DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Benchmarking Distributed Database Performance and Dependability under Partial System Failures

Author:
Ruben BES

Supervisor:
Assis. Prof. Asterios KATSIFODIMOS

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

March 25, 2021

Declaration of Authorship

I, Ruben BES, declare that this thesis titled, "Benchmarking Distributed Database Performance and Dependability under Partial System Failures" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Benchmarking Distributed Database Performance and Dependability under Partial System Failures

by Ruben BES

Many types of database management systems exist, but finding the one that is right for a specific use case is becoming increasingly more difficult. Benchmarks allow one to compare various systems, but in a world where distributed DBMSs are increasingly used for mission critical purposes, we find most existing benchmarks neglect fault tolerance and dependability aspects.

In this Master's Thesis, we design a modular and highly extensible framework capable of introducing partial system failures in a distributed database deployment. We also implement a proof-of-concept version of our framework which we use to evaluate the performance of a CockroachDB cluster deployed through Kubernetes, by running the TPC-C benchmark while we inject faults and measure changes in performance. Using this proof-of-concept implementation we demonstrate the faults our system can introduce and find that the impact of our high-level node failures is strongly dependent on the time a node has to perform a graceful shutdown and notify its peers or connected clients.

Contents

List of Figures

List of Tables

List of Abbreviations

DBMS	**D**ata**b**ase **M**anagement **S**ystem
RDBMS	**R**elational **D**ata**b**ase **M**anagement **S**ystem
SDMS	**S**treaming **D**ata **M**anagement **S**ystem
MR	**M**ap**R**educe
KV	**K**ey-**V**alue
SQL	**S**tructured **Q**uery **L**anguage
NoSQL	**N**ot only **SQL**
ACID	**A**tomicity **C**onsistency **I**solation **D**urability
BASE	**B**asic **A**vailability **S**oft state **E**ventually consistent
DBA	**D**ata**b**ase **A**dministrator
OLTP	**O**nline **T**ransaction **P**rocessing

Chapter 1

Introduction

Database Management Systems (DBMSs) come in many flavours, and asking which is the best will yield an answer commonly given in computer science: *it depends*. There is no one-size-fits-all solution, as the best approach is highly dependent on the problem at hand and which properties are deemed important. To aid the decision-making process, benchmarks have been created for many types of systems, including DBMSs. Benchmarks test a system by running a workload designed to either match real-world scenarios or stress specific parts of the system in order to determine performance levels and other relevant properties.

Ultimately, one looks for the database that has the best performance for the intended use case, so it is understandable that database benchmarking looks mostly at performance aspects like throughput and latency. However, in a world where distributed databases are increasingly used for mission critical purposes, it is important to look at fault tolerance as well. In these complex systems, there are many components that can malfunction or operate sub-optimally; systems could experience hardware failures or network jitter. This is why many systems are designed to withstand various faults without loss of data or availability, but it is often only through experimentation that one learns how performance and other properties are affected under these circumstances.

This thesis aims to evaluate dependability aspects in the benchmarking of distributed database systems, by introducing partial system failures in the system under test and measuring how benchmark results are affected.

1.1 Problem Statement

The demand for systems that can scale beyond the capabilities of traditional relational DBMSs has led to a significant growth in the number of available systems, each with their own benefits and drawbacks. Many of these are considered experimental or not mature enough, yet some are already used in production environments because of the potential they offer. Meanwhile, the benchmarks that help in choosing the right system still focus on the same properties, neglecting the dependability aspects that are of great importance in any distributed system. This is somewhat justified as these systems use fault tolerance techniques that are theoretically sound and practically proven to keep the system available without endangering the data consistency. The problem, however, is that not much is known about how other aspects are affected, most important of which are the effects on throughput and latency.

1.2 Research Questions

Many systems can tolerate faults which enables them to continue serving users or completing requests. While we know that a system can stay operational without loss of data, we have little knowledge of how performance is affected by faults and the system's recovery mechanisms. This problem leads us to our main research question:

RQ 1 *How do faults affect the performance of distributed database systems?*

This question is very broad on purpose, because we intend to find a general answer to the problem. We realise however that evaluating all distributed database systems is impossible, which is why we must find a generic solution and apply it to a specific problem instance. Additionally, we must be able to cause faults ourselves and measure their impact on performance in order to understand the severity of failures in these types of systems. These points lead us to our subquestions:

RQ 1.1 *How can we agnostically evaluate the impact of faults on various distributed database systems?*

RQ 1.2 *How can we reliably introduce and reproduce various types of faults?*

RQ 1.3 *How can we non-intrusively measure the impact of faults?*

RQ 1.4 *What is the performance impact of a high-level failure in a distributed database deployment?*

1.3 Project Goal and Scope

We wish to study the effects that faults can have on the performance of distributed database management systems. In order to do so, we envision a framework that is capable of causing or simulating various types of faults. Performance should be evaluated with a benchmarking framework that applies a workload to the DBMS. The framework should be highly extensible in order to support any database management system, deployment technique and benchmarking framework.

We realise that creating such a framework is an ambitious endeavour and have decided to create a design for the full framework but implement a proof-of-concept version instead. The goal of this limited implementation is to show that the envisioned framework is a feasible approach for introducing faults and evaluating the performance impact in DBMSs. The framework is designed with focus on reusability, reproducibility and extensibility. The proof-of-concept implementation will provide a basis for the full framework and should be able to introduce high-level node failures and use the OLTP-Bench benchmarking framework to evaluate a CockroachDB cluster deployed through Kubernetes.

The scope of this research is covered in more detail in section 4.1, and details regarding the proof-of-concept implementation are covered in section 5.1.

1.4 Main Contributions

The goal of this research is to learn how performance aspects of distributed database systems are affected by various types of faults. We intend to do so by designing a framework that can inject faults in a system-agnostic way, implementing a proof-of-concept framework that can serve as a foundation for future iterations of the framework, and using this version to evaluate a test system. Our main contributions can be summarised as follows:

- A design for a highly extensible framework capable of evaluating the impact of faults in distributed database systems in a system-agnostic manner.
- A proof-of-concept implementation for introducing a number of high-level faults in a distributed database deployment.
- Experiments detailing how the performance of a CockroachDB deployment is impacted by high-level node failures.

1.5 Research Methodology

A number of topics were researched in order to write this thesis. The related works were found through the ACM Digital Library,[1] dblp computer science bibliography[2] and Google Scholar.[3]

We first researched the topic of **Database Benchmarking** to learn about the various benchmarks and frameworks that exist. Next, we looked into **Distributed Databases** and **Database Fault Tolerance** to study how these systems differ from traditional DBMSs and how they manage to tolerate faults. This lead us to research **Database Dependability** and **Fault Injection** in order to learn how the dependability of database systems was tested and how we could introduce faults ourselves. Lastly, we looked into **Benchmarking Database Dependability** and **Database Fault Injection** to find works on benchmarking database systems and their dependability by injecting faults.

1.6 Collaboration

This thesis was performed in collaboration with the company Adyen,[4] a payment provider based in Amsterdam. As the number of transactions performed worldwide increases steadily, scalable and fault-tolerant solutions become ever more important. Traditional database management systems do not scale indefinitely but many of the distributed solutions have not reached the desired level of maturity yet. This research would help Adyen evaluate various systems to find a distributed database system and configuration that satisfies the level of performance and resilience they require.

1.7 Thesis Structure

The rest of this thesis is organised as follows. We first cover some background knowledge on database management systems, benchmarking and system dependability in chapter 2. Next, academic work related to database benchmarking and fault injection is discussed in chapter 3. The scope of this research and design of the

[1] ACM Digital Library website: `https://dl.acm.org/`
[2] dblp computer science bibliography website: `https://dblp.org/`
[3] Google Scholar website: `https://scholar.google.com/`
[4] Adyen homepage: `https://www.adyen.com/`

framework are covered in chapter 4, and decisions regarding the proof-of-concept framework as well as a number of implementation details are explained in chapter 5. The experiments are covered in chapter 6 and chapter 7 discusses the applications of this framework and covers future work. Finally, we answer the research questions and conclude this thesis with a summary in chapter 8.

Chapter 2

Background Knowledge

This chapter provides some background information on a number of concepts that this thesis builds upon. We first discuss the various types of database systems in section 2.1 and then cover database benchmarking in section 2.2. Lastly, section 2.3 provides a brief introduction to fault injection.

2.1 Database Systems

Databases exist in many different shapes and forms. Some are intended as general purpose data storage while others are highly specialised with a single application in mind. We first cover the different types of database systems, then discuss the aspects in which distributed DBMSs differ from the traditional single-machine database systems and finally cover the general anatomy of a DBMS.

2.1.1 Database Types

Roughly speaking, three categories of database management systems exist: traditional relational DBMSs, NoSQL data stores and NewSQL DBMSs. Each came forth out of needs left unfulfilled by its predecessor. Two closely related types of systems are streaming data management systems (SDMS) and MapReduce (MR) frameworks, but these are not truly considered to be database systems because their focus lies on event and batch data processing, respectively. While these systems will not be covered here, we felt the need to mention them because we discuss a number of benchmarking frameworks that target these systems in subsection 3.1.2.

Relational

The traditional relational database systems consist of tables where each entry is a row and each column is a property of this entry. An entry can be related to another, which is often done by including an identifier as a property or constructing additional tables just for linking purposes. The main strength of these traditional systems is the use of the Structured Query Language (SQL) to perform relational algebra using the database schema, which predefines the properties and relations of entries, but not their values. Combining this with ACID guarantees (Atomicity, Consistency, Isolation and Durability) allowed for systems to be used for many business critical applications.

NoSQL

NoSQL data stores were developed as a reaction to the limited scalability of RDBMSs, which could only scale vertically (and with reduced returns), and are known for their ability to scale horizontally by adding additional machines instead of upgrading hardware. As always, there is no free lunch in computer science, so NoSQL systems pay the price by relaxing ACID guarantees, instead adopting BASE (Basic Availability, Soft state, Eventually Consistent), making them less suitable when a high degree of consistency is required. However, by employing many machines and distributing or replicating data over these, NoSQL stores are capable of providing unparalleled availability.

Another important distinction is the data model used by NoSQL databases. Instead of the relational model, NoSQL systems use models based on key-value (KV) pairs. The most basic is the actual KV-pair, which is a unique key that links to a value. Slightly more complex are the document models, which still use KV-pairs, but the value is a JSON document instead. A different approach is using column-families, in which a key indicates a row and the value is a set of column families, each of which also acts as a key for one or more columns it holds, where each column consists of a name-value pair. Lastly, graph databases hold objects whose properties and relations to other objects are in the form of key-value pairs.

NewSQL

NewSQL is the attempt to combine SQL and the ACID properties of RDBMSs with the scalability of NoSQL systems, representing a best-of-both-worlds solution. At the time of writing these systems are often immature, experimental or academic, employing new system architectures or communication models, but they are slowly gaining traction. While users mostly interact with NewSQL databases using SQL and the relational model, different data models such as key-value pairs are often used under the hood for replication purposes.

2.1.2 Distributed Database Systems

Traditional relational database systems typically run on a single machine, but NoSQL and NewSQL systems consist of multiple instances that work together to provide greater scalability. These systems are often deployed over virtual machines which may reside on the same physical machine, so we refer to the instances as nodes instead of machines. Nodes that are deployed together cooperate and form a cluster that acts as a single unit.

These systems present themselves as a single logical application to the outside world, but require a number of additional techniques or components to achieve this. Most importantly, data has to be replicated and consensus must be reached on operations, otherwise nodes could diverge and queries could return different results depending on the node that processed it. Additionally, connections and operations should be load balanced to prevent nodes from becoming overloaded while others remain idle.

2.1.3 Database Anatomy

While the exact anatomy differs per database system, they share a number of components. These parts of the system, often called engines or layers, provide the general functionality required for a database. A schematic overview can be found in Figure 2.1.

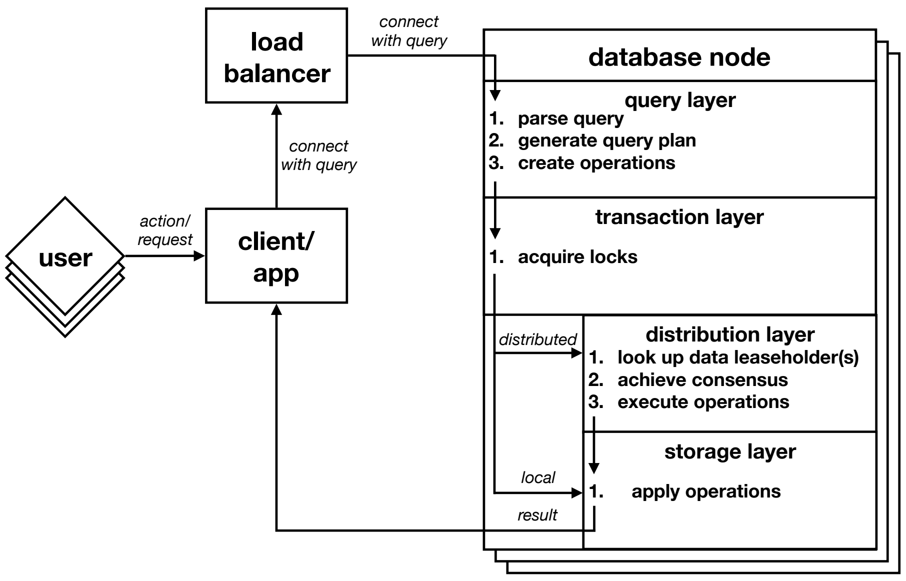


FIGURE 2.1: A high level schematic overview of the anatomy of a (distributed) database management system

A client is used to connect users or applications to the system. Load balancers are often used to spread users over the system in case it is distributed. Users send requests through the clients, after which the system performs that request. The query layer takes these requests, translates them into operations and passes them on to the transaction layer. This layer handles locking and concurrency, and performs the operations. In the distributed case, operations are often first passed to a layer responsible for achieving consensus on these operations instead. The storage layer then persists data to disk and returns requested data. Distributed database systems often include a specific layer for replication, or replicate data through their consensus protocols automatically.

2.2 Benchmarking

In computing, benchmarks are used for assessing the performance of a component or system. Vendors use benchmarking for marketing purposes and identifying (software) bottlenecks, while customers use benchmarks and the results reported by vendors to choose a product from a vendor. Academics often develop new solutions and then use benchmarks to evaluate these.

A number of organisations exist to create proper benchmarks and ensure these are performed correctly, and the most well known for database benchmarking is the Transaction Processing Performance Council (TPC).[1] Founded in 1988 by vendors of various database systems, their aim was to create thorough benchmarks and enforce vendors to produce extensive reports.

2.2.1 Properties of a Benchmark

A benchmark that is performed on a database system consists of three components: a dataset, a workload and a set of metrics. By loading the dataset onto the system

[1]Transaction Processing Performance Council homepage: `http://www.tpc.org`

and then applying the workload, one can learn relevant metrics which can then be used to compare this system to others.

Some benchmarks are made to be generic, making it possible to apply them to a wide range of systems and establish a baseline performance, which has the advantage that it allows the comparison of a broad spectrum of databases. Many (database) systems are however designed with a more specific purpose in mind, meaning generic benchmarks will not fully capture their capabilities. To properly evaluate these more specialised database systems, domain-specific benchmarks are needed.

Jim Gray, who defined the first TPC benchmark, names four key criteria for domain-specific benchmarks to be useful (Gray, 1993). These criteria are:

- **Relevance**: The benchmark workload should be representative of the typical operations performed in the problem domain
- **Portability**: It should be easy to implement the benchmark both on and for different systems
- **Scalability**: The benchmark should be applicable to different system sizes and it should be scalable
- **Simplicity**: The benchmark, workload and especially the results should be understandable

Sometimes benchmarks are referred to as frameworks or suites. These are often a collection of multiple benchmarks wrapped into a single environment, making it easier for users to perform multiple different benchmarks on a single system or perform the same benchmark on multiple different systems. In addition to housing multiple benchmarks, these frameworks often feature automated system deployment and/or configuration as well.

2.2.2 Metrics

When running a benchmark, we are interested in learning certain characteristics of the systems we are testing. Measures and metrics are often used interchangeably, but do not actually mean the same thing. Measures are numbers derived from measurements such as a person's height or weight, while metrics are calculations between measures, like the Body Mass Index. Metrics can be useful because they combine multi-dimensional information into a single dimension making it easier to compare systems, but a metric can also be misleading when the information is not normalised or weighted properly.

Performance is the main metric of interest in benchmarking, but what this constitutes is dependent on the context. In the case of database systems, this is expressed in terms of throughput and latency. Throughput is the number of requests the system can handle per second and latency is the time it takes to process such a request.

Generally, one looks for the system that provides the greatest throughput with an acceptable latency, but depending on the context one might be interested in more granular metrics or entirely different aspects. Some systems excel in their write capabilities so including read operations in the performance metrics would not paint the right picture. Similarly, many distributed DBMSs rely heavily on achieving consensus on data or operations, in which case the replication delay or network utilisation is more important.

Besides the aforementioned performance related metrics, benchmarks often collect basic hardware information as well. Examples are CPU and RAM utilisation, although some properties may depend on the domain. For example, it makes little sense to look at disk IO when testing an in-memory database. Other metrics that are interesting in some cases are data consistency, system cost, system scalability and elasticity, availability and data load time.

2.3 System Dependability

Dependability is often called robustness or resilience, and includes many different aspects of which most are related to availability, reliability and integrity. The notion of dependability and its terminology have been established in 1980 by the International Federation for Information Processing (IFIP) Working Group 10.4,[2] who defined dependability as "the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers". Besides the aforementioned aspects, dependability also includes safety, confidentiality and maintainability, although the main focus is on providing an available and reliable service that can tolerate faults and prevent catastrophic consequences.

2.3.1 Types of Faults

Faults are the cause of errors, which are incorrect or unintended states. When these cause a service to perform incorrectly and this becomes apparent to users or other systems, we refer to this as a failure.

There are multiple types of faults and they can occur on many different levels, with varying severity and occurrence rates. Some faults can be prevented, while others will just occur at some point in time, requiring fault-tolerating mechanisms in order to mitigate them or reduce their impact.

Hardware Faults

Hardware faults can occur in many different components and are most often related to the power supply, CPU, memory or storage. These faults can be evaluated on the component level or lower. For example, we could consider the entire storage medium to be faulty, or only deem a single block of storage to be corrupted.

Hardware faults are becoming increasingly rare as technology advances, and their impact can be reduced depending on the level and component in which they occur. For example, some faults can be corrected by using bit-flip error correcting codes, or mitigated by using a redundant power supply, or reduced by avoiding corrupted sectors.

Software Faults

Software faults, also known as bugs or defects, are one of the most frequent causes of system failures. Generally, software faults are unintentionally introduced during the development phase and not picked up through testing. They can be classified using various methods, based on their origin and resulting effects.

Software faults can also manifest as hardware faults, for example a bad driver could make a hardware component appear faulty. This means software faults could also be used to effectively simulate hardware faults.

[2] IFIP Working Group 10.4 website: `https://www.dependability.org/wg10.4`

Operator Faults

Sometimes the operator of a system will make a mistake that severely impacts various aspects of the system. We call these mistakes operator faults. An example would be an operator shutting down a functioning node instead of the node that was actually marked to be decommissioned, or wrongly modifying some configuration parameters. These faults are very different from hardware and software faults in that they do not occur spontaneously. Instead, they are human errors introduced precisely the same way regular operator actions are performed.

External Factors

Similar to how software faults can manifest as hardware faults, external factors can make it look like a system is faulty even though the system itself is healthy. For example, the performance of a system can be severely reduced when the network is experiencing a lot of jitter or a load balancer is functioning incorrectly. It is both difficult to classify and prevent or mitigate these scenarios, as one often has no control over these factors.

2.3.2 Testing Dependability

Before testing the dependability of a system, one must first analyse field data of related systems in order to identify representative faults. Whether a type of fault is relevant depends on their location and severity, as well as the rate at which they occur. Many systems employ techniques for tolerating faults, but evaluating these can be difficult as faults are generally very rare. To be able to properly study their effects it is often necessary to artificially introduce faults, which is why fault injection techniques have been developed.

Evaluating the consequences of hardware faults is difficult and costly because it requires physical access and specialised equipment in order to modify the hardware. Instead, software is often used to simulate these hardware faults. Software fault injection can introduce code changes at various levels, which then mimic bugs and cause errors. Operator faults are not actually injected, they are simply introduced through the same channels operators would normally use.

The effects of faults are then studied by monitoring systems or applications. This can be done by checking their logs for signs of anomalies or by running some workload and analysing any performance variations by comparing the results to a baseline level established beforehand.

Chapter 3

Related Work

In this chapter we discuss works related to database benchmarking and fault injection. These works originate from academia as well as the industry, as companies often come up with their own solutions when current technologies do not suffice or research is not available.

3.1 Database Benchmarking

Benchmarking is a widely researched topic, especially when applied to database systems. We first cover the two most well known generic benchmarks, TPC-C and YCSB, after which we discuss a number of works on benchmarking dependability aspects.

3.1.1 Industry Standards

TPC-C and YCSB are two benchmarks that have become true industry standards for evaluating database performance. We briefly cover them in the paragraphs below to learn why they are considered standards and what they do right, in order to aid us in choosing a benchmark framework for our proof-of-concept implementation.

TPC-C

Approved by the TPC in 1992 and still an industry standard today, TPC-C has truly withstood the test of time.[1] It is a generic OLTP (Online Transaction Processing) benchmark that models a retailer processing orders and payments, while shipping products and checking stock in its warehouses. TPC-C measures the performance of a database system based on the number of times per second it can perform the `NewOrder` transaction, while also running four other transactions. Scaling is done by increasing the number of warehouses and the rate at which new requests are submitted.

The TPC-C benchmark has become an industry standard for a number of reasons:

- There is no apparent or exploitable bias in its queries or metrics.
- Vendors must produce an extensive report when publishing results, providing transparency and correctness.
- The TPC wrote a specification instead of an implementation, meaning it can implemented both on and for any system.
- Multiple database vendors helped design the benchmark, implicitly supporting it from the start.

[1]TPC-C benchmark website: `http://www.tpc.org/tpcc/`

YCSB

In 2010, Yahoo! created the Yahoo! Cloud Serving Benchmark (Cooper et al., 2010). The benchmark was intended for benchmarking cloud data serving systems using Key-Value pairs. The YCSB benchmark supports just four operations (insert, update, read and scan) and four access distributions (uniform, zipfian, latest and multinomial), measuring performance as the number of operations performed per second. While relatively simple, the operations and distributions can be combined and run with a specified ratio, and five standard benchmarks are included by default. This simplicity is the main strength of YCSB, as this makes it straightforward to extend the benchmark and adapt it for various purposes, with it being used as a basis by dozens of benchmarks.

The YCSB benchmark has become an industry standard for a number of reasons:

- YCSB was one of the first benchmarks for Key-Value stores.
- It was created by a well known internet company.
- The benchmark is straightforward and highly configurable.
- Its simplicity and extensibility made it an excellent basis for more specific benchmarks.

3.1.2 Dependability Aspects

The two benchmarks discussed so far focus only on performance aspects, but dependability is of major importance for distributed database systems. Here we cover a number of benchmarks and frameworks that evaluate dependability aspects in addition to performance.

DBench-OLTP

DBench-OLTP is a dependability benchmark for OLTP systems that measures both performance and dependability aspects (Vieira and H. Madeira, 2003a). DBench-OLTP first runs a benchmark and then in a second phase injects the faults. It checks the detection time and recovery duration, after which data integrity checks are performed. While advanced for its time, it is only capable of injecting operator faults and supports just the TPC-C benchmark and its metrics, meaning results are limited to transactions per second and availability.

DS-Bench

DS-Bench is a software framework that is part of the DS-Bench Toolset (Fujita et al., 2012). It is used for conducting dependability benchmarks, but is not a benchmark in itself. Instead, it supports the execution of benchmarks supplied by the user through a benchmark description. DS-Bench also supports the generation and injection of faults (referred to by the authors as anomalies) that simulate hardware malfunctions. This software framework is the most similar to our proof-of-concept in that it can be used with any system and benchmark combination, but it is limited to hardware fault injection. Furthermore, in order to use DS-Bench, one must use all parts of the toolset making it less lightweight and more difficult to use.

MRBS

MRBS is a benchmarking suite for MapReduce systems which was later extended to evaluate dependability aspects as well (Sangroya, Serrano, and Bouchenak, 2012a,b, 2016). While running a benchmark, nodes can be crashed by prematurely terminating them through the cloud infrastructure API or killing all MapReduce daemons on that node, and MapReduce tasks are crashed by killing the corresponding processes. Injecting software faults and making tasks hang is possible as well, but this requires creating a synthetic MapReduce library. Unfortunately, this suite's portability is limited as it can only be used for MR systems and requires synthesising the MapReduce API of the system under test.

StreamBench

StreamBench is a framework comprising seven benchmarks and four workload suites targeting various aspects of Stream Data Management Systems (Lu et al., 2014). While not very extensive, one of the workloads evaluates the fault tolerance of an SDMS by intentionally failing one of the nodes and comparing the performance to a previously established baseline. This approach is similar to ours, except it is limited to SDMSs and single node failures.

3.2 Fault Injection

Faults generally have low occurrence rates, making it difficult to study their effects on live systems. In this section we discuss a number of works related to artificially introducing faults. We first cover the three types of faults mentioned in subsection 2.3.1 and then look into virtualisation techniques and chaos engineering.

3.2.1 Hardware Faults

Injecting true hardware faults often requires physical access to a system. One of the first techniques was MESSALINE, a general pin-level fault injection tool consisting of hardware modules for the injection, activation and collection of faults, and a software module for test sequence management (Arlat, Aguera, et al., 1990). The injection module is capable of injecting several kinds of hardware faults on up to 32 pins while the activation module ensures the proper initialisation of the target system. In a later publication (Arlat, Costes, et al., 1993), the authors evaluate the distributed fault tolerant architecture of the ESPRIT Delta-4 Project (Powell, 2012) using MESSALINE. By modelling the system as a Markov chain, where state transitions indicate failing or recovering nodes, they are able to evaluate the fault tolerance algorithms and mechanisms' ability to contain faults or safely extract faulty nodes.

Access to hardware is generally not possible, so hardware faults are often injected through software. And example of this is introducing bit-flips in various CPU registers using the XceptionNT tool (Costa and H. Madeira, 1999). This technique was used to assess the robustness of a common of-the-shelf (COTS) DBMS and was later extended by injecting software faults at runtime, targeting a number of assembly-level instructions (Costa, Rilho, and H. Madeira, 2000). In addition to dependability, the performance degradation introduced by the overhead of the recovery mechanisms was studied, as well as the cost of the recovery process. This research showed that software faults are more prone to cause hangs and aborts, especially when occurring in the benchmark client. While valuable, the faults are injected at a very low level making it more difficult to reason about the exact failure, like where a bug in the code originated for example. On the other hand, this approach does not require access to the source code of the system under test.

3.2.2 Software Faults

Software faults can be emulated by introducing mutations in systems at the machine-code level (Durães and H. Madeira, 2002). Using this technique, the authors defined generic faultloads which were injected in various web servers and a number of different applications in order to assess their dependability (Durães and H. Madeira, 2004, 2006). While no DBMSs were tested, the fault injection technique was generic and could in theory be applied to inject software faults in database systems as well.

A different approach is taken by the MapReduce Benchmarking Suite, which we covered in section 3.1.2 (Sangroya, Serrano, and Bouchenak, 2012a, 2016). This suite synthesises a new version of the MapReduce framework library, producing a synthetic library that has the same API as the original, but also provides some handles for activating or injecting faults. Using the synthesised library, MRBS is able to simulate programmer mistakes by throwing runtime exceptions originating from map and reduce tasks, and it can also provoke hanging tasks.

3.2.3 Operator Faults

As mentioned in section 2.3.1, one does not inject operator faults through any special means. Instead one introduces them precisely as they would normally be introduced, by accidentally performing the wrong operations. DBench-OLTP (Vieira and H. Madeira, 2003a) and MRBS (Sangroya, Serrano, and Bouchenak, 2012a), introduced in subsection 3.1.2, both support the injection of operator faults and show these can be used to simulate other types of faults as well. The precise operations a database administrator (DBA) must perform are quite dependent on the DBMS that is used, so it is important to be able to generalise operations and their consequences. Research shows that operations and faults can be classified and that this can be used to establish an equivalence between various DBMSs (Vieira and H. Madeira, 2002).

3.2.4 Virtualisation Techniques

Running containerised applications in virtualised systems is a practice that is becoming increasingly popular, but it is important to know whether there is a one-to-one correspondence between the virtualised and traditional bare metal systems when performing fault injection. Research comparing the effects of Software-Implemented Fault Injection (SWIFI) in bare and virtual machines details the types of failures that occur in both environments, as well as their distribution and causes (Le and Tamir, 2014). The authors also studied the effects of fault injection on system performance to see if both behaved similar and whether injection was not too intrusive. This research showed that the fault injection process is somewhat slower in virtualised systems, but that this is overshadowed by the significantly more rapid boot process, resulting in roughly five times faster injection runs in a VM.

3.2.5 Chaos Engineering

Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production.[2] Netflix developed a tool suite called the Simian Army to cause breakdowns in their production environment.[3] Their first tool, the Chaos Monkey, acts like a wild monkey in a datacenter randomly destroying machines, chewing through cables and causing other unexpected problems. While seemingly counter-intuitive – a production environment should be stable and testing should be done in a test environment – it pushed developers to write code that assumed faults would occur at some point in time, thus improving the resilience of their systems.

[2]Principles of Chaos Engineering website: https://principlesofchaos.org/

[3]Netflix's Simian Army blog post: http://techblog.netflix.com/2011/07/netflix-simian-army.html

Chapter 4

Scope and Design

This chapter covers the scope of this research in section 4.1, followed by the scenario design in section 4.2. We conclude this chapter by detailing the framework's design and its various components in section 4.3.

4.1 Scope

The goal of this research is to study how the performance of distributed database systems is affected by faults. This goal is quite broad, which is why it is important to define the scope of this project. We first discuss the framework and then cover the types of faults we wish to study. We then cover the system under test and lastly discuss the experiments.

4.1.1 Scenario Framework

To answer our main research question – how faults affect the performance of distributed database systems – we need to build a system capable of introducing faults and measuring their influence on performance. Many types of faults, database systems, deployment techniques and benchmarks exist, but it is impossible to support all of them. Instead of a complete system, we will design a framework with focus on reusability, reproducibility and extensibility.

The framework should provide a foundation to evaluate database systems by interacting with orchestration clients, database clients and accessing the machines themselves. The framework should be able to introduce basic faults this way while a benchmarking framework evaluates the performance of the system under test. Introducing complex faults that require specialised fault-injection software is not supported directly. Instead, one should include this software when deploying the system under test and extend the scenario framework to execute it.

4.1.2 Faults and Scenarios

The framework should be able to introduce faults in the system under test. As mentioned in subsection 2.3.1 and section 3.2, faults can occur on various levels and can be introduced through a number of techniques. The framework causes administrator faults by performing database and orchestration operations. Hardware and software faults are not caused by the framework, but can be simulated through the clients or by executing scripts or commands on the machines. Additionally, special software could be deployed on the machines after which the framework could invoke it.

In order to simulate more complex real-world scenarios, the framework should be able to target multiple parts of the system and introduce faults over time. A scenario is the set of faults introduced in a single test. Scenarios should be highly configurable and be capable of introducing different types of faults. Additionally, scenarios should be reproducible, meaning a scenario that is executed multiple times should introduce the exact same faults each and every time.

4.1.3 System under Test

The system under test is comprised of various (virtual) machines on which the DBMS is deployed. The scenario framework should communicate with the system under test using the database and orchestration clients. The DBMS should be deployed using an orchestration technique, and it should be evaluated through a benchmarking framework. The scenario framework should be able to handle any combination of databases, orchestration techniques and benchmarking frameworks, which is done through generic interfaces, and each variation requires its own implementation.

4.1.4 Experiments

An experiment should consist of a benchmark that evaluates the performance of the system under test while the scenario framework runs a scenario. The benchmarking framework should establish a baseline performance before the scenario occurs, and continue its benchmark while the fault is introduced to measure how performance is affected by both the fault and the DBMS' recovery process. Benchmarking frameworks often produce thorough reports which should then be compared to the fault injection timestamps to learn how the performance of the system under test was influenced.

The experiments should provide information to help find the system that is best suited for the task at hand. This means that it should be possible to run the same scenario on different DBMSs, run multiple scenarios on the same system or even run a single scenario multiple times on the same system but change the system's configuration every time.

4.2 Scenario Design

A scenario consists of phases or triggers which describe when and where faults are introduced in the system under test. This section first covers the general design of a scenario and then explains its various components as well as their relation to each other.

4.2.1 Scenario Overview

A scenario is a single experiment in which a set of faults are introduced in the system under test at a specified time and place. Faults determine which failures occur and where they should be injected, and triggers determine the conditions for a fault to be introduced. A scenario can be as simple as introducing a singular fault that affects only one node, but it can also be defined as a complex combination of multiple types of faults, introduced throughout various parts of the system over a period of time once specific conditions have been met.

Scenarios can be defined in two ways, by specifying either phases or triggers. Both will introduce the exact same faults in the system under test, but the former is a high-level description while the latter is much more granular. The rationale

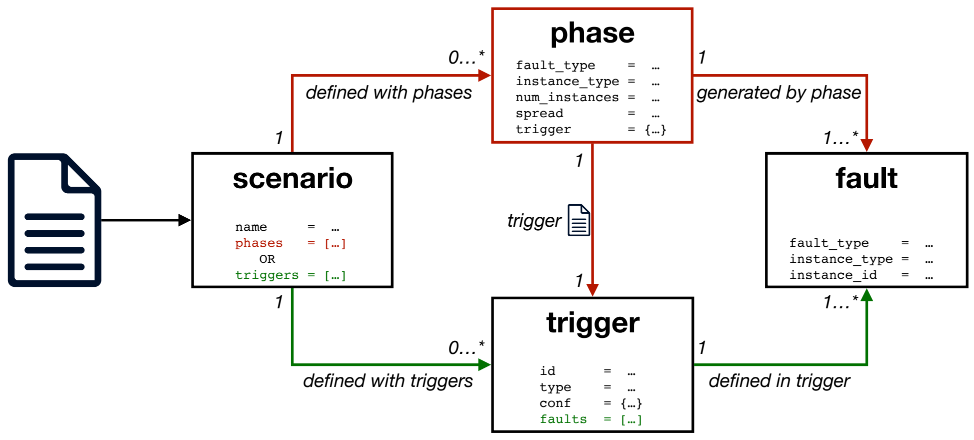


FIGURE 4.1: A schematic overview of a scenario and its components. A scenario is defined with either phases (in red) or triggers (in green). Phases generate faults on creation while triggers define them beforehand.

behind providing two options is that defining a phase-based scenario requires less effort, which makes it easier to quickly try out a scenario, but has the drawback that it is not guaranteed to target the exact same instances when reused. Trigger-based scenarios do provide this guarantee, but require more extensive configuration. In order to reduce the effort of configuring these granular scenarios, it is possible to generate trigger-based configurations from phase-based scenarios.

Details on phases, triggers, faults and instances are discussed below. A schematic overview of a scenario and its components can be found in Figure 4.1, and examples of configuration files are present in Appendix B.

4.2.2 Phases

A phase describes a part of a scenario at a high level. The configuration of a phase specifies the type of fault to introduce, the type of instance to affect and the trigger that determines when injection should occur. A phase does not target instances explicitly but is instead configured to select a number of instances spread over a number of clusters, which the framework does automatically when the scenario is initialised.

The automatic target selection makes it more straightforward to configure, but does not guarantee that the same instances are affected when executing a phase-based scenario multiple times. Furthermore, a single phase is limited to inject the specified fault in all selected instances at the same time, meaning multiple phases are required when properties should vary per injection. Lastly, one should be cautious when using multiple phases, as there is no control over the specific targeted instances so they are not guaranteed to be running if a different phase previously injected a fault there.

4.2.3 Triggers

A trigger describes when a collection of faults should be injected into the system under test. Triggers that are part of a phase specify a type and any required type-specific conditions which, once they are met, cause the phase's fault to be injected into the targeted instances. When using a trigger-based scenario instead, each defined trigger additionally specifies the collection of faults to be injected into the system.

When a fault is to be introduced in the system under test depends on the type of trigger and its configuration. The simplest trigger is time-based, injecting a fault after the specified time has elapsed. Triggers can also be configured to be dependent on another trigger, executing only once that trigger has succeeded, which is useful in the case a scenario should simulate a cascading disaster where the failure of one instance would cause another to fail as well. Triggers can also be used to flexibly schedule the injection of faults over time depending on certain conditions. For example, a scenario could be configured to only inject a new fault once the system has sufficiently stabilised or when an external service signals it to do so.

To make triggers more flexible and enable constructing more complex scenarios, the elements of various kinds of triggers can be combined to form a new type of trigger. For example, a trigger could be configured to introduce a fault five minutes after the system has stabilised from the injection of a previous fault.

4.2.4 Faults

A fault describes what kind of failure to introduce in the system under test. Faults are either generated from phases or defined as part of a trigger, depending on how the scenario was configured. The specified fault type is a high-level description, as the precise fault that is injected depends on a number of factors. For example, a node failure can be introduced in an instance through the database software, by accessing that (virtual) machine or by using the orchestration client. How each of these methods introduce their faults depends greatly on the system under test and how it is deployed, but they all cause a node failure in the end.

A single fault will always affect a single instance of the specified type. This means that targeting a node instance with a node failure will affect only the targeted node, however if a cluster instance is targeted by this fault then all nodes in this cluster will be affected.

4.2.5 Instances

Instances are the components that make up the system under test, most of which can be the target of a fault. Phases refer to a type of instance to affect while faults target specific instances. Commonly targeted instances are the (database) clusters and (database) nodes, but hardware components like disks and CPUs or more abstract concepts like the network can also be affected by a scenario.

As mentioned in the previous section, sometimes a type of fault can be introduced in multiple types of instances, in which case the effects depend on the hierarchy of the instances. A fault that corrupts a storage medium will affect all available storage when the target instance is a node, but when the specified instance is a hard disk then the fault will only be introduced there. The option to affect a single specific instance or target multiple instances further down the hierarchy enables the creation of both broad and granular scenarios.

4.3 Framework Design

The Scenario Framework consists of a main module responsible for creating and executing the scenario, and two modules related to communicating with the system

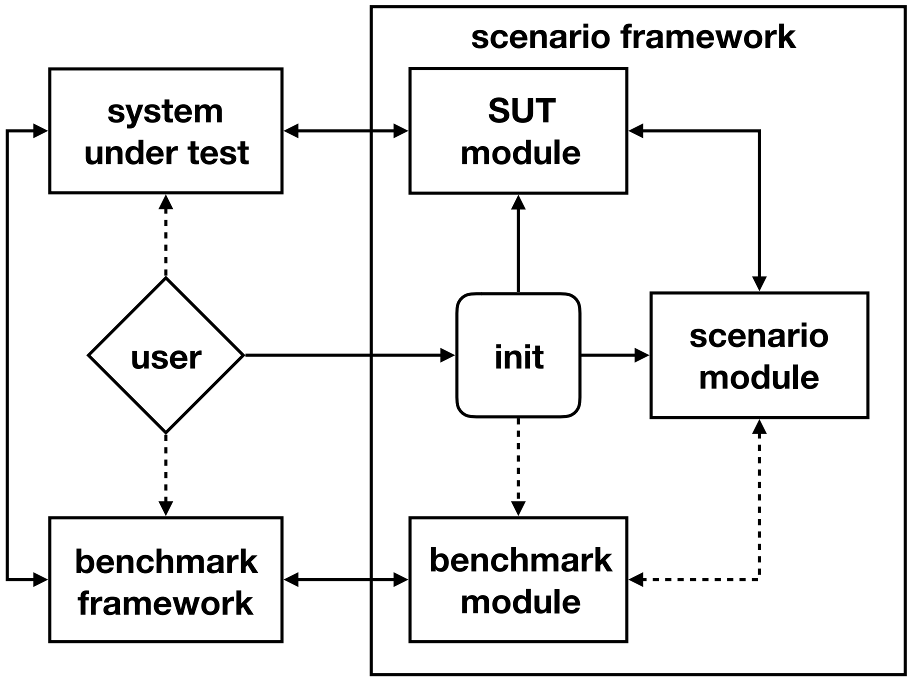


FIGURE 4.2: A schematic overview of the scenario framework, detailing how the modules are connected to each other and the external systems. Dashed lines indicate a connection is optional.

under test and benchmark framework. This section covers the design of the scenario framework and its modules, of which a schematic overview can be found in Figure 4.2.

4.3.1 Scenario Module

The Scenario Module is responsible for creating, scheduling and executing scenarios. It does this through its two main components, the scenario director and scenario builder. A schematic overview of this module can be found in Figure 4.3.

The scenario director instructs the scenario builder to construct the scenario and its components (as discussed in the previous section), based on the configuration supplied by the user and the system composition retrieved from the System Under Test Module. The scenario builder will check whether the scenario is defined as a phase-based or trigger-based scenario, and then instruct the components' respective builders to further assemble the scenario.

After creating a scenario, the scenario director will schedule its triggers. These will execute and introduce their faults once their conditions are met, e.g. a specific time has elapsed. To inject a fault, the scenario director passes it and any information on the intended target to the System Under Test Module, which will then perform the actual fault injection.

4.3.2 System Under Test Module

This module is responsible for communicating with the system under test, informing the scenario director of the system composition and introducing faults provided

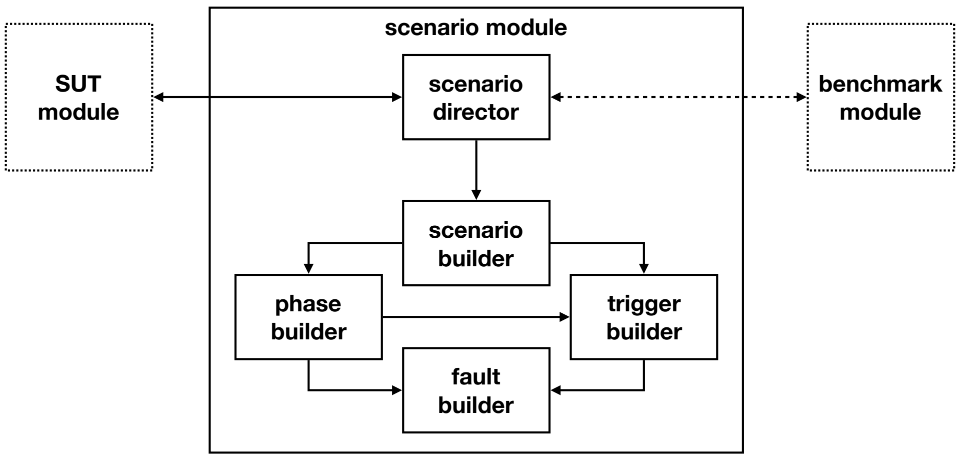


FIGURE 4.3: A schematic overview of the Scenario Module. The scenario director instructs the scenario builder and communicates with the other modules.

by the scenario director. Because many database systems and orchestration techniques exist, this module uses two APIs to provide generic methods for interacting with the system under test. Each distinct database system and orchestration technique requires an implementation of this API, enabling the scenario director to introduce faults the same way regardless of the actual system under test. A schematic overview of this module can be found in Figure 4.4.

Not all parts of this module are mandatory, only the functions related to discovering system composition and injecting faults at the deployment-, machine- and database-level are required. Handling the system's deployment, initialisation and shutdown, and monitoring the system at various levels is entirely optional. This is because while these functions allow coordinating and automating tests, setting them up for a test that will be run only once will take more time and effort than doing so manually.

Orchestration Client API

The Orchestration Client API provides interaction with the specific orchestration technique that is used to deploy the system under test. This enables the framework to introduce faults at the deployment-level by adding, removing or modifying resources. It also simplifies connecting to the individual machines in order to execute commands and simulate various faults at the machine-level.

This design requires an implementation of the API for each orchestration library. This API specifies the minimum required functions to be implemented, which generally are the logic for looking up instances and injecting faults. Naturally, the implementations may include additional features or helper functions.

Database Client API

The Database Client API enables the scenario framework to interact with the database software of the system under test in order to retrieve information and introduce faults. Similar to the Orchestration Client API, each DBMS requires its own implementation that provides the minimum required functionality.

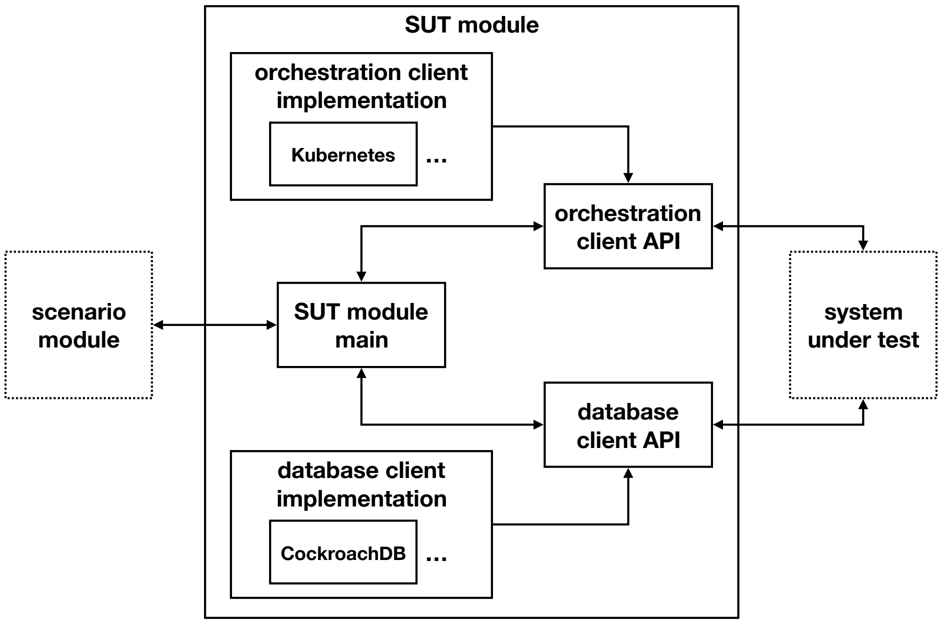


FIGURE 4.4: A schematic overview of the System Under Test module. Communication with the system under test is done through the client APIs, the specific implementations of which depend on the DBMS and orchestration technique.

Interacting with the database can be done in three ways. The database driver is the most straightforward, but it is also the most restrictive as administrative operations are often not permitted this way. The second option is to use the database's library, if available. Lastly, one can use the database client software, which supports (administrative) commands in addition to queries. Doing so locally might be difficult, as it requires that the machine running the scenario benchmark has this client installed. Additionally, if the system under test is deployed securely then it may be required to provide a number of certificates as well. It is generally easier to use the Orchestration Client API to connect to a node and execute the client commands there, as each node should already have access to the database client and required certificates.

4.3.3 Benchmark Module

The module responsible for communicating with the benchmark framework is optional, but can be used to reduce the effort required to run experiments. The purpose of this module is to enable the scenario framework to control the benchmark framework in order to automate and coordinate various processes, reducing the manual labour users would need to perform. The Benchmark Module allows the user to do any of the following: set up the benchmark framework, prepare a benchmark by creating the tables and generating or loading the data, and coordinate the execution of the benchmark and scenario.

Much like the other module, it is optional because the effort required for setting up this module would most likely be greater than the effort required for manually performing a single test. It is therefore intended to be used when a series of identical (or very similar) tests are to be performed. When included however, the Benchmark Module should provide an API and require an implementation for the specific benchmarking framework used for testing, similar to the System Under Test Module.

Chapter 5

Implementation

In this chapter we cover the implementation details, starting with the decisions regarding the proof-of-concept framework in section 5.1. We then discuss the scenario framework in general in section 5.2 and cover the Scenario Module and System Under Test Module in section 5.3 and section 5.4, respectively. We do not cover the Benchmark Module, described in subsection 4.3.3, as this optional module was omitted in the PoC implementation.

5.1 Proof-of-concept

The design of the framework supports any combination of faults, database systems, deployment techniques and benchmarking frameworks as long as an implementation is provided for each. The goal of this proof-of-concept version is to provide a basis for the framework and show that this approach is feasible, which is why it is limited to introducing high-level node failures in a CockroachDB cluster deployed through Kubernetes while running the TPC-C benchmark with OLTP-Bench. The key factors in each decision were the general reusability of the system, extensibility of framework features and reproducibility of experiments. We discuss each choice in detail below.

5.1.1 Scenarios and Faults

The proof-of-concept implementation lays the foundation for the scenario framework and allows us to demonstrate that our approach works. This means that introducing simple faults and providing the functionality for creating low-complexity scenarios suffices, as long as this basic implementation is easily extended to accommodate for more complex faults and scenarios in the future.

The PoC framework is capable of injecting a fault that simulates the high-level failure of a node, by introducing it at the orchestration-, database- or machine-level. In reality these failures can stem from hardware, software and operator faults, but our focus is not on the origin of these failures but their consequences. This scenario is relatively common but can nonetheless have serious repercussions. Node failure is also one of the aspects that is tested for often, and something distributed database systems should be able tolerate, making this a highly relevant scenario as well.

The proof-of-concept features only a single type of fault, but the framework is designed with the intention to include many more. Network issues and hardware component failures are a real-life occurrence as well, but these are both more difficult to implement and occur less frequently than generic node failures, making them less suited for the proof-of-concept implementation. Other scenarios, like modifying system composition to measure scalability and elasticity, or performing DDL changes and database maintenance, are highly realistic yet more niche, which is why they did not make the cut in the proof-of-concept framework.

5.1.2 Deployment

Setting up, running, breaking and subsequently restoring the system under test can be quite a cumbersome endeavour. To ease this process, we wish to make use of a system that, to some extent, can automate this. To make the framework more reusable, it is important to choose a technique that is both versatile and popular.

For our proof-of-concept implementation we have chosen to use Kubernetes, an open-source system for automating deployment, scaling, and management of containerized applications.[1] This orchestration technique makes it easy to deploy the system under test, communicate with it and introduce various faults. By running the system under test as a containerised application, we do not need to worry about any specific hardware or operating system, making our framework more reusable. Additionally, acquiring the instances on which to deploy the system under test can be automated as well, for example through the Google Kubernetes Engine[2] or the Amazon Elastic Kubernetes Service.[3]

An added benefit of using virtualised systems, and Kubernetes especially, is the ability to control many aspects of the deployed applications and their systems, making it easier to introduce faults and reproduce experiments. Furthermore, many database vendors offer detailed instructions or even fully automated solutions for deploying their systems through orchestration techniques.

While Kubernetes is not the only container orchestration technique, some alternatives being Docker Swarm[4] and Apache Mesos,[5] it is the most popular and widely supported technique. For these reasons, using Kubernetes strongly improves the reusability of our system, making it the prime candidate to integrate in our proof-of-concept implementation.

5.1.3 Database Management System

For the proof-of-concept implementation, we want to evaluate a representative distributed DBMS intended for OLTP. It is important that the system is compatible with a common version of SQL in order to provide wide support.

We have chosen to evaluate CockroachDB, a globally distributed NewSQL database that is wire compatible with PostgreSQL 9.5 and employs the Raft consensus protocol for both reaching consensus on operations as well as data replication (Taft et al., 2020). Additionally, CockroachDB is open source, is considered to be relatively mature and has great support for containerised deployment.

YugabyteDB,[6] FaunaDB (Freels, 2018), FoundationDB (Chrysafis et al., 2019), TiDB (Huang et al., 2020) and VoltDB (Stonebraker and Weisberg, 2013) were considered as well, but were not chosen for a number of reasons. These systems are either not sufficiently compatible with MySQL or PostgreSQL, lack certain relational capabilities, or are not available as an open source system, making them more difficult to evaluate.

[1]Kubernetes homepage: `https://kubernetes.io/`
[2]Google Kubernetes Engine website: `https://cloud.google.com/kubernetes-engine`
[3]Amazon Elastic Kubernetes Service website: `https://aws.amazon.com/eks/`
[4]Docker Swarm documentation: `https://docs.docker.com/engine/swarm/`
[5]Apache Mesos website: `http://mesos.apache.org/`
[6]YugabyteDB homepage: `https://www.yugabyte.com/`

5.1.4 Benchmarking Framework

In order to properly evaluate the system under test and conduct reproducible experiments, a reliable benchmarking framework must be used. Because most frameworks are standalone applications and our proof-of-concept implementation does not include the Benchmark Module, interoperability with the benchmarking framework has a low priority.

We have chosen to evaluate our CockroachDB deployment using the OLTP-Bench benchmarking framework (Difallah et al., 2013). It can execute the two industry standard benchmarks, TPC-C and YCSB (see subsection 3.1.1), but also includes 15 others for testing and stressing various aspects of the system. This framework allows the user to configure workloads that change over time, and can collect extensive statistics which aids in evaluating the impact of the injected faults. Furthermore, OLTP-Bench can be used for any JDBC-enabled database and is highly extensible.

We also considered two other frameworks, BigBench (Ghazal, Ivanov, et al., 2017; Ghazal, Rabl, et al., 2013) and BigDataBench (Wang et al., 2014), but these are intended for evaluating data analytics capabilities instead of transaction processing. These benchmarking frameworks are less suitable to use for our proof-of-concept implementation because the DBMS we chose to evaluate, CockroachDB, is intended for OLTP.

5.2 Framework Implementation

In this section we cover a number of aspects of our framework implementation in general. We first justify the programming language used to implement our proof-of-concept framework, then cover some details regarding the modular approach of our system and finally explain how we use configuration files throughout the framework. Additionally, an overview of the faults currently implemented by the framework as well as those that could be supported in the future can be found in Table 5.1.

5.2.1 Implementation Language

The proof-of-concept framework was implemented in Scala for a number of reasons.[7] Being a high-level language that runs on the JVM, Scala makes it easy to prototype new designs while it can still use libraries that were written in Java. Additionally, Scala supports multiple inheritance through traits and provides exhaustive matching, which is beneficial for creating new scenario components and basing the system's behaviour on them. Another reason to use Scala, besides the author's familiarity with the language and its concepts, is the distinction between values and variables, or mutable and immutable collections, which enables a clear separation between the components that should and those that should not be modified. Similarly, the ability to mark a class as sealed, so that it cannot be extended outside the file it was defined in, helps to provide more structure which is useful when creating a framework that is highly extensible and consists of multiple distinct modules.

5.2.2 Modules

As discussed in section 4.3, the Scenario Framework consists of three modules: the Scenario Module, System Under Test Module and Benchmark Module. Each of these has a singleton main class that is instantiated through a companion object. These

[7]Scala programming language homepage: `https://www.scala-lang.org/`

TABLE 5.1: An overview of the implemented faults and those that the framework could support in the future. Please note that the latter is not exhaustive and serves mostly to indicate possibilities for future applications.

Fault	Level	Requires	Implemented
High-Level Failure	Node	CLI & command	5.4.2
	Database	Database CLI & command	5.4.3
		Database Library	–
	Orchestration	Orchestration Library	5.4.2
Hardware Failure	CPU	Injection Software	–
	Memory	Injection Software	–
	Storage	Injection Software	–
Software Failure	Node or Database	Injection Software	–
		Modified Software	–
Administrator Failure	Database	Database CLI & command	–
		Database Library	–
	Orchestration	Orchestration Library	–
Network Failure	Node	Network Proxy	–
	Orchestration	Orchestration Library	–

classes are singletons because only a single scenario should be created, and only one system under test and benchmark framework should be used in an experiment. Similarly, the System Under Test Module provides an API for interacting with the database and orchestration clients, and only one implementation should be instantiated for each. To ensure this, each companion object may instantiate its class only once, after which subsequent calls will return the existing instance instead. Normally one could use static objects for this, but we haven chosen this approach because we must dynamically instantiate and configure different classes based on the provided configuration.

5.2.3 Configuration

We use a number of configuration files in order to dynamically create different scenarios, configure modules and instantiate the correct client APIs. The scenario framework uses the Typesafe Config library to supply and validate files in the HOCON (Human-Optimized Config Object Notation) format because they are easy to understand, in contrast to the machine-optimised JSON format.[8] Additionally, the HOCON format supports substitutions to refer to other parts of the configuration file which lets us avoid copy-pasting properties. The library is also capable of merging and validating multiple configuration files at runtime, which allows us to store the configuration for different modules and components separately, thereby reducing the chance of accidentally modifying the wrong properties. Validating configuration files beforehand also gives the framework the option of failing early while providing an explanation as to which configuration fields are missing. An example of the database client configuration can be found in Listing 5.1 and examples for scenario configuration files can be found in Appendix B.

[8]Lightbend Config library GitHub page: `https://github.com/lightbend/config`

5.3 Scenario Module

The Scenario Module provides all functionality related to creating, scheduling and executing scenarios. The scenario director instructs the scenario builder to create the scenario and its components based on the provided configuration, after which the scenario is scheduled and executed. An overview of the classes in this module and the relations between them can be found in Appendix A. We first cover how the scenario builder constructs scenarios, and then explain how the scenario director schedules and executes them.

5.3.1 Scenario Builder

The scenario builder is responsible for creating a scenario based on the configuration provided by the scenario director. As discussed in subsection 4.1.2, the scenario's components are defined through either phases or triggers, which are created by their respective builders. The challenge is in dynamically instantiating the components and their various subtypes, while providing an extensible basis for future components.

Each component's builder object implements the builder pattern to separate the construction of its instances from their representation. This allows us to construct the components step by step, and only build them once all the parts that are required for these particular instances have been added. Additionally, we can validate the components before building them to assure that the scenario's configuration meets certain requirements or force an early failure if it does not.

All components except phases have subtypes, each of which inherits properties from their respective abstract component. For example, an Instance type can be either a Node or Cluster. This inheritance allows us to easily define new types and match them to ensure faults are introduced in the right place, or even fail at compile time if the match was not exhaustive. Scala also supports multiple inheritance by mixing in multiple traits, which we use to flexibly define more complex types. For example, a new Trigger type could inherit both the Timed and Dependent trait in order to produce triggers that introduce faults a specific amount of time after the trigger on which it depends has executed. An example of a more complex scenario can be found in Listing B.3.

5.3.2 Scenario Director

The scenario director is responsible for scheduling the scenario created by the scenario builder. A scenario's triggers determine when faults should be injected in the system under test: once another trigger has succeeded, after some time has elapsed or both. To make scheduling triggers easier, the scenario director uses the Monix library.[9] This library provides the `Task` data type, which allows us to lazily define operations and execute them asynchronously, after which we chain them together to form a single task that represents the entire scenario.

To execute the scenario, we must schedule all of its triggers. Each trigger holds one or more faults, which should be introduced in the system under test once their trigger's requirements are met. To schedule a trigger, we need to create a task that contains all the operations to inject its faults. A trigger's faults can be injected in parallel, so we create a task for each and combine them into a single task that is

[9]Monix library website: https://monix.io/

considered to be successful if each individual task was a success. If the trigger implements the `Timed` trait, we also apply the appropriate delay. Pseudocode for these operations can be found in algorithm 1. Note that each task is defined lazily and execution only happens once explicitly instructed.

Algorithm 1: Create a Trigger Task

```
Input: trigger, the trigger for which we want to make a task
Result: A task representing the input trigger, injecting its faults in parallel
faultTasks ← ∅
for fault ∈ trigger do
    faultTask ← Task { inject fault }
    faultTasks ← faultTasks ∪ { faultTask }
end
parallelTask ← Task { run faultTasks }
triggerTask ← Task { check result of parallelTask } with delay
return triggerTask
```

Now that each trigger is represented as a task, we can chain them together to create a single task that represents the scenario. Because a task can't be modified once created and any function applied to it returns a new task, we must use a recursive depth-first approach. At the deepest level, we combine the tasks of the triggers with the same dependencies and repeat this for each level until we reach the triggers that can execute independently. We now have a single task per independent trigger, which we combine into a single parallel task that represents the scenario. Pseudocode for these operations can be found in algorithm 2.

In order to execute the scenario, we run the scenario task and await its result. Meanwhile, the individual triggers execute and instruct the System Under Test Module to inject their faults. The result of the scenario will then become available once the last trigger has executed, and is considered to be successful if all triggers successfully introduced their faults in the system under test.

5.4 System Under Test Module

The System Under Test Module is responsible for communicating with the system under test and introducing the faults it receives from the scenario director. It does so through two APIs that provide interaction with the database and orchestration clients.

In addition to fault injection, this module also creates an abstract representation of the system under test based on the information received from both clients. The proof-of-concept implementation supports Cluster and Node instances, but makes a distinction between Client Nodes and Application Nodes. The difference is that the former are special nodes that are not part of the database clusters but only facilitate access to the database client software, effectively acting as a gateway. They can therefore not be targeted by faults, but can instead be used to inject them in the Application Nodes, as discussed in section 4.3.2.

5.4.1 Fault Injection

Faults can be injected at the orchestration-, database- or machine level, which is handled by either the orchestration or database client. If the fault type indicates it

Algorithm 2: Create the Scenario Task

```
Data:
   independentTriggers, a list of triggers that can execute independently
   dependencyMap, a map from a trigger to those that depend on it
   taskMap, a map from a trigger to its task
Result: A task representing a Scenario, scheduling and executing its triggers
chainedTasks $\leftarrow \emptyset$
for trigger $\in$ independentTriggers do
    chainedTask $\leftarrow$ chain(trigger, dependencyMap, taskMap)
    chainedTasks $\leftarrow$ chainedTasks $\cup$ { chainedTask }
end
parallelTask $\leftarrow$ Task { run chainedTasks }
scenarioTask $\leftarrow$ Task { check result of parallelTask }
return scenarioTask
Function chain(trigger, dependencyMap, taskMap)
    triggerTask $\leftarrow$ taskMap[trigger]
    if trigger $\in$ dependencyMap then
        dependencyTasks $\leftarrow \emptyset$
        dependencies $\leftarrow$ dependencyMap[trigger]
        for dependency $\in$ dependencies do
            dependencyTask $\leftarrow$ chain(dependency, dependencyMap, taskMap)
            dependencyTasks $\leftarrow$ dependencyTasks $\cup$ { dependencyTask }
        end
        parallelTask $\leftarrow$ Task { run dependencyTasks }
        chainedTask $\leftarrow$ Task { check result of parallelTask }
        return chainedTask
    else
        return triggerTask
    end
end
```

LISTING 5.1: An example of a configuration file used for the database client API implementation for CockroachDB

```
type = "CockroachDB"

client_config = {
  certs_dir_node = "/cockroach/cockroach-certs/"
  certs_dir_client = "/cockroach-certs/"
  host_public = "cockroachdb-public"
}

command_config = {
  database_command = "cockroach"

  general_flags = {
    certificate_dir = "--certs-dir"
    host = "--host"
  }

  commands = {
    decommission_node = {
      command = ${command_config.database_command} "node decommission"
      flags.wait.flag = "--wait"
      flags.wait.value = "none"
    }
    quit_node = {
      command = ${command_config.database_command} "quit"
      flags.drain_wait.flag = "--drain-wait"
      flags.drain_wait.value = "0s"
    }
    node_status = {
      command = ${command_config.database_command} "node status"
      flags.format.flag = "--format"
      flags.format.value = "tsv"
    }
  }
}
```

is an orchestration- or machine-level fault, then the orchestration client handles it, otherwise the database client takes over.

There are two ways to inject faults, by using either the client library or executing commands directly on the machine. Client libraries provide an interface which allows one to programmatically perform operations on the underlying system, however, such a library is not available for all systems. For example, many database vendors only provide a command-line interface for their systems. Commands can be used if this is the case, or when the library does not support the required operations. A schematic overview of the fault injection process can be found in Figure 5.1.

The fault commands are created based on the provided configuration and are executed on a specific node. Commands are defined as a `command` key-value pair and a `flags` sub-object which specifies the flags and values that can be set for this command. Additionally, the configuration can contain a number of general flags. This is useful when a flag is present in multiple commands but a different value is used each time, which could be the case for a set of commands that must specify the hostname of the target node. An example of a part of the configuration for CockroachDB can be found in Listing 5.1.

Our proof-of-concept implementation evaluates CockroachDB, which does not feature a client library. Instead, we must use the command-line interface to perform any administrative database operations. This means that our database client API implementation only creates the commands required to introduce faults, and that the Orchestration Client is responsible for actually executing them.

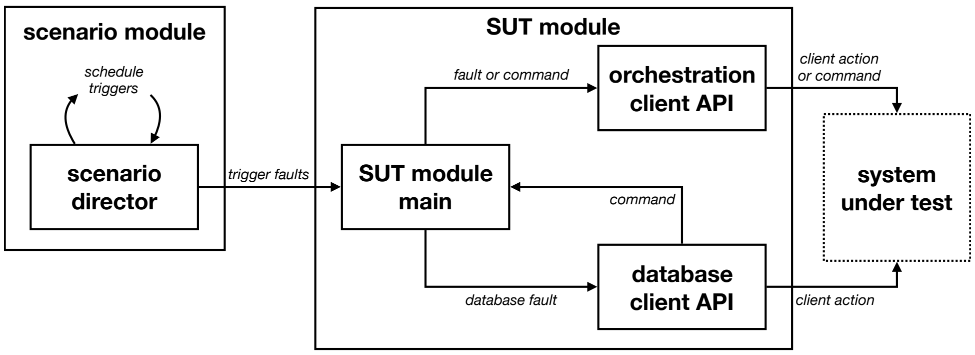


FIGURE 5.1: A schematic overview of the execution of a scenario. The scenario director schedules the scenario's triggers and the System Under Test Module introduces them through the appropriate API once triggered.

5.4.2 Orchestration Client API

The Orchestration Client API specifies the minimum functionality that is required for communicating with the system under test. Each orchestration technique requires an implementation of this API which must be able to retrieve the various components of the system under test, inject faults and execute commands. Our proof-of-concept framework provides an implementation for the official Kubernetes Java Client and can simulate the high-level failure of a node at the orchestration- and machine-level.[10]

Kubernetes Client Implementation

Our implementation for the official Kubernetes Java Client library focuses on three aspects: retrieving resources and representing them as the instances we use throughout our framework so they can be included in the abstract representation of the system under test, executing commands on machines in our system under test, and lastly, injecting the Client Node Failure and Node Process Failure faults.

Retrieving Resources Our framework represents the system under test in terms of clusters, client nodes and application nodes, but Kubernetes does this differently. An important task for our client implementation is retrieving the Kubernetes resources and representing them as the instances we use, effectively providing a mapping between the two.

Kubernetes uses namespaces to refer to clusters. These can be virtual, meaning multiple namespaces may be available on a single physical cluster. A namespace may contain any number of nodes, which are the (virtual) machines that run pods. A pod is a group of tightly coupled containers that share resources, modelling an application instance. Kubernetes resources can be labelled to provide relevant metadata or identify them more easily.

The framework's clusters are created by retrieving the namespaces by name, which are defined in the configuration file beforehand. We create the framework's nodes by retrieving all pods in a namespace and use their labels to tell apart the client and application nodes.

[10]Official Kubernetes Java Client library GitHub page: https://github.com/kubernetes-client/java

Executing Commands We use Kubernetes' Exec API to execute commands on the pods represented by our framework's nodes. This opens a websocket to the target pod through which we can execute any command and read its response. The precise nature of the commands is not relevant here; we simply return the raw response to the calling function which is then responsible for interpreting whether this response indicates the command was executed successfully.

Introducing Client Node Failures We can use the Kubernetes client library to simulate the high level failure of an application node. We do so by instructing it to delete the Kubernetes pod represented by the targeted application node. This fault can also target an entire cluster, in which case we introduce it in each of its application nodes. Each pod will then attempt to perform a graceful shutdown within its configured grace-period, but we can override this in order to make the shutdown more abrupt. The result of this fault injection depends on the HTTP statuscode of the request made by the library, which we consider to be successful if it does not indicate that a client or server error occurred.

Introducing Node Process Failures We can simulate a fatal failure in the main process of an application node by executing a command that kills that process. This command uses the `kill` system call to send a signal to a process with a specific `PID`, but it can be made as complex as necessary for the specific system under test by modifying the configuration file. Our current implementation targets the correct process by reading the `PID` from a file that is created automatically when the container starts, and then kills it by sending the `KILL` signal which will instantly terminate the targeted process. This signal cannot be caught or ignored, but it is important to know that this does not affect the init process with `PID` 1 meaning one should use the `TERM` signal instead. We consider this fault to be injected successfully when the command executes correctly.

5.4.3 Database Client API

The Database Client API specifies the minimum functionality that is required for communicating with the deployed database management system. Each DBMS requires an implementation of this API which must be able to retrieve information on the system and inject faults, or provide the commands that the Orchestration Client API can execute to achieve this. Our proof-of-concept framework provides an implementation for CockroachDB and can simulate the high-level failure of its nodes.

CockroachDB Client Implementation

Our implementation for CockroachDB focuses on two aspects: retrieving information about the state of the system and its nodes, and injecting the Database Node Failure fault. As mentioned in subsection 5.4.1, CockroachDB does not feature a client library so we create commands for the command-line interface instead of performing these operations directly.

Retrieving System Status We use the `node status` command to retrieve the status, ids and full host names of all application nodes in the system. We need this information because most of the commands use the client node as a gateway and require the ids or host names to indicate which application nodes should actually be targeted by a command.

Introducing Database Node Failure We use the `cockroach quit` command to simulate the high level failure of an application node. According to the CockroachDB documentation,[11] nodes will try to finish in-flight requests and gossip their draining state to the rest of its cluster. This is best effort, meaning these operations will time out after the maximum time specified in the cluster settings. We can override this by setting the `drain_wait` flag, which allows us to shut down nodes more abruptly. We consider this fault to be injected successfully when the command executes correctly.

[11]CockroachDB documentation for `node quit` command: `https://www.cockroachlabs.com/docs/v20.2/cockroach-quit.html`

Chapter 6

Experiments

The goal of this thesis is to show that we can use the scenario framework to inject faults and measure their influence on performance. To test this, we use our framework implementation to inject faults into the system under test while we use the benchmarking framework to evaluate its performance. In this chapter, we first cover the experimental setup and then discuss our experiments and their results.

6.1 Experimental Setup

Each experiment uses the same hardware, deployment strategy and benchmark configuration. The only difference is the level at which the fault is injected. While a number of aspects in the experimental setup are not realistic for evaluating a state-of-the-art NewSQL database system – these are often deployed on incredibly powerful machines with finely tuned configurations – it is important to remember that the goal of this thesis is to provide a basis for an extensible framework and show that this is a feasible approach for introducing faults and measuring their impact.

We first detail the hardware and deployment strategy used in our experiments, and then cover the setup and configuration of the benchmark framework and database management system.

6.1.1 Hardware

We run our CockroachDB nodes in the Google Kubernetes Environment (GKE) using Google's n1-highmem-4 machines, which have 4 vCPUs, 26GB RAM and a 50GB hard disk. Each machine runs a single CockroachDB Pod that is provisioned with 3 vCPUs, 20GB RAM and its own persistent SSD storage with a capacity of 100GB. One machine also runs the Pods required for the Kubernetes system, as well as a special client Pod for interacting with the database system. The machines together form a single cluster on which we deploy CockroachDB. The scenario and benchmarking framework are not containerised but have a low resource footprint, so we run these locally on a 2017 MacBook Pro.

6.1.2 Deployment

We deploy CockroachDB version 20.2.0 as a StatefulSet by following Cockroach's guide.[1] A StatefulSet is similar to a regular Kubernetes Deployment, managing Pods with identical container specs, but differs in that it creates Pods that can be identified by name. This identifier remains the same, no matter how many times a specific Pod

[1]Guide for setting up a CockroachDB cluster as a Kubernetes Statefulset: `https://www.cockroachlabs.com/docs/v20.2/orchestrate-a-local-cluster-with-kubernetes.html#manual`

is rescheduled. This makes it possible to provide storage for each Pod separately which ensures that a replacement Pod will always be matched with the correct storage volume.

In the context of running experiments, deploying the System Under Test as a StatefulSet has both advantages and disadvantages. The greatest advantage is that the System's n nodes are always represented as Pods with names ending in a number from 0 to $n-1$, so targeting Pod 0 will always affect the same instance. Something that is both a benefit and a drawback is provisioned storage. It ensures that Pods can be restarted or rescheduled much faster, but also means that redistributing data during recovery is hardly necessary which reduces the impact during an experiment. A significant downside is the StatefulSet's Pod restart policy, which can't be modified and is set to always restart or reschedule Pods. This means that deleting a Pod, killing the Cockroach process or otherwise shutting down a Pod will cause little downtime.

A number of minor modifications were made to the configuration before deploying the system under test. The container's hardware requests were modified to match the n1-highmem-4 machines. Additionally, Pods were modified to share their process namespace and each Container was instructed to write its Cockroach PID to a file. Lastly, the Service meant to be used by clients of the database to load balance connections was changed to expose itself outside the cluster as well, allowing us to connect the benchmark framework more easily.

6.1.3 Benchmark Framework Configuration

The benchmark framework used in the experiments is a modified version of the OLTP-Bench project.[2] Multiple modifications were made by a Cockroach employee in order to clean-up and modernise the codebase, as well as support CockroachDB out of the box.[3]

Benchmarks test the performance of a system and should naturally abort when errors occur because the results would not reflect the optimal conditions regardless. However, the scenarios we execute purposely cause such failures so the benchmark should continue. To ensure this, minor modifications were made to the benchmark framework to tolerate these situations. Any `SQLException` thrown by OLTP-Bench's workers could be the result of a fault we introduced, so we check the exception's `SQLState` to determine whether the benchmark should continue. Return codes starting with `0800` indicate a connection exception and those starting with `57P0` indicate operator intervention, both of which should be tolerated when evaluating CockroachDB. If neither match then we still throw a `RunTimeException`.

The benchmark that is performed in the experiments is the TPC-C benchmark, which was covered in section 3.1.1. All transactions are run at the strictest serialisation level, being `TRANSACTION_SERIALIZABLE`. TPC-C scales the database by increasing the number of warehouses, which we set to 10, producing a dataset of about 2 GiB. We run each benchmark with 10 concurrent terminals that submit a total of 10 transactions per second. These values were determined experimentally by setting the number of warehouses and varying the number of terminals and transactions until a sustainable level was reached. The 10 transactions per second result in 200–250 queries per second, with about 55 % selects, 33 % updates, 10 % inserts and 2 %

[2]OLTPBench project GitHub page: https://github.com/oltpbenchmark/oltpbench

[3]OLTPBench project CockroachDB fork GitHub page: https://github.com/timveil-cockroach/oltpbench

deletes. Because of the low number of requests per second, we also reduced the batch size to 16.

The sustainable throughput of our system under test is relatively low, especially considering its hardware. While we do not know the exact causes or the extend of their impact, we can discuss a number of factors that could have had a negative influence. First and foremost, we deployed our system through Kubernetes, which is known to introduce some overhead, and did not make any specific modifications for performance benchmarking. Secondly, running the benchmarking framework locally is very likely to have had a negative impact on the latency. Lastly, TPC-C restricts the maximum throughput based on the number of warehouses used in order to force more powerful systems to also handle an increase in data. Increasing the number warehouses would also increase the duration of each experiment because we must restore the system to its original state. This process becomes increasingly more lengthy as the amount of data increases, which is why we chose to use only 10 warehouses.

The TPC-C benchmark configuration and benchmark parameters can be found in Appendix C, in Listing C.1 and Listing C.2, respectively.

6.1.4 System Under Test Configuration

We run a CockroachDB cluster of 3 nodes, which, due to the Raft consensus protocol, is the minimum required number of nodes. In addition to this, we deploy a client node which functions as a gateway for executing a number of client commands. Data is replicated threefold, so each node holds a complete copy of the database. This system allows us to introduce our three faults: we can kill the Cockroach process, quit the database node and delete nodes through the Kubernetes client.

We also modified two cluster settings to handle some issues we ran into during testing. `kv.snapshot_rebalance.max_rate` and `kv.snapshot_recovery.max_rate` set the rate limit to use for a few types of snapshots, which can have an impact on the variance of the latency. We reduce both from $8.0\,\mathrm{MiB\,s^{-1}}$ to $1.0\,\mathrm{MiB\,s^{-1}}$ in order to smooth out the snapshot propagation and increase the latency stability during our experiments.

6.2 Experiments

The proof-of-concept implementation of the scenario framework supports three basic faults, each simulating the high level failure of a node. We perform three different experiments, one for each type of fault we can introduce. We first cover the testing procedure and then discuss the experiments and their results.

6.2.1 Testing Procedure

To ensure each experiment runs under the same conditions, we generate the data once and then create a backup. At the start of each run, we restore the database from the backup and then wait 5 minutes before starting the benchmark and scenario framework. We include this wait time to allow the system to come to rest again, because the restore operation requires a lot of resources and introduces a high latency for its entire duration.

During each run we perform a 5 minute warm-up and a 10 minute benchmark, and we inject a fault after 5 minutes of benchmarking. We use the first half a benchmark to establish a baseline performance and the remainder to study the impact of the injected fault and the system's recovery process.

While faults can affect the consistency of the data, we do not validate the correctness for three reasons. First, we run our benchmarks at the `SERIALIZABLE` isolation level, meaning the database system will detect these violations itself and ask the client to retry. Secondly, CockroachDB performs internal consistency checks and uses the Raft protocol for achieving consensus and replicating data, further preventing these violations. Lastly, validation is benchmark-dependent and should be performed by the benchmarking framework, which OLTPBench does not support for TPC-C.

6.2.2 Analysing Results

We perform each experiment 6 times, cycling the nodes targeted by the fault. We do this to prevent killing the same node over and over again, as this had some negative effects during testing. We bin the raw results of each individual benchmark run into non-overlapping 5-second windows and plot the 95th percentile latency for each. We then combine the results of each run and plot the median value for the 50th, 95th and 99th percentile latency over all runs. We do this to filter out any anomalies and provide more accurate results.

The main metric for performance in TPC-C is the number of NewOrder transactions that can be performed per minute. Additionally, it is required that the latency remains below a given threshold. As we have experimentally established a sustainable workload, which we keep constant throughout the entire benchmark, we only investigate the latency of the NewOrder transactions.

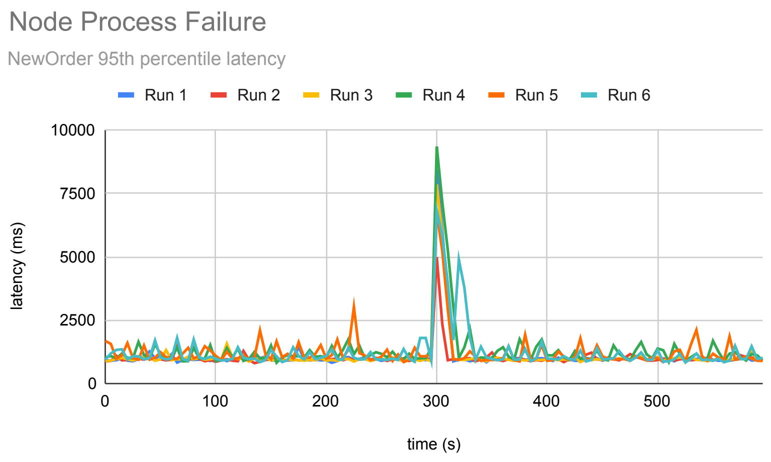
6.2.3 Node Process Failure

In this experiment we simulate the failure of a node by killing its main database process. This fault can be classified as a software fault, but could also be seen as a hardware or operator fault. We use the Kubernetes client to connect to a node and execute a command that kills the main process.

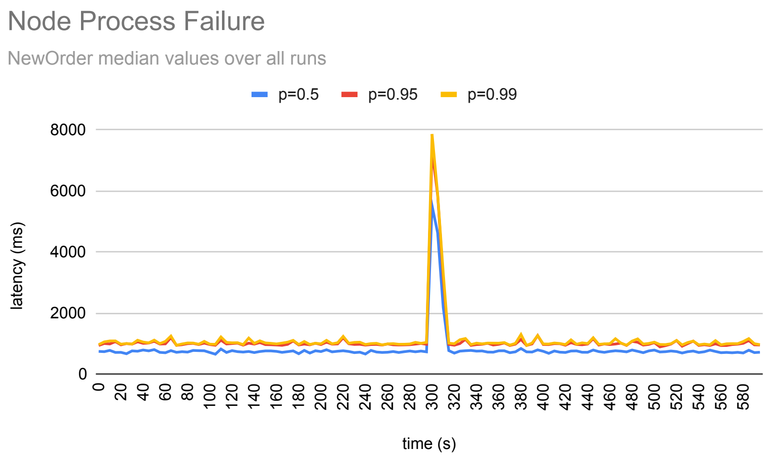
The benchmark results of this experiment can be found in Figure 6.1. We can see a latency peak at around 300 seconds in every run, which can only be the result of the fault we introduce. Additionally, the 6th run shows a second peak 20 seconds later and the 5th run has numerous smaller peaks as well.

The benchmark logs confirm that a number of connections were broken after 300 seconds, which is when we introduced the fault. The logs indicate I/O errors occurred as the result of a broken pipe, and that the connection pool is unable to add a new connection until the load balancer has realised the affected node is not available, causing the client to abort a number of transactions. Additionally, the database aborts a number of transactions due to invalid leases and write conflicts. Lastly, the crashing of the node causes some ambiguous results, meaning the system is unable to tell if a transaction was successfully committed or not.

No other errors or irregularities could be found in either the benchmark or database logs, so no explanation can be given for the second peak in the 6th run or the smaller peaks in the 5th run. We expect that the former is caused by the delay in acquiring new connections and the latter by random network jitter.



(A) 95th percentile latencies for NewOrder transactions for all runs



(B) Median values for 50th, 95th and 99th latencies for NewOrder transactions over all runs

FIGURE 6.1: Results of Node Process fault injection

6.2.4 Database Node Failure

In this experiment we simulate the failure of a node by shutting it down through the database client. This fault is best classified as an operator fault, as we use the client software in the exact same way as an operator would normally do when terminating a node. We set the `drain_wait` flag to 1 second to quit the node as abruptly as possible and thus disrupt the system as much as possible.

The benchmark results of this experiment can be found in Figure 6.2. We can see a latency peak at around 300 seconds in every run except run number 6, and a number of minor peaks between 7 and 8 minutes in run 1 and 3 for all transaction types.

The peak at 5 minutes can only be caused by the fault we introduced, which is confirmed by the benchmark logs. We find multiple connection errors and 'Operator Intervention' notices at around 300 seconds, as well as a number of leaseholder and transaction retry errors. Because the period for performing a graceful shutdown is only 1 second, the node will initiate the transfer of its leases but is unable to finish in time, causing these errors. Transactions that were in process are then either aborted or delayed until the node is back and consensus can be reached again. The retry errors are then caused because a number of write intents did not propagate, preventing the asynchronous writes that were issued before.

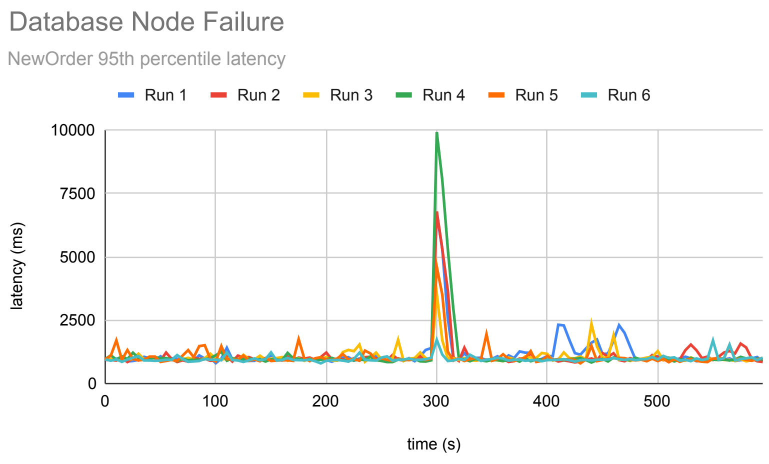
The exact reason why the peak is absent in run 6 is difficult to determine because there are many factors that can influence this. It is possible that the node had no active connections or was not involved in any distributed operations, in which case the impact is reduced. Similarly, the impact could vary based on the type of transactions that were being executed on this node. A transaction performing a large number of distributed writes is more heavily impacted than one that involves local reads. Additionally, if this node was not performing Raft operations and was not the leaseholder of the ranges required by the other nodes then the system could be more capable of operating without it.

The logs indicate that connection errors did in fact occur, but we believe that all transactions involving this node were aborted by the benchmark client and retried on a different node instead of being rejected by the database because the server was able to notify the benchmark client in time. To test this theory, we perform 9 additional runs where we increase the `drain_wait` flag to 1 minute to give the system more time to perform a graceful shutdown. We now expect that the nodes are always able to notify both each other and the benchmark client, so leases should be transferred and traffic rerouted to avoid the affected node. While the node is not able to transfer all its leases in time and latencies seem to fluctuate, we find no peaks that appear to be caused by nodes shutting down, supporting our theory.

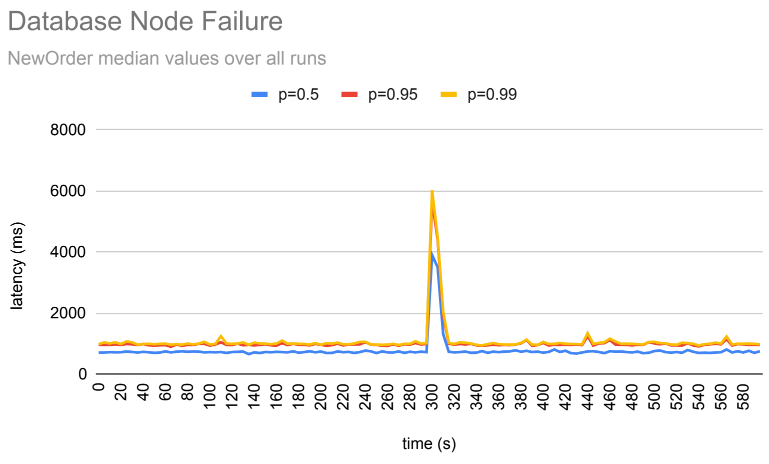
To further confirm the theory that nodes must communicate a shutdown to prevent a latency peak, we want to repeat the first experiment where a node is shut down instantly by killing its main process. We now expect to see a peak again because the nodes cannot communicate that they are performing a shutdown. We repeat the Node Process Fault experiment 9 additional times and observe a peak when the fault is injected in each and every run. Including the previous experiment, we see a peak in all of the 15 runs, providing strong support for this theory.

6.2.5 Client Node Failure

In this experiment we simulate the failure of a node by deleting its pod through the orchestration client. This fault is best classified as an operator fault, as we use



(A) 95th percentile latencies for NewOrder transactions for all runs



(B) Median values for 50th, 95th and 99th latencies for NewOrder transactions over all runs

FIGURE 6.2: Results of Database Node fault injection

the client software in the exact same way as an operator would normally do when deleting a pod. It could however also be seen as the result of some other error, forcing the deletion and rescheduling of the pod by Kubernetes. When performing this operation, we set the grace period for the deletion of this pod to 1 second to delete it as abruptly as possible and thus disrupt the system as much as possible.

The benchmark results of this experiment can be found in Figure 6.3. We can see a latency peak at 300 seconds in runs 2 and 5 while all the others remain relatively stable. Additionally, we see some irregularities around 400 seconds during run 3 and 5.

Even though fault injection only leads to latency peaks in run 2 and 5, benchmark logs indicate that all runs featured multiple broken connections and aborted transactions. Upon inspection of the database logs, we find that the nodes initiate a graceful shutdown upon receiving the termination signal from Kubernetes and try to transfer their leases away. In the 2nd and 5th run this transfer was not started in time causing unresponsive heartbeats and lease issues.

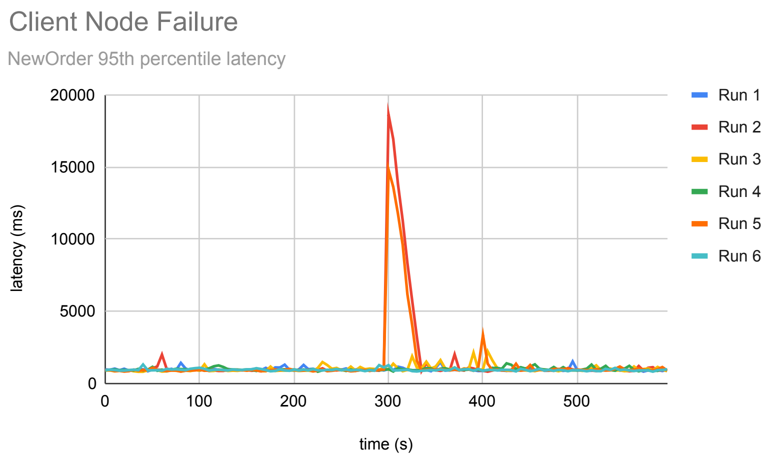
Similar to the previous experiment, we believe that the nodes being able to communicate their shutdown prevents the peaks. To confirm that this was due to the 1-second grace period, we can instruct the Kubernetes client to forcefully terminate a Pod. The reason we first tried non-forceful termination is because the Kubernetes documentation states that this may lead to severe problems and data inconsistency due to how StatefulSets operate. This method might reschedule a pod before the previous one has truly been removed and could cause two nodes to access the same provisioned storage which could have unpredictable consequences for both the system and its data.

We now expect that the nodes are unable to communicate that they are shutting down which should lead to a latency peak in every run, similar to the first experiment. We perform the experiment 3 times and inspect the benchmark results and the logs from the benchmark framework and CockroachDB. To our surprise we find no peaks even though the benchmark logs show connections were broken and transactions were aborted due to new leases. Upon inspection of the database logs, we again find that the nodes initiate a graceful shutdown. Interestingly, the nodes are not shut down instantly as was expected but instead receive the `TERM` signal twice. The database logs show the nodes acknowledge this and continue their graceful shutdown, with their last entry being roughly 1.5 seconds after receiving the signal, meaning that the nodes were actually active for longer than in the initial experiment.

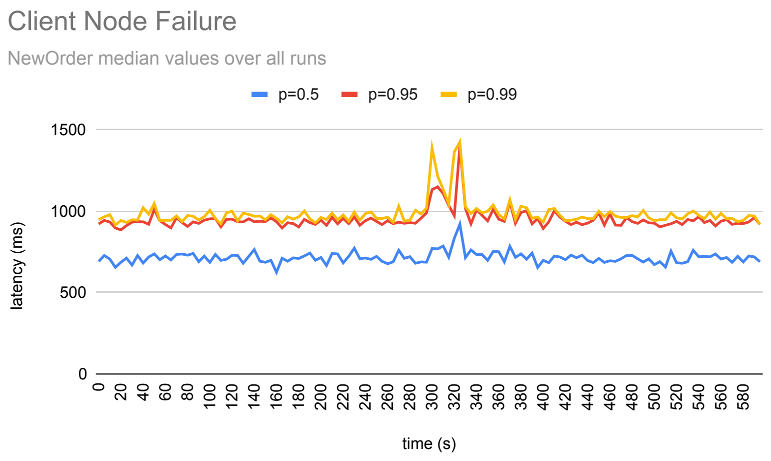
To rule out that this is not due to our implementation or the Kubernetes Java client, we repeat the experiment 3 more times using kubectl, the Kubernetes command-line tool. The first 2 runs again do not produce a latency peak, although the time between receiving the two `TERM` signals and the last log entry has now been reduced to roughly 0.8 seconds. The third run however does produce a latency peak and the logs do not show any entry after receiving the second signal, again supporting our theory.

6.2.6 Results Discussion

To discuss the results of our experiments, we compare the baseline latency to the measurements obtained once each fault was introduced. The baseline values are obtained from the first 300 seconds of each individual run, the remaining 300 seconds provide the values for each fault injection experiment. In addition to the latencies, we also check the recovery duration. Others have defined this as the time it takes to run a specific recovery procedure or how long the system is unavailable (Sangroya,



(A) 95th percentile latencies for NewOrder transactions for all runs



(B) Median values for 50th, 95th and 99th latencies for NewOrder transactions over all runs

FIGURE 6.3: Results of Client Node fault injection

TABLE 6.1: Latency values for the baseline scenario and all fault scenarios. The Client Node scenario is included twice, the second row uses only the data from the runs where the fault actually manifested.

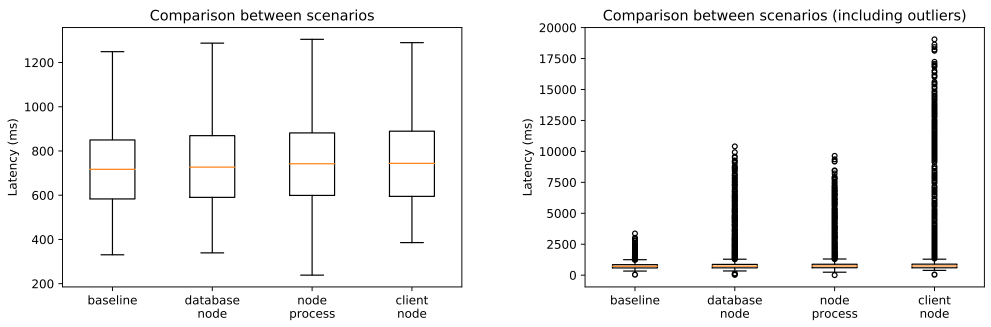
Scenario	μ (ms)	σ (ms)	Percentiles (ms)			Recovery (s)
			$q_{0.5}$	$q_{0.95}$	$q_{0.99}$	
Baseline	728	189	717	1000	1303	–
Database Node	848	761	727	1238	5108	11.40
Node Process	906	866	742	1591	5939	12.87
Client Node	1001	1697	721	1193	12094	–
Client Node – Manifested	1571	2871	744	9486	14802	30.35

Serrano, and Bouchenak, 2012a; Vieira and H. Madeira, 2003b). In our case CockroachDB remains available but we see very high latencies instead, so it makes sense to look at the recovery duration as the period during which we see these latencies. Because high latencies can also occur naturally, we define the starting time as the moment where 5 subsequent transactions have a latency of two standard deviations above the baseline mean. Similarly, we consider the system to have recovered when all transactions are within two standard deviations from the baseline mean for 5 seconds. The recovery duration then is the time between the first and last transaction that fall outside the interval. These definitions are based on what we have seen in our experiments, but would most likely be different when running other experiments.

For this analysis we only use the NewOrder transaction latencies because they are the most important for the TPC-C benchmark and because there is a significant difference in latencies between the various types of transactions. In contrast to the individual experiments, we do not bin the data but use the raw measurements to provide a more thorough evaluation. The TPC-C benchmark executes a NewOrder transaction in 45% of the cases, leading to about 2700 NewOrder transactions per run, half before and half after the fault injection. We used roughly 24000 data points to establish the baseline latencies and 8000 for each type of fault. We split the data obtained from the Client Node experiment into two groups, one containing all data points and one that only includes the data points from the runs where a fault was introduced successfully. The values can be found in Table 6.1 and a box plot comparison is shown in Figure 6.4. A larger table overview including percentage increases can be found in Table D.1 in Appendix D.

We find that our mean baseline latency is 728ms with a standard deviation of 189ms and a median latency of 717ms. The latencies are not quite normally distributed and a heavy right-side tail is present, which is expected for database systems and also illustrates why the 95th and 99th percentile latencies are important metrics. This is also visible in Figure 6.4b, where the baseline scenario shows a relatively large number of values above the top whisker, compared to values below the bottom whisker.

As can be seen in Table 6.1 and Figure 6.4, the Client Node fault impacts the system most severely, followed by the Node Process and Database Node faults. We see an increase in average latency and a large change in the standard deviations, even when compensating for the smaller sample sizes. We see a small shift in the median transaction latency, which is expected because the median is less affected by the outliers we introduced. The recovery duration is similar for the Database Node and Node Process faults, but twice as long for the Client Node fault. This causes the 95th percentile latency to increase significantly for the latter while it increases



(A) Box plot comparison excluding outliers to provide a view of the interquartile range

(B) Box plot comparison including outliers to show their severity

FIGURE 6.4: Box plot comparison of the `NewOrder` transaction latency for the baseline and all experiments

much less for the other two. The 99th percentile latency then shows that not only the duration was greater for the Client Node fault, the latency impact was much greater as well.

The impact of the Database Node and Node Process faults are quite similar, increasing the 99th percentile latencies by 292 % and 356 % with a recovery time of 11.4 and 12.9 seconds, respectively. The Client Node fault can have a much larger impact, increasing the 99th percentile latency by 1036 % with a recovery time of 30.3 seconds, if it manifests. The impact difference is due to the StatefulSet deployment and how Kubernetes reacts to these failures. In the first two cases Kubernetes detects that the container process is no longer running and restarts the affected pod, while in the case of the Client Node fault Kubernetes completely deletes the pod and reschedules it, which is a more intensive operation. Whether a fault actually causes an increase in latency depends on whether or not the affected instance can inform the other nodes and any connected clients before it terminates, meaning more abrupt shutdowns have a greater chance of destabilising the system.

Chapter 7

Discussion

In this chapter we reflect on the research performed in this master's thesis, how it can be applied and what can be done to improve it. We first discuss how the scenario framework can help to better understand various aspects of distributed database systems and then conclude this chapter by covering various avenues for future work.

7.1 Discussion

Traditional database management systems have been around for a long time. We have since grown accustomed to using these systems for numerous purposes and learned about their strengths and weaknesses through prolonged usage and thorough testing. In contrast, distributed database management systems are a recent appearance so it makes sense that we do not have the same mastery over this field. When we also take into account that these systems are more complex and operate at a much larger scale, then it is only logical that much remains to be learned about this technology.

Despite its relatively recent emergence, many advancements have been made to better understand performance-related aspects of distributed database systems through benchmarking. Yet dependability aspects are often overlooked and are, as a result of this, much less understood. Only a small number of benchmarks evaluate this, as discussed in subsection 3.1.2, even though researchers have been advocating the inclusion of dependability aspects (Almeida et al., 2010; Vieira and H. Madeira, 2009).

The framework designed in this thesis can play an important role in becoming more familiar with the intricacies of distributed database systems. We can study how a system behaves during various failures by introducing fault scenarios, and use this knowledge to better mitigate the impact of a failure or even prevent it entirely. The experiments performed in section 6.2 introduce a simple fault and focus on the impact on transaction latency, but it is very much possible to inspect other aspects as well. In addition to measuring the impact of faults, the framework could also be used to evaluate different system compositions or test specific settings before applying them to a production environment. This would require some modifications to the system, but thanks to its extensible design this should take relatively little effort.

7.2 Future Work

As mentioned a this thesis, our goal was to design an extensible framework and implement a proof-of-concept version. Some of the features discussed in chapter 4 are therefore not included in the PoC and remain as future work. In the following subsections we discuss those features and other possibilites.

7.2.1 More Extensive Scenarios

The proof-of-concept framework is limited to introducing relatively simple faults, which can be used for scenarios that simulate high-level node failures. A future improvement would be supporting more types of faults, triggers and instances.

Faults come in many forms and can be introduced in various levels of a system. It would be useful to include the injection of hardware and network faults in future versions of the scenario framework. This would allow us to test more types of components and provide a better understanding of the impact of these real-world situations. This would also require support for additional types of instances such as the CPU or network layers.

Including more types of triggers would also greatly improve the flexibility of scenarios. Conditional triggers could be used to introduce faults only when some condition is true, for example introducing a specific failure if a node's CPU usage reaches a certain threshold. Additionally, triggers could be recurring, introducing the same fault every 2 minutes to simulate some intermittent failure.

In addition to scenarios that introduce faults, we could also include options to test the scalability and elasticity of a system by increasing or decreasing the number of nodes. Another option would be running some special workload next to the database benchmark, for example making DDL changes or performing database maintenance to evaluate their impact.

7.2.2 Unified Controls

The scenario framework was designed to be able to communicate with both the system under test and benchmarking framework, but our proof-of-concept is only capable of limited interaction with the system under test through the database and orchestration clients. Including more features and support for the benchmarking framework would make it easier to control the various systems. Integrating controls for both the system under test and benchmarking framework would then also reduce the effort it takes to set up both systems and run tests.

7.2.3 Analysis Experiment Results

Currently, experiments must be performed by hand and data analysis is a manual effort as well, unless the benchmark framework supports this to some degree. One additional feature that would greatly improve the usability of the scenario framework is automatically aggregating the results from experiments. This would include collecting the results from the benchmarking framework and producing plots which indicate when specific faults were injected. Additionally, this module could be used to automatically calculate how the performance was influenced by each of the faults, further streamlining the testing process.

Chapter 8

Conclusion

In this chapter we look back at the research questions posed in the introduction and discuss how the research performed in this thesis has provided answers to these questions. Finally, we provide a short summary, thereby concluding this thesis.

8.1 Research Questions

In section 1.2 we introduced our main research question and the four subquestions that followed from it:

RQ 1 *How do faults affect the performance of distributed database systems?*

- **RQ 1.1** *How can we agnostically evaluate the impact of faults on various distributed database systems?*
- **RQ 1.2** *How can we reliably introduce and reproduce various types of faults?*
- **RQ 1.3** *How can we non-intrusively measure the impact of faults?*
- **RQ 1.4** *What is the performance impact of a high-level failure in a distributed database deployment?*

To answer our main research question, we designed a framework that is capable of introducing faults in a distributed database management system while it is running a benchmark in order to measure the impact that faults have on the performance of the system under test. The subquestions were vital to properly design and build such a framework. Answering RQ 1.1 helps us to find a solution that can be applied to any system, RQ 1.2 ensures experiments are consistent and repeatable and RQ 1.3 aids us to produce valid and trustworthy results. Finally, RQ 1.4 rephrases the purposely broad main research question to find an answer for a specific use case.

To find a solution to research question 1.1, we looked at various existing frameworks. We found that those we discussed in subsection 3.1.2 are generally limited to specific types of faults, databases and benchmarks. This makes it difficult to introduce the same fault in a wide array of systems, even tough sometimes equivalence between faults can be established. We identified the functionality we required and designed a modular framework that can be used in combination with any database, orchestration service and benchmarking framework. APIs are used for communication, and supporting a new system requires only an implementation of the respective API.

For research question 1.2, we turned to the available literature. We have covered a number of benchmarking frameworks that include dependability aspects in subsection 3.1.2, and further looked into fault injection in section 3.2. We found that there are multiple types of faults and some can be used to simulate another. To introduce these faults reliably, it is important to always be able target a specific instance at a specific time. We have done this by defining the faults and their targets beforehand through reusable configuration files and providing an abstraction of the system under test to target its instances. The Orchestration Client API implementation is then responsible for providing the correct mapping, which Kubernetes, the technique our proof-of-concept framework uses, has multiple methods for.

Research question 1.3 was also answered through literature. The frameworks discussed in subsection 3.1.2 essentially cover this question already. Most of these benchmarking suites evaluate the dependability aspects of the system under test by first establishing a baseline performance and then introducing a fault while the benchmark continues. In order for a benchmarking framework to be relevant and reliable, it must be able to measure the performance of a system without influencing it noticeably. To measure the impact of a fault, it is only necessary to compare the various metrics before and after the fault was introduced. This means that any benchmarking suite that is deemed reliable can be used for our purposes, as long as it can (be modified to) handle systems experiencing faults.

The main research question was stated broadly because the goal was to find an answer to the problem in general. The framework we designed is the answer to this question, although evaluating this solution for all possible systems is impossible. Research question 1.4 was formulated in order to learn how a specific deployment was affected instead. As covered in chapter 6, we used our proof-of-concept implementation to cause high-level node failures on three different levels and study the impact on the transaction latency of a CockroachDB cluster deployed through Kubernetes. We found that the increase in latency and recovery duration depends heavily on the type of fault and how abruptly it would shut down the targeted node.

8.2 Conclusion

In this Master's Thesis we have worked towards answering the following question: How do faults affect the performance of distributed database systems? Directly related to this were our subquestions, which concern system-agnostic evaluation, reliable fault injection and non-intrusive impact measurement. By researching literature and studying existing systems we were able to find answers to the subquestions, which in turn enabled us to propose a solution for our main research question.

As a solution we have designed a framework for injecting faults into a test system and implemented a proof-of-concept version. The goal of this design was to create a framework that was highly reusable so it could be applied to many different test systems, extensible so it would take little effort to support new types of faults or different systems, and capable of reliably reproducing experiments. To facilitate this, the framework is modular and interacts with database management systems, orchestration clients and benchmarking frameworks through APIs. Each of these requires their own API implementation while configuration files instruct the system which components to use and what faults to introduce in the system under test.

The framework can introduce faults through the database client and orchestration client, or it can access the system under test directly and execute commands there. Configuration files specify the faults and their conditions for injection, and the system processes these to produce a scenario. These can be simple, e.g. injecting a single fault after 5 minutes, or very complex, combining multiple faults and conditions to more closely simulate real-world scenarios. All components of a scenario are designed to be straightforward to extend, making it easy to create new components and define different scenarios.

The scenarios that our proof-of-concept implementation is capable of running are limited to three types of faults, each representing the high-level failure of a node. The basic implementation supports the DBMS CockroachDB and orchestration technique Kubernetes, and is capable of killing a database node through either the CockroachDB client, Kubernetes client or directly terminating its main process.

We tested our proof-of-concept framework by introducing faults in a 3-node CockroachDB cluster deployed on the Google Kubernetes Engine. We ran the TPC-C benchmark using the OLTPBench benchmarking framework to measure the performance of the system before, during and after introducing the faults. The impact of the faults was determined by comparing the results before and after injection. We learned that high level failures introduced through the database client and killing the main database process have a similar impact, but a high level failure introduced through the Kubernetes client can have a much larger impact if it manifests. The first two increase the 99th percentile latencies by roughly 300–350 % and take a little over 10 seconds to recover. The latter increases the 99th percentile latency by a little over 1000 % and recovers in roughly 30 seconds. This difference is explained by Kubernetes' recovery process, restarting the pod in the first two cases and completely rescheduling it in the third case. We also conclude that whether a fault has any impact at all on the CockroachDB cluster performance is highly dependent on whether or not the nodes and connected clients can be informed before terminating, meaning more abrupt shutdowns have a greater chance of destabilising the system.

For future work, we see three areas for improvement. The scenarios our proof-of-concept framework can run are limited, so expanding the types of faults and targetable instances would make the system more useful. Secondly, our implementation is only capable of limited interaction with the other systems, so integrating more controls would reduce the manual effort required for testing by coordinating and automating various processes. Finally, including a module for analysing the results of an experiment would greatly reduce the manual effort and make it easier to compare multiple experiments.

Appendix A

Scenario Module Class Overview

This appendix provides an overview of the various objects, classes and patterns used in the Scenario Module, as well as the relations between them. A graphical representation can be found in Figure A.1 and A.2.

A.1 Companion Objects

A companion object is an object that shares the name of a class and is defined in the same file. A number of classes in this module have companion objects, depicted with dashed borders in Figure A.1 and A.2. We use these objects for two reasons: to provide a number of static methods and to instantiate the classes for which they are companions.

The scenario director's companion object ensures only one scenario director instance exists. A companion object and its class can access each other's private properties and functions, so we can make the constructor private to force instantiation through the companion object. This way we can instantiate the scenario director once and return the existing instance when it has already been created.

The companion objects of the `Trigger`, `InstanceType` and `FaultType` classes use the Enumeratum library to retrieve their subtypes by name.[1] This library provides a number of features that are missing in Scala's default implementation, which allows us to dynamically find the types as defined in the configuration files. This is then used by the builders to correctly construct the specified types.

A.2 Builder Pattern

The builder pattern allows us to separate the construction of a complex class from its representation. Builders feature a number of methods for adding properties to an instance, and a build method for instantiating the class once the required properties have been supplied. We use builders to instantiate the scenario and its phases, triggers and faults.

To create a scenario and all of its components, the scenario director calls the `fromConfig` method in the scenario companion object and supplies the scenario configuration. The scenario companion object then extracts the relevant properties and calls the `fromConfig` function of the companion objects of the phase or trigger components, depending on how the scenario is defined. Both again extract the relevant configuration and then call the `fromConfig` function of the other components, and so forth. Finally, the faults are included in the triggers, and the triggers are included in the scenario. As can be seen in Figure A.1, phases are not part of the scenario

[1]Enumeratum library GitHub page: `https://github.com/lloydmeta/enumeratum`

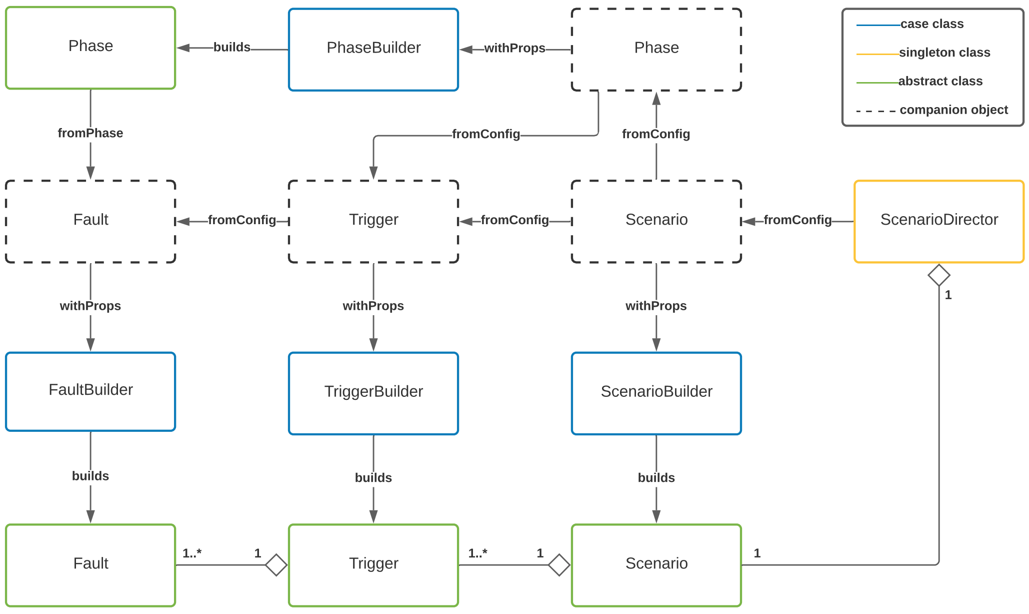


FIGURE A.1: An overview of the component builders and how they are linked to create a scenario

instance but are instead used to generate the faults and create the triggers from the configuration.

A.3 Inheritance

Multiple types of triggers and faults exist, each with their own behaviour thanks to (multiple) inheritance and type-matching. In the current implementation, a trigger is either a `TimedTrigger` or a `DependentTimedTrigger`. The former uses the `Timed` trait to specify a duration after which it should execute. The latter uses both the `Timed` and `Dependent` trait and will execute a specific time after the trigger on which it depends has executed. Defining new traits and combining them with existing ones is a straightforward and low-effort method for creating new types of triggers in the future.

Faults use an `InstanceType` and a `FaultType` to specify the type of instance to target and what kind of fault to introduce. The former is either a `Node` or `Cluster`, but the latter is more complex. At the highest level, three types of fault exist: `NodeFault`, `DatabaseFault` and `ClientFault`. These indicate the level at which the fault should be injected. The actual instances that are used are the `NodeProcessFailure`, `DatabaseNodeFailure` and `ClientNodeFailure`, each of which uses `NodeFailure` trait to indicate that they will cause a high-level failure in a node.

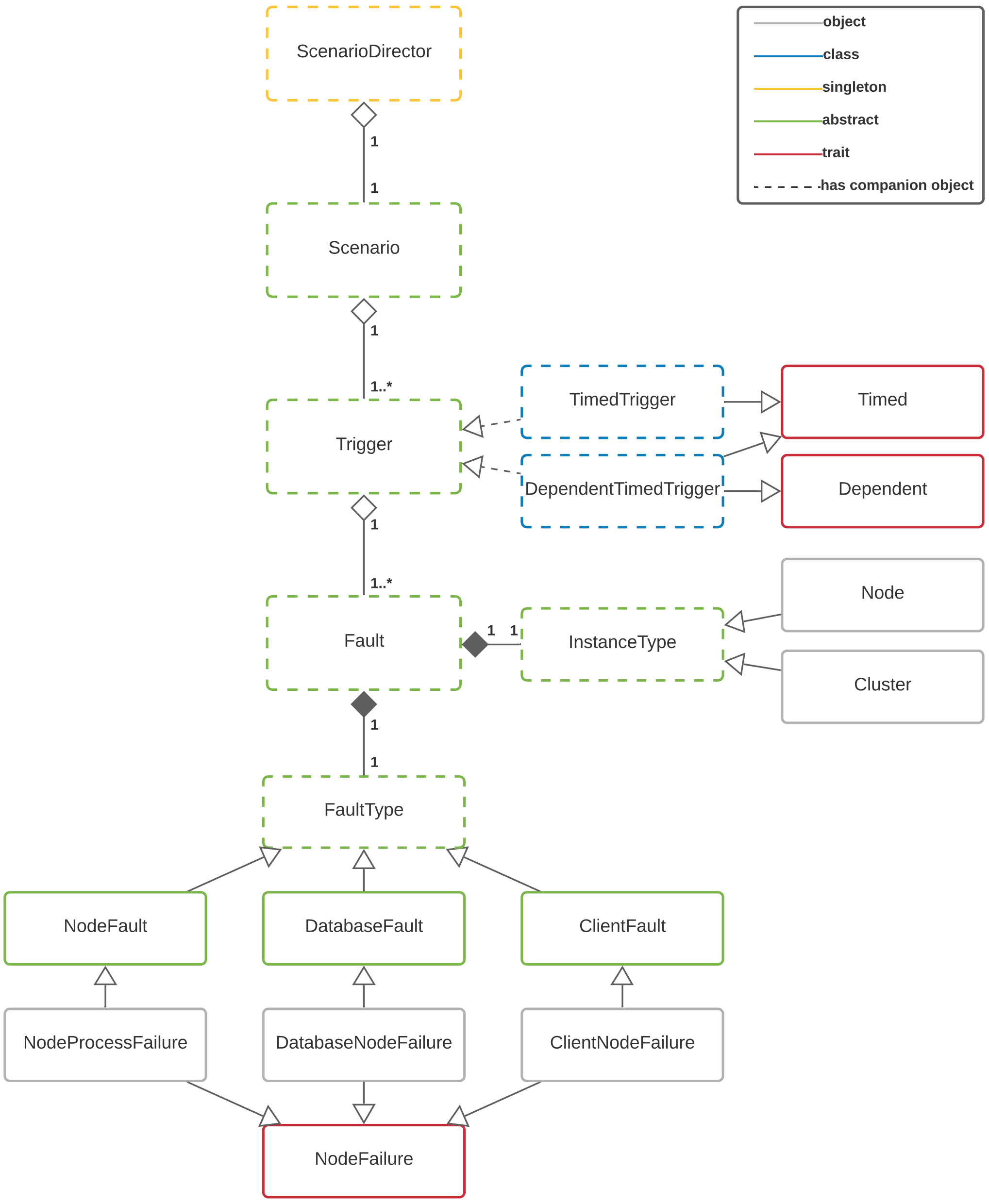


FIGURE A.2: An overview of the scenario component classes and their relations

Appendix B

Scenario Configuration Examples

In this appendix we show a number of examples of configuration files that define a scenario. Listing B.1 contains the configuration of a scenario defined with phases and Listing B.2 contains the configuration of a scenario defined with triggers. Both introduce the same fault, but the former is simpler to configure and does not always target the same instance, in contrast to the second scenario configuration.

LISTING B.1: A scenario defined with phases

```
name = "Single Node Failure - Node Process" # The name of this scenario

# The phases of this scenario
phases = [ ${phase_1} ]                 # Reference to the phase below

phase_1 = {
  fault_type = "NodeProcessFailure"     # The kind of failure to introduce
  instance_type = "Node"                # The kind of instance to introduce faults in
  num_instances = 1                     # The number of instances to affect
  spread = 1                            # The number of clusters to spread faults over
  trigger = ${trigger_1}                # Reference to the trigger below
}

trigger_1 = {
  type = "TimedTrigger"                 # The kind of requirement to execute
  # The additional configuration for this trigger, depends on type
  conf = {
    time = "10 minutes"                 # Time until trigger executes
  }
}
```

A more complex scenario can be found in Listing B.3. Here we simulate a scenario where the failure of a single node increases the pressure on the rest of the system, which then causes the other nodes to fail as well, effectively cascading the failure throughout the system. The `DependentTimedTrigger` used in this scenario injects a fault some time after another trigger has injected its faults. In this scenario we first affect a single node, followed by a second node two minutes after the first trigger executes, and two more a minute after the second trigger has executed.

LISTING B.2: A scenario defined with triggers

```
name = "Single Node Failure - Node Process" # The name of this scenario

# The triggers of this scenario
triggers = [ ${trigger_1} ]                 # Reference to the trigger below

trigger_1 = {
  id = "6e5a88ad-9acf-476f-87de-bdea756c1000" # UUID of this trigger
  type = "TimedTrigger"                     # The kind of requirement to execute
  # The additional configuration for this trigger, depends on type
  conf {
    time = "10 minutes"                     # Time until trigger executes
  }
  # The faults that are introduced when this trigger executes
  faults = [ ${fault_1} ]                   # Reference to the fault below
}

fault_1 = {
  fault_type = "ClientNodeFailure"          # The kind of failure to introduce
  instance_type = "Node"                    # The kind of instance to introduce faults in
  instance_id = "default_cockroachdb-0"     # Unique identifier of the instance
}
```

LISTING B.3: A complex scenario that simulates a failure with a cascading effect

```
name = "Cascading Node Failure" # The name of this scenario

# The triggers of this scenario
triggers = [
  ${trigger_1}
  ${trigger_2}
  ${trigger_3}
]

# The individual triggers of this scenario
trigger_1 = {
  id = "6e5a88ad-9acf-476f-87de-bdea756c1000"
  type = "TimedTrigger"
  conf.time = "10 minutes"
  faults = [ ${fault_1} ]
}
trigger_2 = {
  id = "6e5a88ad-9acf-476f-87de-bdea756c1001"
  type = "DependentTimedTrigger"
  conf {
    time = "2 minutes"
    # UUID of the trigger that needs to be executed before this can execute
    depends_on = "6e5a88ad-9acf-476f-87de-bdea756c1000"
  }
  faults = [ ${fault_2} ]
}
trigger_3 = {
  id = "6e5a88ad-9acf-476f-87de-bdea756c1002"
  type = "DependentTimedTrigger"
  conf {
    time = "1 minute"
    # UUID of the trigger that needs to be executed before this can execute
    depends_on = "6e5a88ad-9acf-476f-87de-bdea756c1001"
  }
  faults = [
    ${fault_3}
    ${fault_4}
  ]
}

# A base fault to reference and reduce copy-pasting
fault_base = {
  fault_type = "ClientNodeFailure"
  instance_type = "Node"
}

# The individual faults of this scenario
fault_1 = ${fault_base}
fault_1.instance_id = "default_cockroachdb-0"
fault_2 = ${fault_base}
fault_2.instance_id = "default_cockroachdb-1"
fault_3 = ${fault_base}
fault_3.instance_id = "default_cockroachdb-2"
fault_4 = ${fault_base}
fault_4.instance_id = "default_cockroachdb-3"
```

Appendix C

Benchmark Configuration

Here we include a number of configuration files related to executing the TPC-C benchmark on our CockroachDB deployment. Listing C.1 shows the configuration used to define the TPC-C workload and Listing C.2 contains all the parameters that were (automatically) set when executing the TPC-C benchmark.

LISTING C.1: Configuration for the TPC-C benchmark

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration>
  <isolation>TRANSACTION_SERIALIZABLE</isolation>
  <batchsize>16</batchsize>
  <poolsize>10</poolsize>
  <scalefactor>10</scalefactor>
  <terminals>10</terminals>
  <works>
    <work>
      <warmup>300</warmup>
      <time>600</time>
      <rate>10</rate>
      <weights>45</weights>
      <weights>43</weights>
      <weights>4</weights>
      <weights>4</weights>
      <weights>4</weights>
    </work>
  </works>
  <transactiontypes>
    <transactiontype><name>NewOrder</name></transactiontype>
    <transactiontype><name>Payment</name></transactiontype>
    <transactiontype><name>OrderStatus</name></transactiontype>
    <transactiontype><name>Delivery</name></transactiontype>
    <transactiontype><name>StockLevel</name></transactiontype>
  </transactiontypes>
</configuration>
```

LISTING C.2: Parameters set when executing the TPC-C benchmark on CockroachDB

```
{
 "disable_partially_distributed_plans": "off",
 "search_path": "$user,public",
 "datestyle": "ISO, MDY",
 "experimental_enable_temp_tables": "off",
 "intervalstyle": "postgres",
 "default_transaction_read_only": "off",
 "require_explicit_primary_keys": "off",
 "server_version": "9.5.0",
 "enable_implicit_select_for_update": "on",
 "application_name": "tpcc",
 "bytea_output": "hex",
 "optimizer_use_multicol_stats": "on",
 "lock_timeout": "0",
 "transaction_isolation": "serializable",
 "max_identifier_length": "128",
 "force_savepoint_restart": "off",
 "default_tablespace": "",
 "enable_zigzag_join": "on",
 "default_int_size": "8",
 "vectorize": "on",
 "idle_in_transaction_session_timeout": "0",
 "server_version_num": "90500",
 "tracing": "off",
 "enable_interleaved_joins": "off",
 "locality": "",
 "serial_normalization": "rowid",
 "transaction_priority": "normal",
 "synchronize_seqscans": "on",
 "experimental_enable_hash_sharded_indexes": "off",
 "sql_safe_updates": "off",
 "crdb_version": "CockroachDB CCL v20.2.0 (x86_64-unknown-linux-gnu, built 2020/11/09
     16:01:45, go1.13.14)",
 "disallow_full_table_scans": "off",
 "extra_float_digits": "3",
 "vectorize_row_count_threshold": "1000",
 "experimental_distsql_planning": "off",
 "max_index_keys": "32",
 "idle_in_session_timeout": "0",
 "transaction_status": "NoTxn",
 "timezone": "Europe/Amsterdam",
 "enable_seqscan": "on",
 "default_transaction_isolation": "serializable",
 "session_user": "roach",
 "distsql": "auto",
 "reorder_joins_limit": "8",
 "server_encoding": "UTF8",
 "database": "oltpbench_tpcc",
 "optimizer": "on",
 "optimizer_use_histograms": "on",
 "prefer_lookup_joins_for_fks": "off",
 "synchronous_commit": "on",
 "transaction_read_only": "off",
 "enable_insert_fast_path": "on",
 "row_security": "off",
 "client_encoding": "UTF8",
 "session_id": "165f9d44fd06749c0000000000000002",
 "client_min_messages": "notice",
 "default_transaction_priority": "normal",
 "foreign_key_cascades_limit": "10000",
 "integer_datetimes": "on",
 "results_buffer_size": "16384",
 "standard_conforming_strings": "on",
 "enable_experimental_alter_column_type_general": "off",
 "statement_timeout": "0"
}
```

Appendix D

Experiments Results Table Full

Due to limited space, the percentage wise latency changes were omitted when displaying the results of the experiments in Table 6.1. To still provide this data, we have included it here in the appendix in Table D.1.

TABLE D.1: Latency values and percentage increases for the baseline scenario and all fault scenarios. The Client Node scenario is included twice, the second row uses only the data from the runs where the fault actually manifested.

Scenario	μ (ms)	σ (ms)	Percentiles (ms)			Recovery (s)
			$q_{0.5}$	$q_{0.95}$	$q_{0.99}$	
Baseline	728	189	717	1000	1303	–
Database Node	848 (16.5%)	761 (302.2%)	727 (1.4%)	1238 (23.8%)	5108 (292.1%)	11.40
Node Process	906 (24.5%)	866 (357.5%)	742 (3.5%)	1591 (59.1%)	5939 (355.9%)	12.87
Client Node	1001 (37.5%)	1697 (797.2%)	721 (0.5%)	1193 (19.3%)	12094 (828.4%)	–
Client Node - Manifested	1571 (115.8%)	2871 (1417.7%)	744 (3.8%)	9486 (848.4%)	14802 (1036.3%)	30.35

Bibliography

Almeida, Raquel et al. (2010). "How to Advance TPC Benchmarks with Dependability Aspects". In: *Performance Evaluation, Measurement and Characterization of Complex Systems - Second TPC Technology Conference, TPCTC 2010, Singapore, September 13-17, 2010. Revised Selected Papers*. Ed. by Raghunath Othayoth Nambiar and Meikel Poess. Vol. 6417. Lecture Notes in Computer Science. Springer, pp. 57–72.

Arlat, Jean, Martine Aguera, et al. (1990). "Fault injection for dependability validation: A methodology and some applications". In: *IEEE Transactions on software engineering* 16.2, pp. 166–182. DOI: 10.1109/32.44380.

Arlat, Jean, Alain Costes, et al. (1993). "Fault injection and dependability evaluation of fault-tolerant systems". In: *IEEE Transactions on computers* 42.8, pp. 913–923. DOI: 10.1109/12.238482.

Chrysafis, Christos et al. (2019). "FoundationDB Record Layer: A Multi-Tenant Structured Datastore". In: *Proceedings of the 2019 International Conference on Management of Data*, pp. 1787–1802. DOI: 10.1145/3299869.3314039.

Cooper, Brian Frank et al. (2010). "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. New York, NY, USA: Association for Computing Machinery, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152.

Costa, Diamantino and Henrique Madeira (1999). "Experimental assessment of COTS DBMS robustness under transient faults". In: *Proceedings 1999 Pacific Rim International Symposium on Dependable Computing*. IEEE, pp. 201–208. DOI: 10.1109/PRDC.1999.816230.

Costa, Diamantino, Tiago Rilho, and Henrique Madeira (2000). "Joint evaluation of performance and robustness of a COTS DBMS through fault-injection". In: *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pp. 251–260. DOI: 10.1109/ICDSN.2000.857547.

Difallah, Djellel Eddine et al. (2013). "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases". In: *Proceedings of the VLDB Endowment* 7.4, pp. 277–288. DOI: 10.14778/2732240.2732246.

Durães, João and Henrique Madeira (2002). "Emulation of Software Faults by Educated Mutations at Machine-Code Level". In: *13th International Symposium on Software Reliability Engineering (ISSRE 2002), 12-15 November 2002, Annapolis, MD, USA*. IEEE Computer Society, pp. 329–340. DOI: 10.1109/ISSRE.2002.1173283.

— (2004). "Generic faultloads based on software faults for dependability benchmarking". In: *International Conference on Dependable Systems and Networks, 2004*, pp. 285–294. DOI: 10.1109/DSN.2004.1311898.

— (2006). "Emulation of Software Faults: A Field Data Study and a Practical Approach". In: *IEEE Transactions on Software Engineering* 32.11, pp. 849–867. DOI: 10.1109/TSE.2006.113.

Freels, Matt (2018). *FaunaDB: An architectural overview*. Tech. rep. Fauna. URL: https://fauna-assets.s3.amazonaws.com/public/FaunaDB-Technical-Whitepaper.pdf.

Fujita, Hajime et al. (2012). “DS-Bench Toolset: Tools for dependability benchmarking with simulation and assurance”. In: *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*. Ed. by Robert S. Swarz, Philip Koopman, and Michel Cukier. IEEE Computer Society, pp. 1–8. DOI: 10.1109/DSN.2012.6263915.

Ghazal, Ahmad, Todor Ivanov, et al. (2017). “BigBench V2: The New and Improved BigBench”. In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, pp. 1225–1236. DOI: https://doi.org/10.1109/ICDE.2017.167.

Ghazal, Ahmad, Tilmann Rabl, et al. (2013). “BigBench: Towards an Industry Standard Benchmark for Big Data Analytics”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: Association for Computing Machinery, pp. 1197–1208. ISBN: 9781450320375. DOI: 10.1145/2463676.2463712.

Gray, Jim (1993). *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. 2nd. Morgan Kaufmann. ISBN: 1-55860-292-5.

Huang, Dongxu et al. (2020). “TiDB: a Raft-based HTAP database”. In: *Proceedings of the VLDB Endowment* 13.12, pp. 3072–3084. DOI: 10.14778/3415478.3415535.

Le, Michael and Yuval Tamir (2014). “Fault injection in virtualized systems—challenges and applications”. In: *IEEE Transactions on Dependable and Secure Computing* 12.3, pp. 284–297. DOI: 10.1109/TDSC.2014.2334300.

Lu, Ruirui et al. (2014). “Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks”. In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pp. 69–78. DOI: 10.1109/UCC.2014.15.

Natella, Roberto, Domenico Cotroneo, and Henrique S. Madeira (2016). “Assessing Dependability with Software Fault Injection: A Survey”. In: *ACM Comput. Surv.* 48.3. ISSN: 0360-0300.

Powell, David (2012). *Delta-4: a generic architecture for dependable distributed computing*. Vol. 1. Springer Science & Business Media. DOI: 10.1007/978-3-642-84696-0.

Sangroya, Amit, Damián Serrano, and Sara Bouchenak (2012a). “Benchmarking dependability of MapReduce systems”. In: *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, pp. 21–30. DOI: 10.1109/SRDS.2012.12.

— (2012b). “MRBS: A comprehensive mapreduce benchmark suite”. In: *LIG, Grenoble, France, Research Report RR-LIG* 24.

— (2016). “Experience with benchmarking dependability and performance of MapReduce systems”. In: *Performance Evaluation* 101, pp. 1–19. DOI: 10.1016/j.peva.2016.04.001.

Stonebraker, Michael and Ariel Weisberg (2013). “The VoltDB Main Memory DBMS”. In: *IEEE Data Engineering Bulletin* 36.2, pp. 21–27.

Sullivan, Mark and Ram Chillarege (1992). “A Comparison of Software Defects in Database Management Systems and Operating Systems”. In: *Digest of Papers: FTCS-22, The Twenty-Second Annual International Symposium on Fault-Tolerant Computing, Boston, Massachusetts, USA, July 8-10, 1992*. IEEE Computer Society, pp. 475–484.

Taft, Rebecca et al. (2020). “CockroachDB: The Resilient Geo-Distributed SQL Database”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. New York, NY, USA: Association for Computing Machinery, pp. 1493–1509. ISBN: 9781450367356. DOI: 10.1145/3318464.3386134.

Ventura, Luís and Nuno Antunes (2016). “Experimental Assessment of NoSQL Databases Dependability”. In: *2016 12th European Dependable Computing Conference (EDCC)*, pp. 161–168. DOI: 10.1109/EDCC.2016.30.

Vieira, Marco and Henrique Madeira (2002). “Definition of Faultloads Based on Operator Faults for DMBS Recovery Benchmarking”. In: *9th Pacific Rim International Symposium on Dependable Computing (PRDC 2002), 16-18 December 2002, Tsukuba-City, Ibarski, Japan*. IEEE Computer Society, pp. 265–274. DOI: 10.1109/PRDC.2002.1185646.

— (2003a). “A Dependability Benchmark for OLTP Application Environments”. In: *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*. Ed. by Johann Christoph Freytag et al. Morgan Kaufmann, pp. 742–753. DOI: 10.1016/B978-012722442-8/50071-9.

— (2003b). “Benchmarking the dependability of different OLTP systems”. In: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* Pp. 305–310.

— (2009). “From Performance to Dependability Benchmarking: A Mandatory Path”. In: *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*. Ed. by Raghunath Othayoth Nambiar and Meikel Poess. Vol. 5895. Lecture Notes in Computer Science. Springer, pp. 67–83. DOI: 10.1007/978-3-642-10424-4_6.

Wang, Lei et al. (2014). “BigDataBench: A big data benchmark suite from internet services”. In: *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, pp. 488–499. DOI: 10.1109/HPCA.2014.6835958.