres client until its forwarded back to the YCSB client. Execution times are derived for a workload of 100 million `INSERT` and `SELECT` queries with 3200 and 800 threads respectively.

Profile of `INSERT` queries

To identify straggler causes in `INSERT` queries, we construct a bar chart containing all execution times of the fragments of a distributed query. The latency part in figure 5.3 is calculated by $L = W_c - C_c$, meaning that the latency is calculated by the time a distributed executor on the parent node waits until it receives a response back from the child node minus the compute time on the child node. It thus shows the average amount of time that the query is stalled and no compute on the parent nor child node is performed.

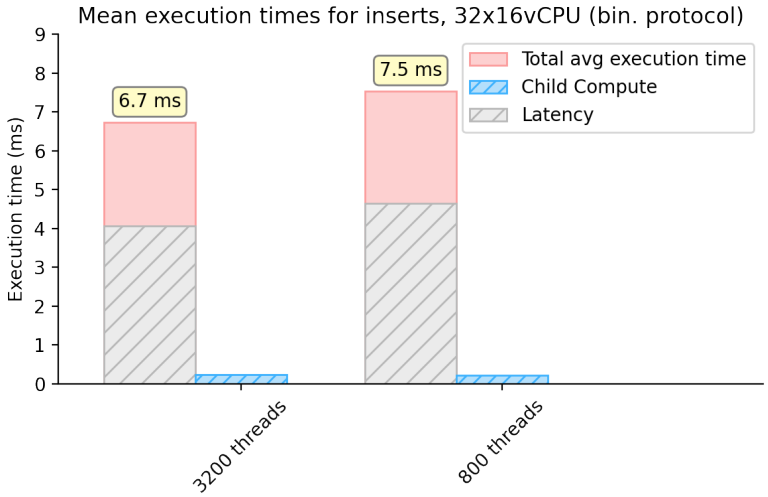

Figure 5.3: Profile of time of `INSERT` query spent in different monotasks, binary protocol enabled. *fsync()* not captured.

The results of the `INSERT` workload exhibits that most time of an insert is spent waiting

to receive a `SUCCESS` from on the child node. This part of the query execution seems to be the straggler cause for inserts. By decomposing the profile of the insert, we suspect that *fsync()* system call is not taken into account on the compute on the child node since the duration of the execution times on the child node are arguably too short. *fsync()* is called to write a chunk of WAL to disk during an insert or update of a query. This method makes sure that all inserts are physically written to disk. *fsync()* thus is a very expensive function since the database needs to wait until the WAL is flushed to disk. Disabling *fsync()* to increase performance is possible, but makes it impossible to recover to a consistent state when a crash occurs. After investigating the PostgreSQL source code, we found that the `finish_xact_command` is not always called from a `exec_execute_message`, but only if it is an explicit `COMMIT` query. Therefore, the *fsync()* duration is not captured when the binary protocol is enabled. It is surprising that PostgreSQL does not log correct execution times in its default mode of operation for queries involving a WAL write.

```
if (is_xact_command || (MyXactFlags & XACT_FLAGS_NEEDIMMEDIATECOMMIT))
        {
            /*
             * If this was a transaction control statement, commit it.  We
             * will start a new xact command for the next command (if any).
             * Likewise if the statement required immediate commit.  Without
             * this provision, we wouldn't force commit until Sync is
             * received, which creates a hazard if the client tries to
             * pipeline immediate-commit statements.
             */
            finish_xact_command();
```

Binary protocol

For inserts we suspect that most time is actually spend at *fsync()* to flush WAL files to disk instead of the latency itself being a straggler cause. The sync duration is not captured by the execute message when the binary protocol is enabled. This gives a distorted view of where most time is spend within an `INSERT` transaction. When the binary protocol is enabled, PostgreSQL sends `bytea` type fields directly as bytes as opposed to converting them. This is beneficial if the workloads consist of data that require less storage if converted to byes. To capture the data of the full commit, we disable the binary protocol by setting `citus.enable_binary_protocol` to false. This leads to PostgreSQL executing other code that does not make use of the binary protocol, which does capture the execution

time of the commit. The downside however is that slightly more overhead is expected when disabling the binary protocol.

Binary protocol vs no Binary Protocol
To see whether the disabling of the binary protocol influences the throughput, we compared 2 runs with and without the binary protocol enabled. The results show that on average for inserts with clusters of 2x16vCPu, standard `E_16ds_v5` disabling the binary protocol does not lead to a significant performance decrease. The average for the binary protocol enabled for YCSB workload c with 10 Million `SELECT` queries on 10 Million records is 93.519. For binary protocol disabled this number seems a bit higher, approximately 100000. However, in this experiment we only executed 2 iterations per protocol on different clusters. A possible explanation that disabling the binary protocol does not lead to a performance degradation is that the the values generated by YCSB benchmarks consist of strings of random values. If these random strings are converted to bytes, the strings are not necessarily more compact. This leads to a less efficient execution. As an example, a `BIGINT` with a value could be 1 byte in text but 8 bytes in binary. If this is the case for most data used for the benchmark, this will result in an amplification of network traffic and ultimately slow down performance.

Profile without binary protocol
Without the binary protocol we get a clear picture of where most time is spend within an insert. The percentiles of execution times are reported in table (5.4). Note that the shard in which queries are inserted is calculated by calculating the hash of the primary key. If clusters are small, for example consisting of two nodes, then the probability is *50%* that the insert will take place on the same node. The record to be inserted is then first appended to the WAL file, which is subsequently written to disk. As a result, the child compute is almost equally as long or even longer as the compute on the parent node.

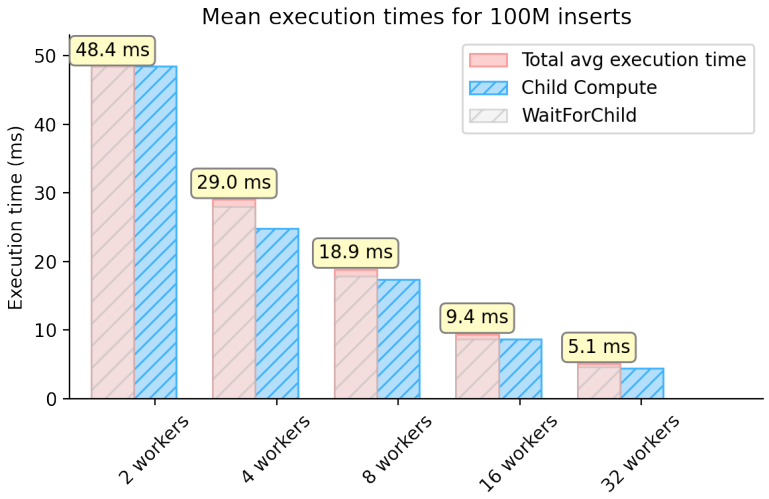

Figure 5.4: Profile of `INSERT` query without binary protocol, *fsync()* captured

Table 5.4: Percentiles query execution times for 100 million `INSERT` queries without driver-to-cluster latency, cluster-to-driver latency and YCSB overhead, 3600 threads, 16 vCPUs

Workers	50th pctl. (ms)	95th pctl. (ms)	99th pctl. (ms)	YCSB latency
2	33.36	124.34	192.48	56.63
4	14.24	81.83	128.05	32.1
8	10.19	60.88	118.08	20.9
16	5.86	18.57	55.31	13.3
32	4.23	10.61	22.77	11.1

YCSB or driver bottleneck

We are merely interested in the behavior of a transaction executed against a Citus cluster from the point a distributed query arrives at a parent (coordinator) node until the query is forwarded back to the YCSB client. This is, since Citus itself cannot influence the application used to perform a workload which in this case YCSB. Ideally, the latency introduced by YCSB should therefore not be taken into account. However, the YCSB latency influences the total throughput reported by YCSB to some extent. The difference in the latency reported by YCSB and the mean execution time of a query is for example $11.1 - 5.5 = 5.6$ ms at a 32-node cluster, so we expect that the YCSB itself introduces a significant amount overhead ($> 50\%$ in the example calculation). This could indicate

that either the latency from the driver to the cluster and vice versa is high, or YCSB itself incorporates significant overhead by spawning up millions of queries in a short amount of time. Most likely however it is a combination of both.

Higher sampling frequency

To explain the difference in latency reported by YCSB and the average total execution time, we increase the sampling frequency to 10% of the total amount of queries to extend the run time of the monitor inserts. With a 0.1% sampling frequency, the sampling run time is around 6 seconds for inserts, which is short. Moreover, if we compare the latency reported by YCSB for the user *Citus* with the user *Monitor*, we observe a value of 11.1 ms against 5.9 ms respectively. This could indicate that the profiles of the `INSERT` queries are not accurate. To account for this we increase the sampling frequency to 10% of 100 Million. Then we investigate the statistics of $0.1 * 100000000 = 10$ Million monitored distributed transactions. We get the results depicted in table 5.5. The average total execution time is **8.04** ms, the average execution time on a child node is **7.22** ms and the average execution time that a parent node waits for the results on the child node is **7.48** ms. The latency reported by YCSB is **13.78** ms and thus remains high compared to the average total execution time of **8.04**, as there is a difference of > 5 ms.

Table 5.5: 10 percent sampling frequency for a cluster with 32 worker nodes, 16 vCPUs, 128GB RAM and 2TB disk and 100 Million inserts and 3600 connections

Workers	50th pctl. (ms)	95th pctl. (ms)	99th pctl. (ms)	YCSB latency (ms)
32	5.25	20.37	59.22	13.78

Multiple Drivers

To exclude that the bottleneck is the driver itself, we performed benchmarks using two separate drivers. The results with two drivers show that the transactions per second (TPS) for inserts with 32 workers and 3200 threads increase from 250000 to 275000. This is roughly an increase of 10%. Since the increase seems non-negligible, we benchmark with two drivers from this point on.

Upper bound of connections

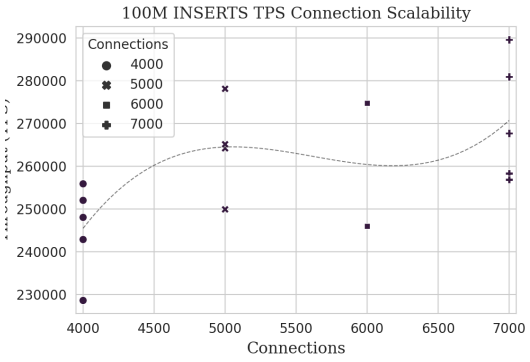

Figure 5.5: Y-axis represents throughput. Scalability of Citus for 100 Million INSERTS (YCSB workload a), varying connections, 16 worker nodes with 32vCPUs (`V5_E16_ds`), 256GB RAM, 2TB disk. Data is fitted with Numpy `Polyfit` function.

One of the potential limitations we want to investigate is whether a large amount of connections becomes a bottleneck at a large Citus cluster. To do so, we aim to find the maximum throughput for an `INSERT` workload. We increase the amount of connections for two drivers and a Citus cluster where each worker node is a `Standard_E32ds_v5` (256 GB RAM) machine (figure 5.5). Table 5.6 shows that the TPS increases with an increasing amount of connections, though with diminishing returns and a relatively high variance.

Table 5.6: Benchmark results of Citus clusters with `Standard_E32ds_v5` (256 GB RAM) hardware. Coordinator and worker nodes contain 32 vCPUs, 256GB Ram, 2TB disk. Benchmark workload is 100 million inserts and 16 worker nodes

Connections (w)	**TPS** (μ)	**Std. Dev.** (σ)	**Conf. Interval (p=0.95)**
4000	245498	1554.87	$236206.1866 < X < 254789.8653$
5000	264526	2374.32	$255767.5434 < X < 273284.46404$
6000	260429	7473.57	$232189.3626 < X < 288669.94813$
7000	270723	3216.34	$258209.6226 < X < 283237.4695$

The results show that the throughput slowly increases with an increasing amount of connections. However, at 8000 connections YCSB started to report out of memory (OOM) errors, thus we did not take these results into account. This indicates that the Citus cluster is memory limited.

Connection limit per CPU

The memory bound is an interesting finding and to explore its limitations we perform experiments to map how many connections a node can handle per GB RAM. When benchmarking a Citus cluster of 16 `Standard_E16ds_v5` machines (128GB RAM) we often get OOM errors when the amount of connections exceeds 3600, and for when we surpass 4000 connections. This could be the result of the random loadbalancing of the JDBC driver. The JDBC loadbalancing algorithm randomly chooses a suitable candidate node which often leads to an unequal distribution of connections from the driver to a worker node. Since the connections from the driver are slighly more CPU intensive as process more request, this will more quickly lead to CPU saturation. If we upscale the nodes such that they contain machines consisting of 32 vCPUs and 256GB RAM instead of 16 vCPU and 128GB RAM, the connection limit increases until $\tilde{8}000$ connections before OOM errors occur. The amount of connections per vCPU in `Standard_E32ds_v5` machines can thus be approached as $\frac{8000}{32} = 250$ until out of memory errors occur. This means that for example a cluster with workers that contain 8 vCPUs, the maximum amount of connections is approximately $8 \times 250 = 2000$. More specifically, when calculating in terms of GBs the connections are limited to $\frac{8000}{256} = 31$ **connections per GB**. An increase in amount of connections after this point does not lead to an increase in average response time reported by YCSB due to the OOM errors.

Why do more connections lead to better performance for inserts?

The WAL is committed directly through *fsync()* if synchronous_commit = on . However, more connections lead to a performance increase since more Write-Ahead logs are committed simultaneously in a group commit.

Comparing a 32x16vCpu with a 16x32vCpu cluster

An open question is whether it is better to have many nodes but of a small size, or smaller cluster containing larger nodes (more vCpus). This highly depends on the workload as both architectures have different limitations. To map the limitations of these two different architectures handling a similar workload, we experiment on a 32x16 and 16x32 Citus cluster.

Table 5.7 shows the results of experimenting with different cluster sizes and workers from `Standard_E16ds_v5` machines, containing 16 vCpus. If we compare the results of average TPS for 32 workers with 16 vCpus and 16 workers with 32 vCpus, the former outperforms

Table 5.7: Benchmark results of Citus clusters with `Standard_E16ds_v5` hardware. Coordinator and worker nodes contain 16 vCPUs, 128GB Ram, 2TB disk. Benchmark workload is 100 million inserts and 3600 connections

Workers (w)	**TPS** (μ)	**Std. Dev.** (σ)	**Conf. Interval (p=0.95)**
2	62479	2374.32	$60398.15 < X < 64560.44$
4	111322	7473.57	$104771.32 < X < 117872.83$
8	160498	3216.34	$156858.22 < X < 164137.35$
16	246396	6253.53	$240268.12 < X < 252524.80$
32	292447	19700.41	$273140.76 < X < 311752.85$

the latter under an `INSERT` only workload. A cluster of 32x16 with a thread count of 3600 has an average throughput of 290k TPS, while a cluster of 16x32 worker nodes with a thread count of 7000 has a throughput of 270k TPS (table 5.6). The difference in TPS is likely due to the larger worker nodes (`Standard_E32ds_v5`) being I/O bound. In other words, transactions are waiting for a longer amount of time on the WAL to be flushed to disk. Since disk I/O is possibly saturated, there will form a queue of transactions waiting for their record to be flushed to disk. A 32-node cluster has twice as much disks, so this process is quicker for transactions on a 32 node cluster. However, the TPS on a 32 node cluster is not that much higher. This could indicate that the disk I/O is not fully utilized and the optimal relation between connections and cluster size is at a smaller Citus cluster. As also visible in table 5.7, a 16x16vCpu cluster reaches a throughput of 240k, which is close to the performance of a 16x32vCpu cluster, with machines that consist of half of the amount of Cpus. The optimal relation with cluster size and connections thus probably lies somewhere in the middle of 16 and 32 worker nodes. For example, when testing the limiting amount of connections before OOM errors occur, we tried a run with *20* worker nodes and 32 vCores, 258 GB RAM, 2TB disk and 7600 connections. We performed a workload of 100 Million INSERT queries executed by two driver VMs and obtained a throughput of **357217** TPS as reported by YCSB. This throughput outperforms the 32x16vCpu cluster, but at higher costs.

Errors

For inserts, if the cluster exceeds 3600 connections from the driver Out of Memory (OOM) occur in the YCSB logs. This is, due to the high amount of connection either the driver or worker nodes lack free memory. Remarkably, the CPU is not busy but this is expected due

to the wait for flush on disk. In addition, it may occur that `ExclusiveLock` errors appear in the `pg_logs` when processes hold locks for more then 1000ms.

Utilization

The CPU usage for inserts with 800 connections and 3200 connections are exhibited in figure 5.6. The figures show that the `INSERT` workload is not CPU bound. This workload only consumes little CPU, even with a high amount of connections and an intensive workload. This can be explained due to the transaction being stalled when it is waiting for I/O from disk. If a transaction is stalled most of the time, no CPU is used and thus CPU usage is low.

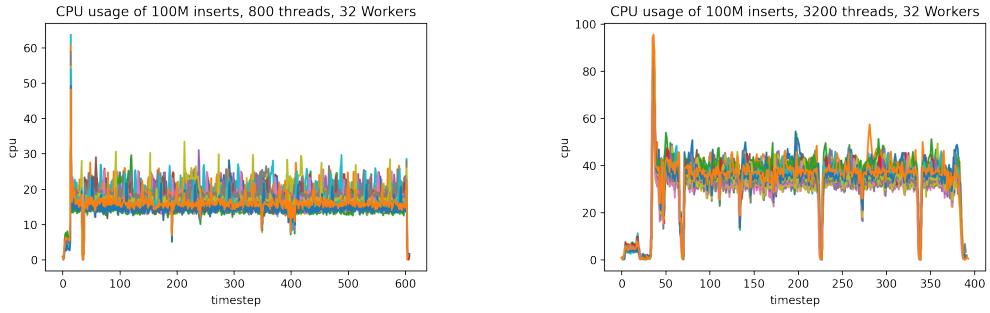

Figure 5.6: CPU utilization for Citus cluster of 32 worker nodes, shard count of 64, 100M inserts, 800 and 3200 threads respectively and a single Driver VM of 64 vCores. Only inserts. Threads in this case means connections to the entire Citus cluster.

Waiting for I/O

In these plots the `await` value at dm-0 from iostat for I/O is plotted. `await` includes the time in queue and execution time of a thread. This execution time should roughly correspond to the compute on the child node, since the large part of this estimated compute involves waiting for I/O.

The graphs in figure 5.7 shows that there is a fat peak in waiting for I/O at around 75 seconds into the benchmark run. The most likely cause for this is checkpoints that force dirty heap pages containing newly inserted data to flush to disk.

Saturation

For the `INSERT` workload, the CPU nor disk seem saturated at 32 worker nodes with 16 vCpus. However, for smaller clusters the disk is the bottleneck. For larger clusters, the workload increases with the amount of connections but the amount of connections are

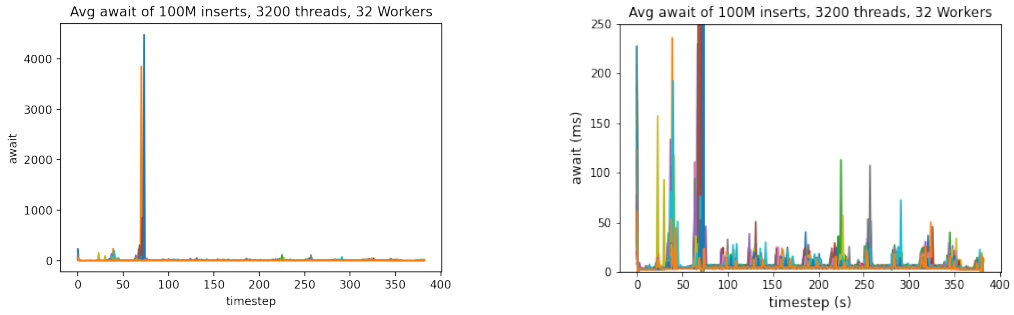

Figure 5.7: Await reported by iostat during an Insert workload run for Citus cluster of 32 worker nodes, shard count of 64, 100M reads on 100M records, 3200 threads and a single Driver VM of 64 vCores. 1b is a zoomed version of 1a, truncated at 250 ms waiting time.

memory bound.

Profile of a `SELECT` query

If we compare the performance of Citus on a read-only workload to the performance of Aurora (section 4.3), it is evident that there is a lot of room for improvement in Citus with regards to read-only queries. Therefore, it is interesting to profile the SELECT workload to identify straggler causes and explore possible optimizations. To investigate the bottlenecks of the `SELECT` workload, we performed experiments with **800** connections and varying cluster sizes with `Standard_E16ds_v5` worker nodes and binary protocol disabled. The graph in figure 5.8 exhibits an evident decrease of query execution size with an increase in cluster size. However, after 16 workers this execution time stagnates and even increases a little bit.

If we decompose the profile of the `SELECT` queries, we observe that the a query spends more then 50% of its execution time on the parent node. This means that the latency to and from the child node, along with the compute on the child node, is less then the compute on the parent node. There is thus room for optimization for `SELECT` queries in the computation(s) on the parent node. On the other hand, the compute times on child nodes are very short which indicates little room for improvement.

Shortcomings

Table 5.8 shows that the YCSB latency is smaller then the 50 percentile execution times for clusters with 2, 4 and 8 workers. This is remarkable, since the latency reported by YCSB includes the YCSB overhead and the latency from the cluster to the YCSB driver and vice versa. An explanation is thus that the distribution of values is very skewed or corrupted.

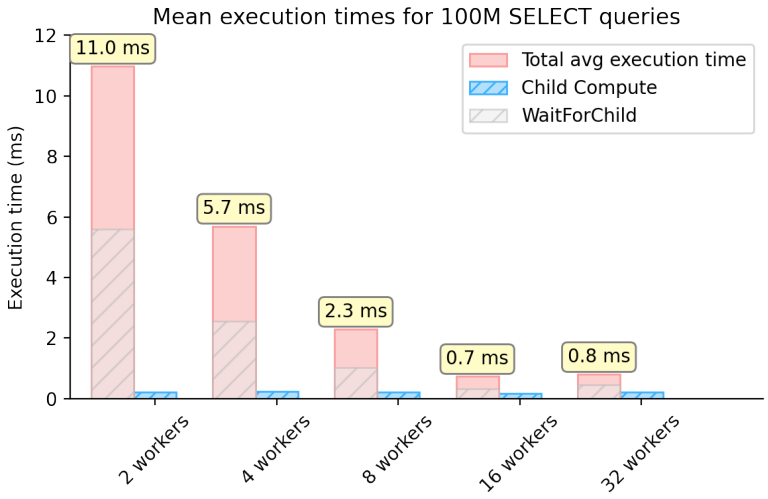

Figure 5.8: Profile of query execution times for 100 million `SELECT` queries without driver-to-cluster latency, cluster-to-driver latency and YCSB overhead, 800 threads, 16 vCPUs

Table 5.8: Percentiles query execution times for 100 million `SELECT` queries without driver-to-cluster latency, cluster-to-driver latency and YCSB overhead, 800 threads, 16 vCPUs

Workers	50th pctl. (ms)	95th pctl. (ms)	99th pctl. (ms)	YCSB latency Citus
2	11.59	17.06	39.67	6.56
4	4.88	15.08	21.69	3.77
8	1.01	7.75	12.95	2.04
16	0.77	0.99	1.35	1.25
32	0.81	1.90	2.97	1.06

However, if we compare the latency reported by YCSB this is significantly lower then the mean total execution time of a query of 11 ms. Since the YCSB latency is the latency reported by the Citus user, this could mean that the queries from the Monitor user are remarkably slower, since the plotted execution time are derived from the PostgreSQL logs from queries executed by *monitor*.

Another issue arises when we perform a simple calculation with the depicted execution times. If we have 11ms times 100 Million, we get

$$\frac{11 * 100000000}{1000} = 11 * 100000 = 1100000 \tag{5.2}$$

The calculated amount of 1.1 million TPS by no means reflect the reported amount of a little over 100k TPS reported by YCSB. Part of the 'lost' throughput can be explained by the YCSB driver-to-cluster and cluster-to-driver latency, but the difference of 1.1 Million - 0.1 million is 1 million, which is too large.

Profiling with perf

To investigate the limit of 800 connections for `SELECT` queries and possible optimizations for `SELECT` queries we attempt to profile what happens during its execution. To do so we profile a PID that refers to a `SELECT` query using `perf`. This however is challenging since 1) a query could be executed in less then 1 ms and 2) Azure VMs disabled some perf functionalities such as recording the amount of CPU cycles. To account for this we took eight different samples of a PID that referred to a `SELECT` query, to obtain an overall overview of what happens on a Citus Worker node during a heavy workload consisting of solely `SELECT` queries. By using `perf`, we found that `finish_task_switch` system call causes most overhead, around 4.17% (table 5.9). This kernel function is called at the end of a `context_switch` function by another thread. The high usage of `finish_task_switch` is likely due to the large amount of processes (i.e. connections). Therefore, more connections could incur a lower performance due to the high amount of context switches.

The `postgres`-specific functions that appear most often are `SearchCatCache1`, `AllocSetAlloc`, `hash_search_with_hash_value` and `hash_seq_search` respectively. The first Citus specific function that appears in all samples occurs always after some `postgres`-specific functions. The Citus function which introduces most overhead (0.64-0.90%) is `ConnectionStateMachine`, often followed by
`AfterXactConnectionHandling`. Note that the percentages are not very representative on Azure, but they still could give an indication. The `ConnectionStateMachine` opens a connection and goes into transaction state machine when finalized. `AfterXactConnectionHandling` is a function initiated by the global transaction callback of Citus. It handles both `COMMIT` and `ABORT` paths and manages the connections after a transaction is completed. The Citus-specific functions seem to be related to the handling of the connections rather then the execution of the query itself. Moreover, it is noteworthy that all connections use SSL which

can incur computational overhead.

Table 5.9: Most CPU-consuming methods during a `SELECT` query across eight samples

Function Name	Shared Object	CPU overhead (μ)
`finish_task_switch`	[kernel.kallsyms]	4.2 %
`__memcpy_ssse3_back`	libc-2.17.so	2.8 %
`_raw_spin_unlock_irqrestore`	[kernel.kallsyms]	2.6 %
`__do_softirq`	[kernel.kallsyms]	2.1 %

Errors

Since for every cluster size the amount of connections beyond 800 leads to a performance decrease, we keep the amount of connections at 800. As a result, no OOM errors occur. The only errors that occurred during YCSB workload c was with two drivers, but this was likely due to a software fault. Therefore, all benchmarks for the `SELECT` workload are performed with a single driver VM.

Utilization

To try and find an explanation for the sudden decrease in performance for a read-only workload if the amount of connections increases we plotted the CPU usage of two separate experiments with either 800 and 3200 connections.

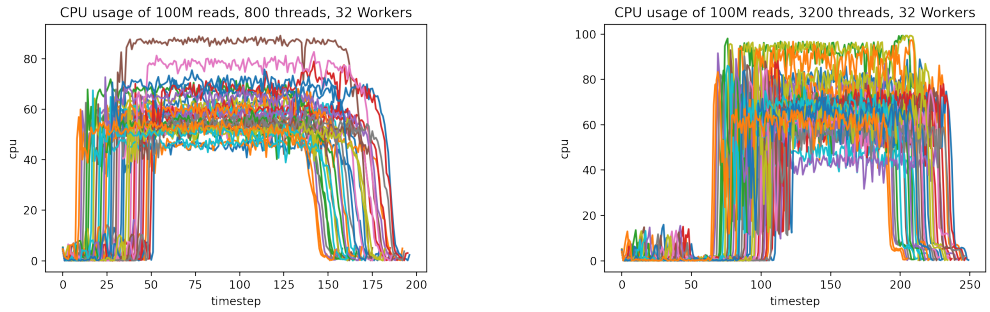

Figure 5.9: CPU utilization for Citus cluster of 32 worker nodes, shard count of 64, 100M reads on 100M records, 800 and 3200 threads respectively and a single Driver VM of 64 vCores. YCSB workload c.

As evident in figure 5.9, reads are consuming a lot of CPU. The left figure 5.9 shows the CPU usage of a cluster with 32x16vCpus worker nodes and 800 connections. The CPU

usage is constantly above 50% and reaches 80%. If we compare this with the figure on the right where 3200 connections are used, we see that the second figure consumes a lot more CPU. This is probably due to the increase in connections.

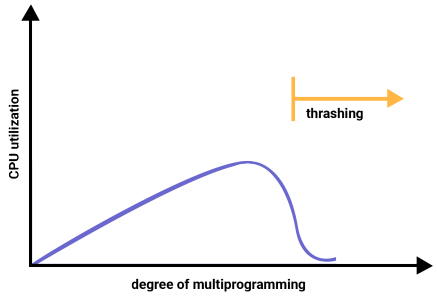

Figure 5.10: Trashing explained in a graph. Image derived from `https://www.studytonight.com/operating-system/thrashing-in-operating-system`

Another remarkable fact is that the 3200 connections require almost 75 seconds of warm-up time in order to even establish the connections, as opposed to a very short warm up time for 800 connections. The large warm-up time could be subject to trashing (figure 5.10). *Trashing* can occur when throughput or CPU usage suddenly drops after a critical load. This is due to higher loads leading to an increase in page faults since the average partition size decreases when the memory size is fixed. A *page fault* occurs when a required page does not reside in physical memory but is mapped into a virtual address space. On top of that, the CPU is much faster compared to the paging server. As a result, the waiting jobs from the CPU queue are moved to the paging server (68). In other words, trashing occurs for example when the OS is busy with swapping pages in and out of memory which in its turn lead to less actual CPU usage. Another, maybe more plausible reason for the warm-up time is that YCSB waits until all connections are established. This takes longer for 3200 connections than for 800 connections.

Saturation

The system behaves as CPU bound for a large amount of reads although CPU metrics do not actually reach 100%. With a large amount of connections, at least larger then 800 connections, the CPU usage of a worker node acts as if it becomes saturated and slows down the whole process. Therefore, one needs to look into how to use less CPU for reads. More connections require CPU in terms of connection overhead and context switches. This is also visible if we look at the result of analysis with `perf`, where `finish_task_switch` introduces

most CPU overhead, even with 800 connections. Furthermore, postgres processes that contribute to the high CPU usage are searching in cache (`SearchCatCache`), allocating memory and hash value search (`hash_search_with_has_value`). Citus itself spends most time to connection management (`ConnectionStateMachine` and `AfterXactConnectionHandling`). If some of these processes can be sped up, CPU utilization will decrease and eventually TPS will increase.

5.4 Bottlenecks

In the results section (5.3) we first exposed that the binary protocol does not capture the duration of *fsync()*, which is surprising from PostgreSQL. Therefore, all experiments are performed with the binary protocol disabled. Also it is worthy to note that the JDBC loadbalancer does not distribute all connections evenly across the Citus cluster. In addition, the main bottlenecks found differ for `SELECT` and `INSERT` workloads and therefore we summarize the for both workloads findings separately.

INSERT workload

A Citus cluster does not scale well after 32 worker nodes as identified in chapter 4. For a write-intensive workload, the cluster will become memory bound. The most important findings are summarized below.

- Citus clusters containing a large amount of nodes are memory limited. For large nodes with 256 GB RAM, the amount of connections are limited to $\frac{8000}{256} = 31$ **connections per GB**. This entails around 8000 connections for worker nodes that contain 256 GB of RAM. An increase in amount of connections after this point does not lead to an increase in average response time reported by YCSB due to the OOM errors.
- For a lower amount of worker nodes the bottleneck becomes disk I/O since WAL writes are waiting to be flushed to disk if the WAL size increases.
- The average CPU usage under an insert workload is relatively low due to a large part of the transaction is spent by waiting for I/O.

SELECT workload

Read-intensive workloads are not dependent on disk I/O if all data fits in RAM. Since we experimented with clusters large enough to fit their entire workload in RAM, disk I/O to fetch pages from disk in memory is largely avoided. The most important findings are summarized below.

- Read-intensive workloads are performing most optimal when no more then 800 connections are used as shown in chapter 4. This is likely due to the system being bound by the amount of context-switches invoked by connection handling. The overhead introduced by context-switching if more connections are used negatively outweighs the benefits of executing more concurrent read-only queries which all make use of a separate connection.

- Most time of a `SELECT` query is spend on the parent node that coordinates a distributed query. The majority of the time on the parent node is likely spend by context switches.
- For a read-intensive workloads the system behaves as CPU bound. Even with a low amount of connections (800) the CPU utilization is high. For example, if we establish 25 connections from the driver to a worker, the CPU utilization is most of the time above 60%. This could indicate that the efficiency of a `SELECT` query is low and that there are some extensive operations involved in coordinating a 'simple' read query. Profiling with perf however only indicated that few context-switching related functions that take up most cpu during execution.
- Optimizations for select queries lie in the avoidance of network hopping to a second worker node or in the compute operations for coordinating the query on the parent node.

Utilization of resources

A surprising result is that the system behaves as CPU bound, but CPU metrics do not reach 100%. This phenomenon is also visible in the work of e.g. Dipietro et al. (2) who researched the behavior of Cassandra. This behavior could be related to virtualization, but can also be the result of other causes, such as contention or network effects. A node that is fully utilized would take longer to send new internal queries to other nodes, which would reduce overall utilization creating an equilibrium. Another cause could be the interrupt handles (IRQ) that could lead to a bottleneck due to the numerous context switches. As a consequence of the continuous IRQs the CPU is not fully utilized. However, the exact reasons are yet unknown and subject to future research.

6

Discussion

Experiment with larger clusters

In this thesis we mainly investigated Citus clusters of 32 worker nodes each containing 16 vCpus. While this cluster is larger than the largest cluster in production (32x8 vCpu), it is smaller then the largest cluster on premises[1]. A 32x16 cluster is thus not the maximum size of a running cluster, but scaling beyond this size for experiments remained challenging. Out of around ten attempts to spin up a 32x32 cluster, zero succeeded and is therefore let out of scope for this thesis. However, to investigate the true limits of Citus a larger cluster could be examined along with a larger workload, that runs for a longer time. Instead of a maximum amount of records in the database, putting a time limit might be more beneficial to simulate a longer run. This will ultimately lead to pages being fetched from disk, which on its turn will lead to a disk I/O bottleneck for larger workloads and thus different behavior in general.

Large-scale Profiling and Sampling

Lots of valuable information gets lost if one takes the mean or percentile of execution times reported in the PostgreSQL logs. To avoid aggregating this information, one should be able to identify individual transactions on a large scale *after* these transactions are committed. The current version of Citus does not enable to analyze the detailed behavior of a transaction after its execution, unless data derived from build-in UDFs such as `EXPLAIN ANALYZE` is manually stored by its user. This is due to a GPID only storing data about the current transaction and nodes it passes with the connections involved. It is inevitable, especially during a large workload, that there exists transactions that follow the exact same path. These transactions cannot easily be distinguished after execution. A solution to account

[1] https://www.citusdata.com/customers/heap

for this is to analyze the data in the PostgreSQL logs and attempt to manually connect the GPIDs to each other within a certain timespan. However, since a Citus cluster usually is a distributed system and execution times of CRUD queries are arguably short, this would be unreliable due to asynchronous clocks. To account for this, one could investigate the possibility to implement a persistent unique identifier in the GPID that accompanies each transaction. This however means that either the existing GPID needs to be expanded in size, which could introduce overhead. This increase potentially outweighs the costs since profiling transactions on a large scale could expose valuable information, but this should be investigated first.

Another option in this extent is to implement a LPID (Logging Process ID), which functions as a GPID but is specifically intended for logging. The LPID should function as a GPID, but then carries a unique identifier so that after execution query fragments can be coupled. In other words, if the query is specifically used for logging, it takes a LPID instead of a GPID. By including this unique identifier one is able to combine different fragments of a single transaction and construct a complete profile of a single query. In this manner a more accurate impression of what happens during a workload will be obtained, making it easier to identify new areas for improvement by using e.g. blocked-time analysis (61).

Instead of the sampling architecture we used to profile Citus by means of a different PostgreSQL user *'monitor'*, randomly sampling and profiling a fraction of all transactions executed in a Citus cluster might be a better option. There exist a sampling setting enabled by PostgreSQL (`log_statement_sample_rate`), but this does not work yet for distributed transactions in Citus. A version of `log_statement_sample_rate` for distributed transactions would solve this. However, since queries are distributed accross multiple nodes this is challenging.

Simulating Bottlenecks

We derived potential bottlenecks for large Citus clusters by observing the behavior of Citus under simple read- and write workloads. To take this one step further, one should attempt to model this behavior or simulating potential bottlenecks by making use of tooling such as `dm-delay`[1] to simulate a delay in writes, or `netem` (69) to simulate package drops on the network. By simulating the potential bottlenecks, we can state with more certainty that

[1] https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/delay.html

the indicated bottlenecks truly are bottlenecks and not just noise.

Load Balancing

The JDBC loadbalancer lists suitable worker nodes and then randomly chooses one to connect to. As a consequence for Citus, some worker nodes do not end up as coordinators for distributed transactions. This could eventually lead to a performance decrease on the nodes that function as coordinators, since the coordination of a transaction involves slightly more overhead. Other disadvantages from the JDBC loadbalancer is that it involves a decent amount of manual work by the user to set it up. In addition, JDBC is a single point of failure since it does not guarantee high availability and does not provide any auto-scalability. To account for this, Citus could consider to make its own auto-scalable loadbalancer with the option to loadbalance in a round-robin fashion across the cluster. Another solution is to let the user choose to which set of nodes to loadbalance to, mainly to provide some flexibility. At last Citus could adapt or mimic the JDBC loadbalancer and randomly distribute the connections to suitable nodes. The suitability should then be determined e.g. by assessing the memory space available to prevent OOM errors caused by too many connections. It should also check for average CPU usage of a node since this is a potential bottlenecks for simple `SELECT` queries.

Range Partitioning for Inserts

Transactions are committed faster if they are fully executed on only one node. This is, these transactions do not have to travel across the network and queue for CPU compute on a second node. An option to force queries to be executed on the parent node is to enable range partitioning of a pre-defined workload and insert directly into a designated worker node. This way transfer of data across the network is avoided which is particularly beneficial since the network *always* introduces latency and on top of that the network could be saturated during large workloads. Another way is to look at optimizations that Aurora and AlloyDB implemented to reduce network I/O. For example only writing WAL directly to storage without keeping any buffers, while asynchronously issue all the writes on disk. However, the latter options requires to redesign the whole architecture of Citus and therefore is infeasible.

Improving the performance of a read-heavy workload

For the 100% `SELECT` workload Citus can optimize at least on the parent node which functions as a coordinator for a read transaction. These transactions are merely CPU intensive

and more then 800 connections does not improve the performance of a read-heavy workload. However, it is evident that the performance of Citus becomes better when network hops and I/O by swapping pages are minimized. Thus to increase the performance of reads on Citus, straightforward tuning such as increasing the buffer cache should be experimented with. Since reads on Citus consume lots of CPU, one should research why this is and how the CPU usage could be reduced. The first findings show that within a connection or transaction, switching threads (`finish_task_switch`) causes most overhead during a saturated CPU. PostgreSQL functions that search in cache or search for hash values are also relatively expensive. The most overhead from Citus specific functions during a read workload is caused by connection handling as opposed to the execution of the transaction itself. Other possible optimizations for a 100% `SELECT` workload lie in page swapping techniques such as smart prefetching from main memory for Postgres.

Cluster Configuration & Tuning

This thesis exposes that connection management is crucial to obtain a good performance while using Citus. However, the main challenge is then to manage and adapt the amount of connections for different workloads. Extensive documentation about this phenomenon or, perhaps in the far future, automated connection management will be beneficial for users. Moreover, to improve the scalability of the `INSERT` workload, the `wal_writer_delay` and the `wal_writer_flush_after` GUC parameters should be experimented with to see if different settings can improve the TPS for `INSERTS`. In addition, the binary protocol does not increase the performance significantly with the YCSB workload. Moreover, execution times are not properly measured on the child (worker) node for a certain transaction. This should be fixed in any future versions of Citus or PostgreSQL.

Investigating more Workloads

For this thesis, we only extensively investigated the behavior of Citus under workload a and workload c from YCSB. Considering the performance impact in the different amount of connections, the optimal amount of connections for other workloads should be determined. Moreover, YCSB is just one of many available benchmark tooling. Looking at different workload types to investigate how much impact the connections management has would therefore be an important goal.

7

Conclusion

The aim for this thesis is to find the bottlenecks of Citus 11 when a Citus cluster size increases. The corresponding research question was "**what are the bottlenecks of Citus 11 when a Citus cluster size or workload increases?**". To compose an answer for this aim, we first had to instrument and analyze metrics in a distributed system under test (SUT), without causing significant overhead. This led to research question 1.1:

How can we instrument and analyze metrics in a distributed system under test (SUT), without causing significant overhead?
To do so, we created a novel infrastructure that runs a simple Create, Read, Update, Delete workload by means of using YCSB. Instead of monitoring every executed transaction, we log a fraction of the executed transactions by means of a second PostgreSQL user '*Monitor*'. By only sampling a fraction of the transactions, the performance is not significantly impacted and the PostgreSQL logs remain compact. Moreover, we run `iostat` with a frequency of 1 Hz in a distinct process on every worker node while executing a benchmark run. Lastly, we collect the data generated by iostat and all the PostgreSQL logs to analyze this by aggregating the iostat values and PostgreSQL logs. To test whether bottlenecks occur we scale out Citus cluster until the performance does not increase significantly and decreases eventually. We vary different parameters and settings, such as the amount of connections to the cluster, the amount of nodes, the binary protocol and the workload.

How does Citus perform in comparison with AWS Aurora and Google Spanner?
Another goal of this thesis is to compare the performance of Citus to its two main competitors; AWS Aurora and Google Cloud Spanner. We found that Aurora performs better on all workloads compared to both Citus and Spanner, *except* for inserts. Aurora reaches

around 700k TPS for the 100% `SELECT`. This is very high considering that Citus does not reach this TPS of one Aurora instance even with 32x16vCpu worker nodes. Spanner does not perform well at all on all workloads, but the warm-up time at the benchmarks for Spanner is not taken into account. Citus however outperforms spanner on every YCSB workload and greatly outperforms Aurora on the 100% `INSERT` workload. It remains important to realize that different database optimizations benefit from one workload at the expense of others.

Which bottlenecks occur when the Citus cluster size increases?
With this novel infrastructure, we found various answers on which bottlenecks occur when the Citus cluster size increases. We found that a workload consisting of 100% `SELECT` queries, i.e. YCSB workload c, behaves as CPU bound. However, it never reaches a full 100% of CPU usage, indicating that the system is likely bound by context switching overhead. With more then 800 connections for `READS`, the performance seems to drop 4. For this workload there is room for improvement in the code that is executed on the the worker node that coordinates the query. Moreover, the most optimal amount of connections for the best performance for 100% `SELECT` workload is 800 for a large ($>$ 4 worker nodes) Citus cluster. The 100% `INSERT` workload is not CPU bound since transactions often are stalled while waiting until WAL is flushed to disk. More connections tend to improve insert performance and no limit is found for the Citus cluster itself. Connections are however memory restricted and the optimal performance is achieved with a right balance between the amount of nodes and connections. With too few nodes in a Citus cluster, the main bottleneck becomes disk I/O for `INSERTS`.

Future Work
Citus introduces overhead to coordinate a distributed query and due to the single-threaded nature of PostgreSQL it is very much connection dependent. However, if one enables to directly route a query to the right worker node, network hops will be avoided and a query can be directly executed on that node. However, to do so, one could consider a 'smart' JDBC driver that enables to steer a connection to the shard owner directly. The JDBC driver has to somehow know in this case which node to steer the connection to. This could be e.g. by maintaining a lookup table or syncing with metadata that holds information on where which shard resides. The overhead introduced by coordinating the distributed query on the parent node could also be decreased in order to efficiently execute `SELECT` queries, however, to determine which functions this entails exactly more investigation is

needed. In particular, currently prepared statements in Citus do not work internally and implementing this feature should speedup an internal Citus query and herewith the total TPS. For a 100% `INSERT` workload the cluster is memory limited. To enable further scaling for inserts it is thus necessary to work on the efficiency of a single connection in Citus, but this would be very challenging and requires changes in the PostgreSQL architecture.

References

[1] Brendan Gregg. **Thinking Methodically about Performance: The USE method addresses shortcomings in other commonly used methodologies.** *Queue*, **10**(12):40–51, 2012. iii, 33, 36, 37

[2] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. **A queueing network model for performance prediction of apache cassandra**. 2016. iii, 40, 41, 95

[3] B.F Cooper. **Yahoo! Cloud Serving Benchmark (YCSB)**. `https://github.com/brianfrankcooper/YCSB/wiki`, 2010. vii, 31

[4] Michael Stonebraker and Lawrence A Rowe. **The design of Postgres**. *ACM Sigmod Record*, **15**(2):340–355, 1986. 1

[5] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. **Amazon aurora: Design considerations for high throughput cloud-native relational databases**. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017. 1, 24, 25

[6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. **Spanner: Google's globally distributed database**. *ACM Transactions on Computer Systems (TOCS)*, **31**(3):1–22, 2013. 1, 26

[7] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. **Citus: Distributed PostgreSQL for Data-Intensive**

Applications. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2490–2502, 2021. 1, 29

[8] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. **Ycsb and tpc-h: Big data and decision support benchmarks**. In *2014 IEEE International Congress on Big Data*, pages 800–801. IEEE, 2014. 2, 30

[9] Yingjie Shi, Xiaofeng Meng, Jing Zhao, Xiangmei Hu, Bingbing Liu, and Haiping Wang. **Benchmarking cloud-based data management systems**. In *Proceedings of the second international workshop on Cloud data management*, pages 47–54, 2010. 2, 29

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. **Benchmarking cloud serving systems with YCSB**. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010. 2, 30

[11] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. **Benchmarking scalability and elasticity of distributed database systems**. *Proceedings of the VLDB Endowment*, **7**(12):1219–1230, 2014. 2, 12, 39

[12] Edgar F Codd. **A relational model of data for large shared data banks**. *Communications of the ACM*, **13**(6):377–387, 1970. 5

[13] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. **A comparison of approaches to large-scale data analysis**. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178, 2009. 5

[14] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an embeddable analytical database**. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019. 6

[15] Baktagul Imasheva, Nakispekov Azamat, Andrey Sidelkovskiy, and Ainur Sidelkovskaya. **The practice of moving to big data on the case of the nosql database, clickhouse**. In *World Congress on Global Optimization*, pages 820–828. Springer, 2019. 6

[16] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. **The snowflake elastic data warehouse**. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016. 6

[17] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. **OLTP through the looking glass, and what we found there**. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 409–439. 2018. 6, 30, 33, 65

[18] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017. 6, 7, 8, 10, 12, 13, 19, 39, 78

[19] Narayanan Venkateswaran and Suvamoy Changder. **Simplified data partitioning in a consistent hashing based sharding implementation**. In *TENCON 2017-2017 IEEE Region 10 Conference*, pages 895–900. IEEE, 2017. 7

[20] Mazedur Rahman, Samira Iqbal, and Jerry Gao. **Load balancer as a service in cloud computing**. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 204–211. IEEE, 2014. 8

[21] Rahul Soni. *Nginx*. Springer, 2016. 8

[22] Sasalak Tongkaw and Aumnat Tongkaw. **A comparison of database performance of MariaDB and MySQL with OLTP workload**. In *2016 IEEE conference on open systems (ICOS)*, pages 117–119. IEEE, 2016. 13

[23] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. **The implementation of POSTGRES**. *IEEE transactions on knowledge and data engineering*, **2**(1):125–142, 1990. 14

[24] M David Hanson. **The client/server architecture**. In *Server Management*, pages 17–28. Auerbach Publications, 2000. 14

[25] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a database system.* Now Publishers Inc, 2007. 14, 19

[26] PostgreSQL Global Development Group. **PostgreSQL Documentation**. 2010. 17

[27] Michael Stonebraker. **The case for shared nothing**. *IEEE Database Eng. Bull.*, **9**(1):4–9, 1986. 18

[28] Erhard Rahm. **Parallel query processing in shared disk database systems**. *ACM SIGMOD Record*, **22**(4):32–37, 1993. 24

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. **Bigtable: A distributed storage system for structured data**. *ACM Transactions on Computer Systems (TOCS)*, **26**(2):1–26, 2008. 26

[30] M Pasumanskyl. **Inside capacitor, bigquery's next-generation columnar storage format**. *Google Cloud Blog*, 2016. 26

[31] Sheldon Ross. *Simulation: Fifth Edition*. San Diego: Academic Press., 2012. 27

[32] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. **SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python**. *Nature Methods*, **17**:261–272, 2020. 28

[33] Avinash Lakshman and Prashant Malik. **Cassandra: a decentralized structured storage system**. *ACM SIGOPS Operating Systems Review*, **44**(2):35–40, 2010. 29, 35

[34] Lars George. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.", 2011. 29, 35

[35] Jianfeng Zhan, Rui Han, and Roberto V Zicari. *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 6th Workshop, BPOE 2015, Kohala, HI, USA, August 31-September 4, 2015. Revised Selected Papers*, **9495**. Springer, 2016. 30

[36] Peter Boncz, Thomas Neumann, and Orri Erling. **TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark**. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013. 30

[37] Rim Moussa. **Tpc-h benchmark analytics scenarios and performances on hadoop data clouds**. In *International Conference on Networked Digital Technologies*, pages 220–234. Springer, 2012. 30

[38] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. **An overview of decision support benchmarks: TPC-DS, TPC-H and SSB**. *New Contributions in Information Systems and Technologies*, pages 619–628, 2015. 30

[39] Pinar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. **From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored**. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 17–28, 2013. 30

[40] Tomas Kalibera, Jakub Lehotsky, David Majda, Branislav Repcek, Michal Tomcanyi, Antonin Tomecek, Petr Tuma, and Jaroslav Urban. **Automated benchmarking and analysis tool**. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, pages 5–es, 2006. 31

[41] Martin Grambow, Fabian Lehmann, and David Bermbach. **Continuous benchmarking: Using system benchmarking in build pipelines**. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 241–246. IEEE, 2019. 31

[42] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. **Diametrics: benchmarking query engines at scale**. *ACM SIGMOD Record*, **50**(1):24–31, 2021. 31, 32

[43] Christoph Boden, Alexander Alexandrov, Andreas Kunft, Tilmann Rabl, and Volker Markl. **PEEL: A framework for benchmarking distributed systems and algorithms**. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 9–24. Springer, 2017. 31, 32, 33, 34, 65, 69

[44] Thomas Bodner, Tobias Pietz, Lars Jonas Bollmeier, and Daniel Ritter. **Doppler: understanding serverless query execution**. In *Proceedings of The International Workshop on Big Data in Emergent Distributed Environments*, pages 1–4, 2022. 31

[45] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. **Dapper, a large-scale distributed systems tracing infrastructure**. 2010. 33, 34, 35, 63, 76

[46] Zachary Benavides, Keval Vora, and Rajiv Gupta. **DProf: distributed profiler with strong guarantees**. *Proceedings of the ACM on Programming Languages*, **3**(OOPSLA):1–24, 2019. 33, 35

[47] Lexiang Huang and Timothy Zhu. **tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces**. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 76–91, 2021. 33, 35, 36, 66

[48] Brian K Tanaka. **Monitoring virtual memory with vmstat**. *Linux Journal*, **2005**(140):5, 2005. 33

[49] Scott Helvick. **A Survey of Hardware Performance Analysis Tools**, 2008. 33

[50] Brendan Gregg. **Linux performance analysis and tools**. Technical report, Technical report, Joyent, 2013. 34

[51] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019. 34

[52] Thomas Neumann. **Evolution of a compiling query engine**. *Proceedings of the VLDB Endowment*, **14**(12):3207–3210, 2021. 34

[53] Arnaldo Carvalho De Melo. **The new linux'perf'tools**. In *Slides from Linux Kongress*, **18**, pages 1–42, 2010. 34

[54] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. **Using Magpie for request extraction and workload modelling.** In *OSDI*, **4**, pages 18–18, 2004. 34

[55] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. **{X-Trace}: A Pervasive Network Tracing Framework**. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, 2007. 34

[56] Christian Stuart. *Profiling Compiled SQL Query Pipelines in Apache Spark*. PhD thesis, Master's thesis. Universiteit van Amsterdam, 2020. 34

[57] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. **lprof: A non-intrusive request flow profiler for distributed systems**. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 629–644, 2014. 35

[58] Aidi Pi, Wei Chen, Xiaobo Zhou, and Mike Ji. **Profiling distributed systems in lightweight virtualized environments with logs and resource metrics**. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 168–179, 2018. 36, 66

[59] Philippos Papaphilippou and Wayne Luk. **Accelerating database systems using FPGAs: A survey**. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–1255. IEEE, 2018. 36

[60] Jared Casper and Kunle Olukotun. **Hardware acceleration of database operations**. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 151–160, 2014. 36

[61] Kay Ousterhout. *Architecting for Performance Clarity in Data Analytics Frameworks*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2017. 38, 98

[62] A Vijay Srinivas and D Janakiram. **A model for characterizing the scalability of distributed systems**. *ACM sigops operating systems review*, **39**(3):64–71, 2005. 39

[63] Prasad Jogalekar and Murray Woodside. **Evaluating the scalability of distributed systems**. *IEEE Transactions on parallel and distributed systems*, **11**(6):589–603, 2000. 39, 40, 46

[64] Alex Delis and Nick Roussopoulos. **Performance comparison of three modern DBMS architectures**. *IEEE Transactions on Software Engineering*, **19**(2):120–138, 1993. 40

[65] Yan Qiang, Yi Li, and Junjie Chen. **The workload adaptation in autonomic DBMSs based on layered queuing network model**. In *2009 Second International Workshop on Knowledge Discovery and Data Mining*, pages 781–785. IEEE, 2009. 40

[66] Rasha Osman, Irfan Awan, and Michael E Woodward. **Application of queueing network models in the performance evaluation of database designs**. *Electronic Notes in Theoretical Computer Science*, **232**:101–124, 2009. 41

[67] Firas Alomari and Daniel A Menasce. **Efficient response time approximations for multiclass fork and join queues in open and closed queuing networks**. *IEEE Transactions on Parallel and Distributed Systems*, **25**(6):1437–1446, 2013. 41

[68] Peter J Denning. **Thrashing**. In *Encyclopedia of Computer Science*, pages 1776–1777. 2003. 92

[69] Stephen Hemminger et al. **Network emulation with NetEm**. In *Linux conf au*, **5**, page 2005. Citeseer, 2005. 98

Appendix

7.1 Example YCSB Insert

```
INSERT INTO usertable (YCSB_KEY,field1,field0,field7,field6,field9,field8,
field3,field2,field5,field4) VALUES('user82957131023132999O9', '+)p?3.-18
4-|%,48-|''(r"?8!Bq>Oa>%6%K1/4~;4.&Ee#4.%D.,8:X#@w/Mg/-$+)‘27b<A!-\{,N1
(H? !b>/  D#%];$\'',','$+<!Vs*7|;D/:?|"Oj9@+,7> Yc3T6]w?C54C/ORq-5d''W};;
~2.j#<z6)|)Ys2W; (:%Si?%$=&z7/<9.h:Ja.Fy;Ra7?p>?v:',',3l2%|,B/%L1?''21Fy&9
!]e4X+A=*P''7 /?$9I''(9h''#l,%|77(04n+=|;)f#@{+G5%R:.d?>8*Qo*3j5:.(^;,&v
>92:U}6',’8321>h41*!\.X{;,<!$$1Cs;,4<[{:681[k&_=59‘2?:%^}#M/-3,''[y73.68b
#By1. 9S3]#Ov)C3<!&,+v.U''+,.<:t7A)5', ’65$$-~>Ok*Hc9Zk7Pu:3t&Le"6n3Y#>Oc
/>,4R74@a:=v7!,7>p6P37X!9.*!5* 8x-;d.J)9Z''!"8+.(<.4-4f:Xg3Ma6[98''d"', ’
4:d;94-=f>4t/=‘#VkO>h;U6K{2Ia$Ra&>&O[1)9v6)x79r2Vw&3h.;>5K;3>",)*$:j''>v
1Ng3=2OU?<Tu3\q:J!>P;$5~:"*5', ’2L#,.,''G5!Js<J)<F99M''5:*.Dw:*f828!,$!!r’
’:4,M/OOl.L1,Q;%J)=)h;,h#;4-;<$]u1" %?t:Uq:=h+Wy?Q#:8|2U=4H'',', ’’’_+$F-
(_o2R!%)d2Zm>#41A''3W35L3<?j&9~9":= ,"C%4Ma.Sa&T!="f274,*&!G'' 6t* &1&.-L
92O(9":+P1-X3!Qg&,xOBc:', ’8[%1P''(:n;Fm,H}>/h2L+4&l:)‘/*j*1r?Xq4Xi29x#Y{
2-2+?n>%z=\s/K=;I78G{$Ro*$$(^!$M!27v2''2OT-$Dm",r*A/6#b;', ’’’K{6[oODk2R=’
’8, 0‘OJa3@w "t7)p(Ds*$l> *$M}2Bc3Bq5O%!L)%U;1''.&7h:Fc:A#<Ys!7"1&t:@1!E1>
H34>d5S!1U56Q+9’)
```

7.2 PostgreSQL logs

The structure in the `pg_log` is determined in the `postgresql.conf` file. The logging schema that we use is shown in the code snippet below.

```
log_line_prefix = ‘%m [%p][%v] : [%l-1] %q[app=%a] ’
```

This schema consists of the timestamp (`%m`), process id [3392] (`%p`), virtual transaction ID

(backendID/localXID) [11/245] (%v), Number of the log line for each session or process (%l-1), always starting at [2-1], %q produces no output, but tells non-session processes to stop at this point in the string, application name (%a) and the duration is printed. An example of a resulting log is listed below.

```
2022-05-30 07:35:12.035 UTC [3392][11/248] : [12-1]
[app=%a]: [app=PostgreSQL JDBC Driver] LOG:  duration:  0.305 ms
2022-05-30 07:35:12.037 UTC [3392][11/248] : [13-1]
[app=PostgreSQL JDBC Driver] LOG:  duration: 1.878 ms
2022-05-30 07:35:12.037 UTC [3392][11/248] : [14-1]
[app=PostgreSQL JDBC Driver]
LOG:  execute <unnamed>: INSERT INTO usertable
```

When applying the log structure to above log snippet, this means that we get the following:

```
%m: 2022-05-30 07:35:12.035 UTC
[\%p]: [3392]
[%v]: [11/248]
[%l-1]: [12-1]
[app=%a]: [app=PostgreSQL JDBC Driver]
```

A group of logs for one query can be can be identified by the same loxalXID (e.g. [11/248] in the example). Using the local id and same process id (id of connection) we can calculate the total execution time for the query. However, since we are merely interested in the average or percentiles of execution times we aggregeate all values.

```
2022-05-30 07:36:12.314 UTC [8805][11/807] : [36-1]
[app=PostgreSQL JDBC Driver] LOG:  duration: 0.046 ms
2022-05-30 07:36:12.315 UTC [8805][11/807] : [37-1]
[app=PostgreSQL JDBC Driver] LOG:  duration: 0.076 ms
2022-05-30 07:36:12.315 UTC [8805][11/807] : [38-1]
[app=PostgreSQL JDBC Driver]  LOG:  execute S_1:
SELECT * FROM usertable WHERE YCSB_KEY = $1
2022-05-30 07:36:12.315 UTC [8805][11/807] : [39-1]
[app=PostgreSQL JDBC Driver] DETAIL:  parameters:
$1 = 'user8008620038384188605'
2022-05-30 07:36:12.316 UTC [8805][11/807] : [40-1]
[app=PostgreSQL JDBC Driver] DEBUG:  task execution (0)
for placement (5) on anchor shard (102012) finished
in 757 microseconds on worker node
private-w4.5w32-s-Citus-0-nohup-30-07-21-40.marlin-development.com
:5432 2022-05-30 07:36:12.316 UTC [8805][11/807] : [41-1]
[app=PostgreSQL JDBC Driver] LOG:  duration: 0.891 ms
\end{lstlisting}
```