Computing Science — University of Amsterdam

Bachelor Computing Science

University of Amsterdam

Consistency analysis of CockroachDB under fault injection

Max Wouter Grim

June 8, 2016

Supervisor(s): Raphael 'kena' Poss

Signed:

Abstract

With the exponential growth of data we as humans collect, data storage is more important than ever. Storage systems are generally assumed to be fault tolerant and database engines rely on these systems working properly. Lesser known is that silent data corruptions do occur. Data engines often promise to be robust, highly consistent, fault-tolerant, survivable and durable. But what happens in the event that the database receives corrupted data? This project will set a stepping stone towards new research on database robustness in the presence of simulated data corruptions. For this project an open-source database testing framework named Jepsen is extended with a script simulating silent data corruptions. Additionally two workloads are defined, one simulating money transfers for a bank, and another simulating a monotonic function in order to test index corruption. All in all, two bank tests showed erroneous behaviour under fault injection. Moreover, one case is identified where inconsistencies occur when using database indexes.

Contents

1 Introduction **7**
1.1 Challenge . . . 8
1.2 Contribution . . . 8
1.3 Thesis outline . . . 9

2 Background & Related work **11**
2.1 Related work . . . 11
2.1.1 Fault injection frameworks . . . 11
2.1.2 Database testing . . . 12
2.2 Data corruption . . . 13
2.3 Database consistency . . . 15
2.3.1 ACID . . . 15
2.3.2 ACID transactions . . . 16
2.3.3 Snapshot isolation . . . 17
2.4 CockroachDB . . . 17

3 Methodology **19**
3.1 Test environment . . . 19
3.2 Workloads . . . 20
3.2.1 Banking test . . . 20
3.2.2 Index corruption . . . 22
3.3 Fault injection . . . 26

4 Results **29**
4.1 Banking test . . . 29
4.1.1 Invalid tests . . . 32
4.2 Index test . . . 35
4.2.1 Invalid tests . . . 36

5 Conclusion **39**
5.1 Discussion . . . 40
5.2 Future work . . . 40

6 Appendix **41**
6.1 Failed test 1 . . . 41
6.2 Failed test 2 . . . 42
6.2.1 Transaction details . . . 43

CHAPTER 1

Introduction

We are living in an economy that is becoming more and more driven by data [61]. Where in the past decades software primarily generated money, in the present this role has arguably been transferred to data. The amount of data we as humans collect grows exponentially (see Figure 1.1). Data production in 2020 will be 40 times greater than it was in 2009 [19]. With this development, storing huge amounts of data efficiently and reliably is paramount.

Much of this data is stored in databases. Databases are used in software from small web applications, such as web-shops or Internet fora, to large corporations like banks or the government. We trust database software with important data and therefore have certain expectations of their functioning. Arguably data from an Internet forum is not that critical. However, for a more sensitive application, such as a bank or governmental organisation, reliably storing the data is critical. Database systems are considered reliable if their transactions comply with a certain set of properties. ACID (Atomicity, Consistency, Isolation, Durability) is probably the most important set of properties defined for reliable transactions, providing a good judgement for the quality of a database management system [26]. Atomicity, consistency and isolation describe the semantic properties between server and client, whilst durability concerns itself with the integrity of committed data. Once data is successfully stored it should remain as so, without losing its integrity. Adhering to these properties might seem like a trivial task, yet this is not as straightforward as it seems. For non-clustered database systems these properties pose implementation challenges, for distributed database systems they are even more challenging [5]. Not all database systems implement these properties equally well, leading to a variance in quality.

Meanwhile the popularity of Cloud infrastructures has surged. Companies including Amazon[1], Google[2] and Microsoft[3] all responded to this demand by providing Cloud services in various forms. Cloud infrastructures are highly scalable and cost-effective, making them ideal for companies with increasing demands. Be that as it may, Cloud infrastructures do not provide the same hardware stability that dedicated servers bring. Instances might undergo physical migrations within a datacentre or temporarily suffer from decreased availability of resources. Furthermore, all large Cloud companies had several substantial outages in the past [4][18][58]. Though inconspicuous, these instabilities could have a hefty impact on database systems. These instabilities may not, under any circumstance, cause the database system to violate the ACID properties. For example, pending transactions may not leave any traces visible for the client after a crash. In other words, transactions should be atomic. These types of potential complications are well known and therefore much effort has recently been put into preventing these kinds of errors, making database systems Cloud-ready.

[1]Amazon Web Services: https://aws.amazon.com/
[2]Google Cloud Computing: https://cloud.google.com/
[3]Microsoft Azure: https://azure.microsoft.com/

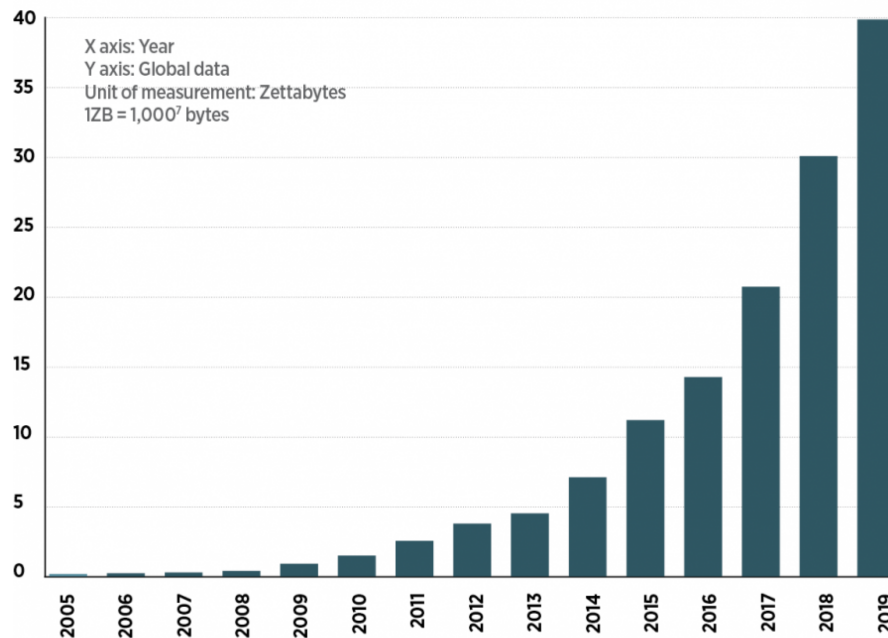


Figure 1.1: Data growth plotted per year. Post 2013 figures are predicted (Data source: UNECE, image source: [62]).

1.1 Challenge

In contrast to these familiar problems there is another, lesser known category of issues that could lead to erroneous behaviour in database engines. Storage systems are generally assumed to be fault tolerant and database engines rely on these systems working properly. Should errors occur, the storage system is expected to report these problems to the software requesting the data, either by responding with an error code, closing the stream or ultimately crashing the operating system. This way database systems are given the chance to respond accordingly, preserving ACID guarantees. Errors that are not detected by the storage system are called silent data corruptions.

Data corruption occurs more often than one might think, both in memory and hard drives [54]. Hardware is susceptible to ageing, bit rot, component failures and external factors such as cosmic rays. State of the art datacentres have grown immensely, containing millions of storage devices. As a consequence, chances of data corruption have increased [23]. Under such circumstances it is not unthinkable that silent data corruptions might slip through.

Database engines often promise to be robust, highly consistent, fault-tolerant, survivable and durable. The robustness of database engines relies on the fact that errors in the storage are detected by the storage system. But what happens in the event that the database receives corrupted data? Does the database system provide some method of corruption detection, and does this method work? Is it despite this still possible for the client to receive corrupted data? Do other errors occur during fault injection? What are those errors?

1.2 Contribution

Research on the effects of silent data corruptions on database systems is sparse. This project will set a stepping stone towards new research on database robustness in the presence of simulated silent data corruptions.

This project will investigate CockroachDB, a distributed transactional SQL database [35].

Around 2009 NoSQL databases started to gain popularity as an alternative to relational database systems [52]. Throughout the years, their developers started to value the advantages of transactional databases. This is where NoSQL databases started to embrace features from these transactional systems. CockroachDB is such a database. It is built upon RocksDB, a strongly-consistent and transactional key-value store [51]. CockroachDB is developed to support distributed strongly consistent ACID transactions. When configured correctly it is claimed to survive disk, machine, rack and even datacentre failures with minimal latency.

Simulating silent data corruptions is generally done with the help of a fault injection framework. The developers at Cockroach Labs devised several tests in order to verify the capabilities of their engine, though none of them are conducted under the influence of simulated data corruptions [48]. The existing tests are built with the help of an open-source database testing framework called Jepsen [31][32].

This research will extend the tests devised by Cockroach Labs in order to test how the engine holds under simulated silent data corruptions.

1.3 Thesis outline

This thesis will start at Chapter 2 with a study on related work done in this field of research. Next, Chapter 2 will study three subjects relevant to this project: data corruption, database consistency and the CockroachDB database system. Chapter 3 describes the methodology used to test the engine and explains the implementation details. Results collected with the experiments are depicted in Chapter 4. Finally, Chapters 5 and 6 will draw conclusions from the collected data and discuss the research in further detail.

CHAPTER 2

Background & Related work

This chapter will start by reviewing related work done in this field of research. Next, three important topics relevant to this project will be studied. First of all we will cover the possible causes of (silent) data corruption and discuss how often these corruptions do occur. Next database consistency will be addressed. Finally, we review some details of the relatively new CockroachDB engine.

2.1 Related work

This section will describe related work done in this field of research. Firstly several fault injection frameworks will be described. These frameworks are used in similar scenarios and have their own advantages and disadvantages. Thereafter related research on databases is reviewed.

2.1.1 Fault injection frameworks

The frameworks considered for this project are all SWIFI (software fault-injection) frameworks, which in contrast to hardware fault-injection frameworks do not require specialised hardware. Data corruptions are easily generated with software by flipping bits in memory cells, hard drive sectors or individual files [30]. Using software not only eliminates hardware costs but also eases creating experiments with different fault models.

There is a variety of test frameworks available, all with their own pros and cons. Still, most of them share the same system structure. Fault injection frameworks generally control one or multiple target systems (see Figure 2.1). The goal of fault injection frameworks is to run one or more experiments containing a workload. These workloads are generated by a workload generator. The controller, which can be located either on the system itself or on an external control host, makes sure all experiments are monitored and starts the different fault injectors and workloads [30]. A few of these frameworks are shown below.

NFTAPE is a framework designed to inject a high variety of fault models. It does so by using LWFI's (Lightweight Fault Injectors) [56]. The designers have chosen this approach as other frameworks proved hard to port to new systems. LWFI's are still system dependent, but easy to implement as they are lightweight. Other functionality, including logging and communication, is taken care of by the framework independently. This allows the user to easily implement new fault scenarios, as only the LWFI needs to be written. LWFI's follow a default interface, making them easy to interchange. Tests are coordinated by a control host, which in turn communicates with all the target nodes. Each target node runs its own process manager that makes sure the correct workloads are executed. The type of fault injection is defined by the LWFI, which in turn is triggered by a fault trigger. Fault triggers exist in many forms: application state, timer, performance counter, random, et cetera.

Stott et al. used NFTAPE to inject memory faults into a scientific image processing application [56]. It resulted images clearly distorted to the human eye. The research does not describe a fault injector that simulates data corruption. However, according to the paper it should be

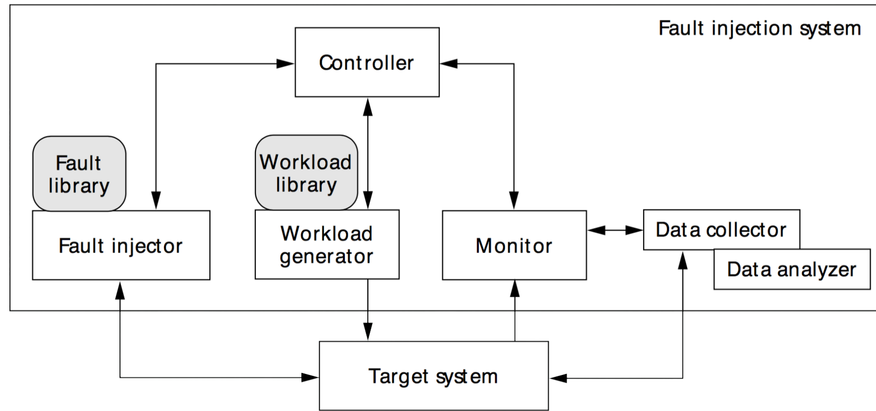


Figure 2.1: Typical fault injection system (Source: [30]).

relatively easy to do so by by using a LWFI. The NFTAPE framework is unfortunately only available under license.

Xception is a software fault injection framework that uses the processor to inject faults. Carreira et al. utilised the debugging and performance monitoring features of the PowerPC 601 processor to inject faults into the software [7]. This allowed monitoring the effects of the injections with minimal interference to the application. Their results show that 73% of all faults led to the application producing incorrect results. Xception is an older framework, implemented on the PARIX operating system, which makes it unfit for our purposes.

The Library-Level Fault Injector (LFI) is designed to inject faults at (shared) library level. The framework consists of two main components: the profiler and the controller [34][36]. The profiler scans for exported functions and their corresponding error response codes in libraries, and automatically generates test cases for all the functions it finds, called the fault profile. This fault profile, combined with a fault scenario, is then fed to the controller. The fault scenario describes a sequence of faults to be injected based on defined triggers. From this data the controller creates so called interception stubs. These stubs sit between the application and the original library. When triggered, the interception stub manipulates the response by modifying the stack, returning an error code instead of the original response. LFI only supports fault injection by response codes, though it could probably be expanded by manipulating other data structures as well. All execution information is collected in a log together with a replay script. This replay script can be used to replay the experiment and diagnose/debug the results. In practice, LFI is used in combination with MySQL, revealing several bugs in the engine [37].

2.1.2 Database testing

Subramanian et al. tested the influence of type-aware pointer corruption (TAC) on the MySQL engine [57]. As it is possible that disks become corrupted, the database engine should detect these inconsistencies. However, their results show that of the 145 faults they injected 110 resulted in serious issues. Moraes & Martins used a fault injection tool called Jaca validated an ODBMS component called Ozone. In 45 of the 2700 conducted experiments they observed a failure [43]. Ng & Chen investigated the influence of reliable memory on Postgres95 (a predecessor of PostgreSQL [49]) under fault injection [42]. They showed that in 2.3% to 2.7% of the test cases the database got corrupted. Zheng et al. tested the resilience of eight widely used databases under power faults [63]. They found seven of the eight databases failing to adhere to the ACID rules under power faults. In contrast Brown found the SQL 2000 server to be robust to a wide range of storage faults [6].

2.2 Data corruption

Nowadays, especially with large-scale IT infrastructures housing thousands of storage devices, component failures occur frequently [23]. Hard disk drives are known to have many (moving) components, which slowly degrade during their lifetime [55]. Electrical components might corrode over time or the motor may fail, resulting in the entire disk to fail. Hence, disk drives are known to fail coincidentally. Schroeder & Gibson analysed 100.000 disks over an extended period of time and found annual disk replacement rates of 2-4% [54]. Though problematic, whole disk failures are not the focus of this project.

Besides disk failures, storage systems also suffer from (silent) data corruptions. Data corruptions are errors in the storage system that occur unnoticed, resulting in the storage system possibly returning faulty data to the user. Bairavasundaram et al. have shown data corruption occurrence is substantial [2]. They studied a large production storage system containing 1.53 million disk drives of various models over a period of 41 months. During this period the disks encountered a grand total of over 400.000 checksum mismatches. Moreover, for some models a distressing 4% of the disks suffered from checksum mismatches in a period of 17 months. As a final point, multiple repair and/or checking tools exist for several database systems including SQL server [39], MySQL [40] and Oracle [45]. Altogether this shows that data corruptions occur and should be taken into consideration when developing applications. Data corruptions can be caused by both software- and hardware errors. In many cases the cause of the corruption can not be identified.

To start with, disk controllers contain more firmware, chips and processing power than one might think. Whereas old computer systems controlled the disk directly using the CPU, over time this responsibility shifted to the disk drive. As an example, firmware from Seagate contains more than 400.000 lines of code [28]. Big software projects inevitably contain bugs, and disks are no exception. Issues in the firmware could cause numerous data corruptions, such as lost- or misdirected writes [41][57].

Secondly, disk drives suffer from a phenomenon called “bit rot” or bit flip. Traditional hard disk drives are mostly magnetic. Bit rot is a term used to indicate that one or several bits on a magnetic platter have turned sides, resulting in a change of data. Firmware of disk controllers correct most of these errors with the help of error correcting code (ECC), though not all errors are detected [33]. Not only magnetic drives but also flash/DRAM based storage devices suffer from bit rot. SSDs gained popularity over the past decade as they perform better than traditional disk drives. Cosmic rays, which are high-energy particles from space, have the power to flip bits inside the flash memory of SSDs or DRAM memory [22]. Bit rot in DRAM memory poses its own problems. In the event that data is successfully stored on a (magnetic) disk drive, down the line retrieving the same data could still be perceived as corrupt when DRAM is corrupted. As a matter of fact, even controller firmware could be modified as a result of bits being flipped, possibly leading to erroneous firmware execution. Expensive ECC memory is available on the market, able to detect and correct bit errors and immune to single-bit errors. Because ECC memory is relatively expensive, it is mostly used by scientific and financial organisations. Moreover, it is due to this reason datacentres rarely use this ECC memory in their servers.

Lastly, single disk storage sizes have shown a constant growth over the last decades (Figure 2.2). Growing the size of a single disk drive comes down to increasing the number of platters and the data density per platter. Consequently, reading data becomes both more complex and error prone. Finally, latent sector errors recently gained attention.

Techniques have been developed over the years trying to prevent these kinds of errors. Checksums exist at different levels (i.e. block-, sector- or page level), attempting to detect and recover from errors. Furthermore, effort has been put into improving error correcting code. Additionally RAID, which depending on the level generates mirrors or parity data, can be configured in order to improve redundancy. Altogether this results in a list of possible techniques, where ironically a poor combination of choices may lead to problems itself [33].

RAID (redundant array of independent disks) is a technology primarily developed for combining multiple disks redundantly into one virtual disk [46]. Depending on the RAID level, several methods are used to recover from the loss of one or more whole disks. RAID 5, for example, generates parity information on every write. To improve redundancy this parity information is

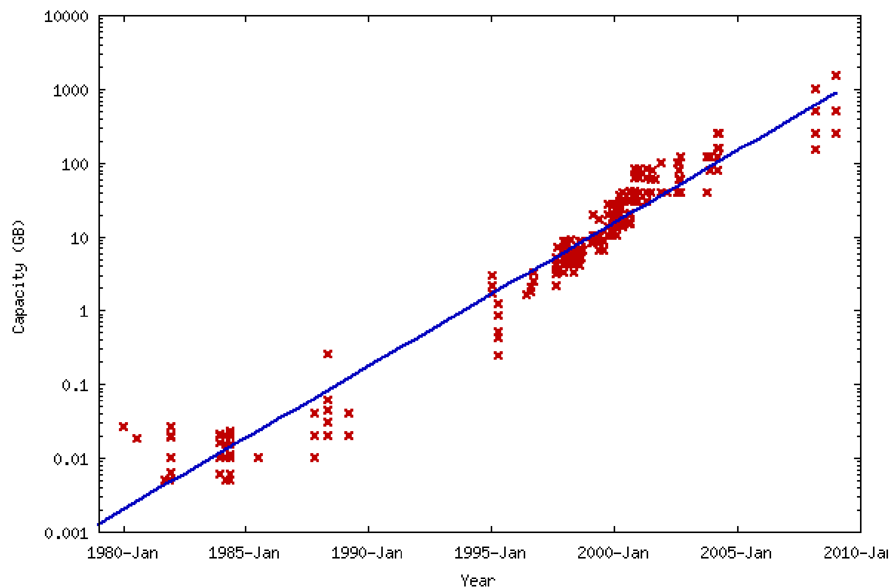


Figure 2.2: Hard drive capacity over time (Source: [27]).

distributed over the remaining available disks. In the event of a failed disk RAID 5 is able to recover the data of the lost disk by recalculating it using parity information residing on the other disks [12]. Thus RAID improves the reliability of storage systems in the event of a lost disk by adding redundancy. However, RAID is not designed to detect silent data corruptions on its own [2]. With RAID 5, when a parity block is corrupt, the parity computation will be incorrect. Yet, RAID can not detect which block is corrupt.

Ironically, RAID is in some reconstruction cases the actual cause of data corruptions. Krioukov et al. studied production systems and found cases of parity pollution [33]. Parity pollution is a type of corruption that occurs in the RAID parity blocks. In the event of a disk failing, RAID will reconstruct this disk using the possibly corrupted parity data, consequently spreading the corruption across the array. Bairavasundaram et al. found that on average 8% of the drive corruptions were detected during RAID reconstruction. One might say that data scrubbing, a method periodically scanning the disk for errors, will reduce these changes. Not only is data scrubbing computationally expensive, but recent insights prove data scrubbing to be the main cause for parity pollution [33].

It should be noted that not all systems consist of multiple disks, making them unsuitable for a RAID configuration. Even if machines have multiple disks, RAID is not configured by default. Moreover, little info is available on the internals of Cloud infrastructures, making it difficult to verify whether RAID is configured on those platforms.

Moreover, in the world of system design there is a principle called the end-to-end principle. This principle states that error correction should always be done on the highest level [53]. Solving errors in lower levels helps the database system correcting errors, allowing for better performance, but does not solve the whole problem.

In summary, (silent) data corruptions do occur, and although several attempts have been made to reduce the odds of these corruptions, they can not be fully prevented. The right combination of faults and repair activities may still result in data corruptions, and higher level software, such as a database system, should take care into detecting these errors [33]. The risk of receiving corrupted data from the storage system is existent, and should be acknowledged by software developers.

2.3 Database consistency

Database consistency can be defined in different ways. One definition states that any transactions started in the future must see the effects of transactions committed in the past [24]. Another definition states that database constraints must not be violated, meaning that triggers, cascades and constraints should hold under transaction commits. And yet another definition states that consistent transactions should bring a database from one valid state to another. Consistency exists in many forms, from field type constrains all the way up to data consistency over many nodes in a distributed system. Over the years multiple theories have been formed on consistency models, including ACID, CAP, BASE, and PACELC.

Database engines are expected to be (atomically) consistent. We exemplify this with a bank transferring money. Consider a scenario where money is transferred from account A to account B. Doing so requires the system to subtract a certain amount of money from account A and adding this same amount to account B. Translated to database commands this sequence of operations requires two SELECT and two UPDATE queries. Take for example a transfer of €10 from account A to account B. The starting balance of account A is €150, and account B holds €225.

1. The first SELECT query retrieves the balance of account A, and establishes this is €150. This SELECT query is done for two reasons. Firstly, the current balance is needed to determine whether there is enough money available for the transfer. Secondly, the retrieved value is used to set the new balance.
2. The first UPDATE query updates the balance of account A to €140.
3. The second SELECT query retrieves the balance of account B, and establishes this is €225.
4. The second UPDATE query updates the balance of account B to €235.

In a real world situation, the database system of a bank would execute thousands of transactions simultaneously. This is where isolation and consistency become very important. Below we will illustrate a situation where this could go wrong. For this example we will use the same accounts A and B but add an extra account C with a balance of €75. In this example two transactions are described, T1 and T2, where T1 is the transaction described above and T2 is a new transaction, transferring money from account A to account C.

1. T1 is processing a transfer from account A to account B.
2. T2 is processing a transfer from account A to account C.
3. T2 updates the balances of account A and account C
4. Meanwhile, T1 is also calculating the new balance for account A, but does this with the same value T2 initially received. When T1 now updates the balance of account A, the changes made by T2 are lost, and extra money is created.

Another discrepancy that may occur in step 4 is seeing the newly written value of T2, breaking isolation. This could make A negative for example, where this may not be allowed. Moreover this illustrates the importance of atomic transactions. Imagine the database system has updated the value for account A but crashes before the update on account B is executed. Would this be the case, €10 is lost in the process. Such a scenario would be unacceptable in a real-world application. This can be prevented by implementing snapshot isolation, which is reviewed in the CockroachDB section.

2.3.1 ACID

ACID describes four characteristics (Atomicity, Consistency, Isolation, Durability) every transaction on a database should enforce in order for a database engine to be reliable [26]:

Atomicity Only transactions that completely succeed should be committed. If a part of the transaction fails, the whole transaction should be rolled back as if it never happened.

Consistency A transaction should bring a database from one valid state to another, without ever observing inconsistent data or producing inconsistent data. This means checking before and after a transaction whether the data is consistent, specified by rules based on constraints, cascades and triggers.

Isolation All transactions should execute as if executed serially. One transaction may not observe the transitional state of another ongoing transaction.

Durability Committed transactions should never be lost, even under the influence of crashes or errors.

By processing all requests to a database engine serially these restrictions would be relatively easy to maintain, but this would have a significant impact on performance, as any form of concurrency is eliminated. Therefore concessions have to be made.

2.3.2 ACID transactions

CockroachDB achieves distributed ACID transactions by using the following phases during a transaction: switch, stage, filter, flip and unstage [59].

Before a transaction modifies a value it first creates a *switch* [59]. This switch can not be accessed concurrently and is represented by a Boolean: it can either be on or off and is off by default. Together with the switch a *transaction record* is created. CockroachDB uses transaction records internally to manage transactions, and has either a state of **PENDING**, **ABORTED** or **COMMITTED**.

When the transaction record is created and linked to the switch, the staging phase is entered. In the staging phase the engine writes the modified value to the database. The original value is not overwritten but instead a new record is inserted called a *write intent* (see Figure 2.3).

The next phase is the staging phase. In this phase the database engine stages the transaction changes. It does so with a 'write intent'. A write intent does not overwrite the original value but stores it in its proximity.

Would another client ask for the value which the transaction is updating, it would find the intent. Through this intent, it will find the corresponding transaction record with the switch. Would the switch be off, the original value would be returned to the client. In the case the switch is on, the new value which resides in the write intent will be returned. This is shown in Figure 2.4 and is called the filter phase.

In the flip phase the transaction record will update its state to **COMMITTED** and turn the switch on. This will return the updated value to all new clients requesting the record. This completes the transaction. Because the values still consist of write intents, there is some cleaning up to be done. This is done in the final 'unstage' phase. The cleaning up is done because of performance reasons, as filtering is expensive (it requires to communicate between nodes to

MVCC Store with Intent on Key A

Key	Timestamp	Value
A<intent>	500	"proposed_value"
A	400	"current_value"
A	322	"old_value"
A	50	"original_value"
B	100	"value_of_b"

Figure 2.3: Write intent on key A (Source: [60]).

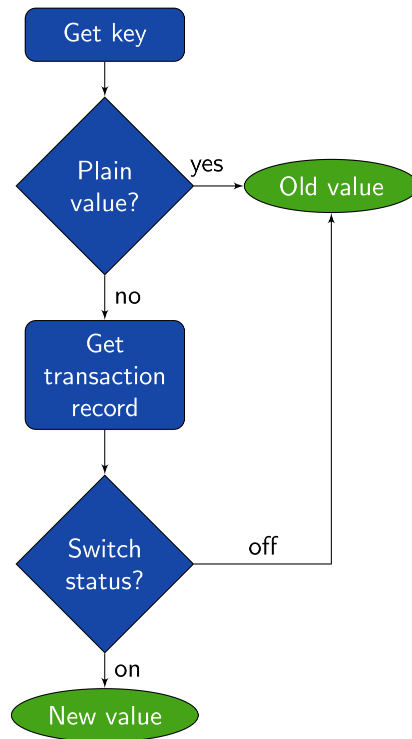


Figure 2.4: Transaction flowchart.

obtain the transaction record). This is done by removing the write intent and writing the final value [60]. Figure 2.4 shows an example of a request.

2.3.3 Snapshot isolation

Transactions should not see intermittent or uncommitted data that results from other unfinished transactions. This is where SI (snapshot isolation) comes into play.

SI is something that is not mentioned by ACID. One of the oldest models of transactional databases was the ACID model. In the time that the ACID model was developed, there was more focus on individual nodes than the notion of a distributed database. Database engines were mostly sequential, implying linearisability. Therefore there was not any distinction between linearisability and serialisability. As a result, ACID does not suffice anymore when talking about distributed databases.

Snapshot isolation essentially enables two things. Firstly it enforces that transactions only see data from transactions that are already committed. It has some form of a snapshot of the database's data of the moment the last transaction is committed [3]. This means that it reads the last committed value from a list of committed values at the beginning of the transaction. Secondly it only allows a transaction to commit when the updates it has made have not resulted in a conflict with other updates that were done concurrently. Snapshots can be extended by serialising. This is called SSI (serialised snapshot isolation).

2.4 CockroachDB

Databases can roughly be divided into two types: relational databases and NoSQL databases. The concept of relational databases is developed around 1970 [10], whereas NoSQL databases started to gain popularity around 2009 [52]. Currently more than 255 different NoSQL databases of different types exist, including key/value-, document-, graph- and columnar databases [13][52]. Because the amount of data grows, database systems have to scale.

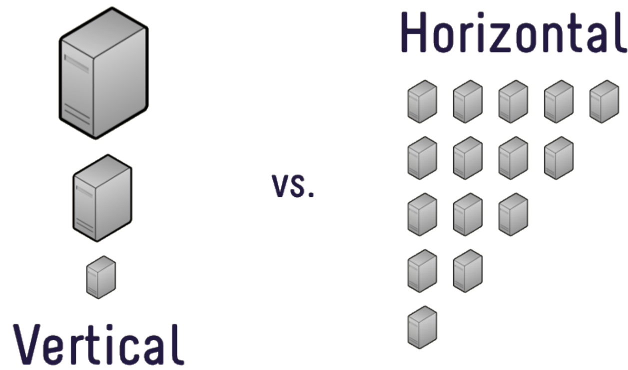


Figure 2.5: Scaling vertically/up vs. horizontally/out (Source: [20]).

Scaling can be done in two dimensions, either vertically (up) or horizontally (out) (see Figure 2.5). Scaling vertically means adding processors, storage and/or memory to a single machine. Not only are bigger machines more expensive, there is also a limit to which you can size a single machine. The better alternative is to scale horizontally by combining multiple machines into a single cluster. Scaling horizontally is more cost effective and contributes to the overall availability of a database.

For relational databases scaling horizontally was troublesome as they were (initially) not designed to run in clusters [52]. NoSQL databases tend to scale more easily, allowing vast amounts of data to be stored in a distributed manner. However, by distributing a database across multiple machines, it is harder to maintain ACID transactions.

Throughout the years NoSQL developers started to value the advantages of transactional databases. This is why NoSQL databases such as CockroachDB started to embrace features from these transactional systems. CockroachDB is an open-source lock-free distributed SQL database and currently in beta stage[1]. It is developed to support distributed strongly consistent ACID transactions using the RAFT consensus algorithm [44]. It is built upon the transactional and strongly-consistent key-value store named RocksDB [51]. The developers claim CockroachDB is able to survive disk, machine, rack and even datacentre failures with minimal latency [35].

Though it might seem as a new approach to use a key/value store as the back end of an SQL database, other database engines have had the same design, including MySQL, Sqlite4 and other engines [47]. Though it is possible to setup multi node clusters, CockroachDB is also able to run as a single instance. It can scale horizontally by joining a running node (that may be connected to a cluster).

CockroachDB is configured by default to store 3 replicas of its data. In the case a node crashes, the replicate data is automatically rebalanced among the other nodes. This makes sure the database is highly available. New locations in the cluster are identified and missing replicas are re-replicated in a distributed fashion [47].

[1] Version beta-20160421 will be used in the course project.

CHAPTER 3

Methodology

This chapter will go into detail on how CockroachDB is tested under fault injection. First of all details on the test environment will be depicted. This section studies the network topology and shortly discusses the Jepsen framework. Subsequently the second section will review the bank and index workloads and explains how they are defined in the framework. Finally fault injection techniques and methods are addressed.

3.1 Test environment

The test environment consists of a cluster of five interconnected nodes and a single master node, all running on their own virtual machine (see Figure 3.1). For the remaining of this project the nodes will be referred to as N1, N2, N3, N4 and N5 and the master node will simply be named master. The job of the master is to execute the individual tests. This involves configuring the cluster, opening the client connection, executing the queries and collecting the results.

Both the nodes and the master run on a VM (virtual machine). Several reasons motivate the use of virtual machines. Firstly, there are numerous companies offering low-priced VM instances, including Amazon [1], Microsoft Azure [38], Google Cloud Engine [25] and DigitalOcean [11]. Secondly, installing and configuring six physical machines would be more time consuming and more expensive. Last but not least, VM infrastructures allow for easy scaling. New instances are configured in a matter of minutes. Accordingly, additional master and/or node instances can be instantiated with ease, allowing for extra experiments if desired.

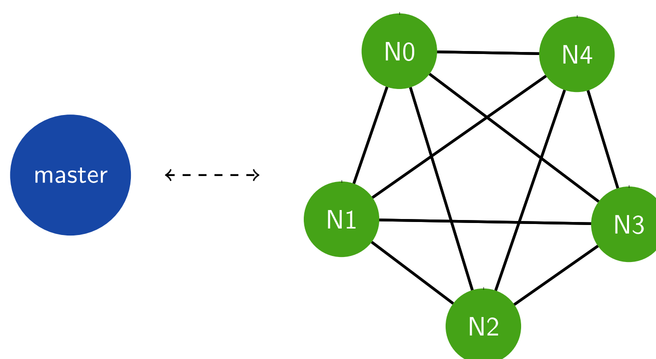


Figure 3.1: Master coordinating five nodes. Connections between the nodes are not shown.

Both the nodes and the master run on a Linux environment, specifically the Ubuntu 15.10 (Wily Werewolf) distribution. Although this distribution is used, the framework is configurable for other distributions as well. This project uses the Cloud infrastructure of the Google Cloud Engine [25]. As Cockroach Labs provided credit for the Google Cloud Engine, this infrastructure is used for this project.

As mentioned before, the experiments will be powered by the Jepsen framework [32]. Specifically, a branch of the test framework is used which Cockroach Labs used in their previous tests. This branch will be extended by adding specific workloads that will be reviewed later in this chapter. The Jepsen framework is written in the LISP programming language Clojure [29] and houses a sizable amount of useful functions for running experiments. These functions include logging data, communicating over SSH, plotting results, controlling nodes and generating random events.

3.2 Workloads

To test the database engine we devised two tests, a banking test and an index test. Both simulate different workloads under which the database is expected to maintain consistency. Workloads are a set of generated events to which the client responds, i.e. a bank transfer or a read request. A test definition defines how and when events are emitted, which is mostly done in a random fashion. A coded example of such a workload definition is shown below:

```
(->> (gen/mix [bank-read bank-diff-transfer])
     (gen/clients)
     (gen/stagger 1)
     (cln/with-nemesis (:generator nemesis)))
(gen/clients (gen/once bank-read)))
```

This example defines a mix of read and transfer events of a bank. These events are emitted to all the clients (nodes) and are uniformly randomly generated every $0 > x > 1$ seconds, and at the end of the test one final read event is emitted. When fully executed, a checker analyses the history of the test and verifies whether all results are valid according to a set of predefined constraints. If all these constraints are met, the test is marked as valid.

3.2.1 Banking test

A bank is a good example of a situation in which we take for granted that it works according to our rules of finance. For example, it is not possible to transfer money to another person if we do not have the sufficient balance. Also, we trust our bank to preserve our correct balance without ever losing track. Imagine the catastrophes that might occur if this went terribly wrong.

It might seem logical that this all works out well, yet it is anything but straightforward. Records could get manipulated, damaged or even lost. Moreover, if not implemented correctly, account transfers might lead to inconsistencies. Consider the following scenario:

1. Account X has a balance of €20.
2. Transaction 1 verifies whether account X has enough balance to transfer €12 from account X to account Y, and states this transfer is possible.
3. Concurrently, transaction 2 verifies whether account X has enough balance to transfer €18 from account X to account Z, and also states this transfer is possible.
4. Transaction 1 executes the transfer and moves €12 from account X to account Y.
5. When transaction 2 continues to execute its transfer, either one of two situations may occur if snapshot isolation isn't implemented properly:
   (a) transaction 2 subtracts €18 from the balance, resulting in a negative balance of €-10.
   (b) transaction 2 ignores the balance updated by transaction 1 and overwrites the balance with its previously known value of €20 minus €18, thereby creating extra money.

Data corruption might influence the consistency of a database system, and therefore this section will describe the tests devised to check the database system under fault injection.

Table 3.1: The most simplistic structure for storing balances of accounts.

account	balance
A	90
B	155

Table 3.2: Transactions represented by tuples in the format $(\Delta, Balance)$.

timestamp	A	B
T0	(0,10)	(0,15)
T1	(-5,10)	(5,15)
T2	(15,5)	(-15,20)

Table 3.3: The structure used for storing transactions during the bank test. For the multi table method the account column is redundant as each account owns its own table.

	account	balance	Δ
T0	A	10	0
T0	B	15	0
T1	A	10	-5
T1	B	15	5
T2	A	5	15
T2	B	20	-15

For a start, the balance of every account has to be stored. This can be achieved using several data structures. The most straightforward structure would be storing the balances plainly in one table, each account owning its own row in the table (see Table 3.1). However, this approach has its disadvantages. Would anything go wrong there is no transaction history available, and the balances are easy to tamper with.

Therefore we choose a more intricate structure. Every money transfer has a fixed set of fields, including the amount of money that is transferred, the source and target accounts, and likely time and date values. Table 3.2 shows a simplified representation of transactions using tuples, in which each tuple stores a value $(\Delta, Balance)$. In this example account A owns €10 and account B owns €15 at T0. At T1 account A has the value (-5,5), indicating that €5 is subtracted from its balance, resulting in a balance of $10 - 5 =$ €5. At the same time, B has the value (5,20) indicating that 5 is added to its balance, resulting in a balance of $15 + 5 =$ €20. For this to be implemented a table is created with the columns *timestamp*, *account*, *balance* and Δ, shown in Table 3.3. From this example we can see that the current balance is calculated by calculating $balance + \Delta$.

This table still stores all the accounts in a single table, and therefore we refer to it as the single table method. Alternatively a multi table method is devised. With the multi table method each account is stored in its own table. CockroachDB internally uses key ranges to distribute its data among nodes [8]. Key ranges are always split at table level, so by storing each account in its own table the transactions are more spread among the nodes. Due to the fact that CockroachDB distributes the tables of the multi table method differently, this method is added as a separate test.

As described earlier the checker at the end of the test will verify whether the test is considered valid. Tests are considered valid if all constraints are met. For the bank test the following rules are defined. If one of these three constraints is broken, the test is considered invalid.

Delta rule At any moment in time for each individual transaction the equation $balance + \Delta \geq 0$ should hold.

Balance rule At any moment in time the total balance should remain constant (no withdrawals or deposits are possible during the tests).

Transaction history rule For every account, the last known balance should be equal to the balance calculated using the transaction history.

The workload is defined by three distinct events: read, transfer and delete. Delete events are added to reduce the amount of information to be processed by the framework. As tests run for

5 minutes, they become quite data intensive. Therefore, delete events are triggered by the test, deleting all but the last three records. This still allows the framework to run the tests on all read events with the last three transactions for every account.

All three are fired randomly during the test every $0 > x > 0.2$ seconds. An overview of the events fired during the tests, and their frequencies, are shown in Tables 6.4, 6.6 and 6.7.

To summarise, the database is expected to adhere to the three rules defined above. Fault injections may lead to events failing to execute, but may never lead to inconsistencies in bank transactions.

3.2.2 Index corruption

In order to illustrate how and why indexes are used by database systems, this section will use a music database as an example. Imagine this database stores a table with artists, as shown in Table 3.4. In this table the birth name, artist name, birth year and place of birth are stored. As it is a database, we can query it for specific data. We could for example query the table for all artist that have an artist name starting with the letter D:

```
SELECT artist_name FROM artists
    WHERE artist_name LIKE 'D%';
```

This will result in one row, namely the row of the artist "David Bowie". For small tables this query will be fast. However, if the tables contain millions of rows the query will become slower. This is especially the case if queries become significantly more complex than this sample query. SQL databases often provide extra queries that can be used to analyse efficiency, and so does CockroachDB:

```
EXPLAIN SELECT artist_name FROM artists
    WHERE artist_name LIKE 'D%';
```

Explain queries give more information on how the query is executed by the database system. In this case, the system will return:

```
+-------+------+-------------------+
| Level | Type |    Description    |
+-------+------+-------------------+
|     0 | scan | artists@primary - |
+-------+------+-------------------+
```

The type "scan" in combination with a "-" signifies that CockroachDB will use unbounded range for this query. When the range is unbounded, the whole table will be scanned sequentially, which is both inefficient and time consuming for large tables [9].

This is where indexes come into play. Indexes are frequently used in databases for optimising queries. By storing key/value pairs in a (binary) search tree a database system can quickly locate data. We can instruct CockroachDB to create an index on the artist table by executing the following query:

```
CREATE INDEX ArtistNameIdx ON artists (artist_name);
```

As a result, the EXPLAIN query will return a different response:

Table 3.4: Artist table in music database.

name	**artist_name**	**birth_year**	**place_of_birth**
Prince Rogers Nelson	Prince	1958	Minneapolis
David Robert Jones	David Bowie	1947	London
Farrokh Bulsara	Freddie Mercury	1946	Stone Town

```
+-------+------+---------------------------------+
| Level | Type |           Description           |
+-------+------+---------------------------------+
|     0 | scan | artists@ArtistNameIdx /"D"-/"E" |
+-------+------+---------------------------------+
```

This response indicates the search space is reduced, as it is now bounded to all artists with an artist name greater than D and smaller than E. This reduces the overall response time on such a query [21][50].

Additionally, indexes are used for sorting data. Indexes are generally stored in a tree structure. If the data is requested in order, it is just a matter of traversing the tree to retrieve the results in an ordered fashion. A simplified version of how such a tree could be represented is illustrated in Figure 3.2a. In this case, an index is placed on the column storing the year of birth. Would the tree increase in size, it is obvious how the use of such a search tree will speed up queries in contrast to scanning all the values in a table. Also, when doing a in-order traversal the records will return ordered.

However, just as all other data, indexes can experience corruption. Multiple scenarios are plausible. Firstly, one of the pointers can be manipulated. An example is shown in Figure 3.2b. In this case, the 1947 pointer is corrupted, referring to a wrong record in the database. Would we request all artists born in 1947 we would get “Prince Rogers Nelson” as a result. Moreover, retrieving all data from the table would result in “Prince Rogers Nelson” being returned twice. Secondly, the structure of the tree could get damaged, essentially resulting in a wrong order of records (see Figure 3.2c). Finally, whole records could get lost. This is illustrated in Figure 3.2d. As a matter of fact, there are even more situations in which a index could corrupt. To summarise, index could just as well corrupt as other data structures. That is why a workload is devised to evaluate the influences of fault injections on indexes.

For this test a monotonic client will be used. A monotonic function is a function that remains in order over time. A similar test is earlier defined by Cockroach Labs and will be extended to cope with indexes [48]. The tests uses a simple table containing one value column, as shown in Table 3.5.

The test defines events with the following code sample:

```
:generator   (gen/phases
  (->> (range)
    (map (partial array-map
      :type :invoke
      :f :add
      :value))
    gen/seq
    (gen/stagger 1)
    (cln/with-nemesis (:generator nemesis)))
  (->> {:type :invoke, :f :read-withindex, :value nil}
    gen/once
    gen/clients)
  (->> {:type :invoke, :f :read, :value nil}
    gen/once
    gen/clients))
```

This can be explained as follows: every $0 > x > 1$ seconds an add event will be emitted by the framework. Each add event will firstly request the database for the maximum known value in the table, increment the value by one, and insert this value to the table. For this the following query is used:

```sql
SELECT MAX(val) FROM mono;
```

In the end we expect each value to be unique. This tests will be executed both with and without indexes placed on the table, as an index could also have an influence on add events of the test. At the end of the test, two read events will be emitted. One event makes use of the

val
0
1
2

Table 3.5: Monotonic client table.

index, the other will perform the same query without the index. This difference in index vs. non-index is achieved by retrieving an additional "bogus" column for the non-index which is not included in the index. These read events will retrieve all the values from the database in an ordered fashion. If everything works as it should, it should return an incrementing list of values, where no values are missing and everything is in order. If duplicates are detected, this could also suggest an index corruption, as the "MAX(val)" also uses an index look up.

The checker at the end of the tests verifies the following constraints:

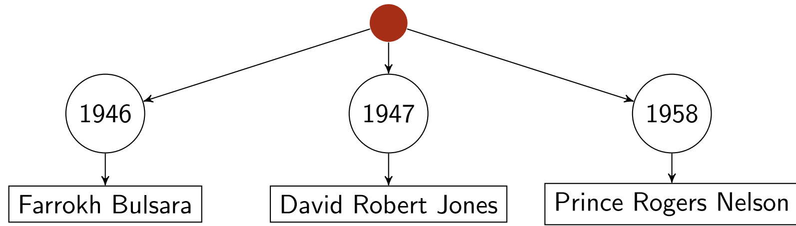
Duplicates The resulting read should not contain any duplicates.

Lost Lost records are those we definitely added but were not read.

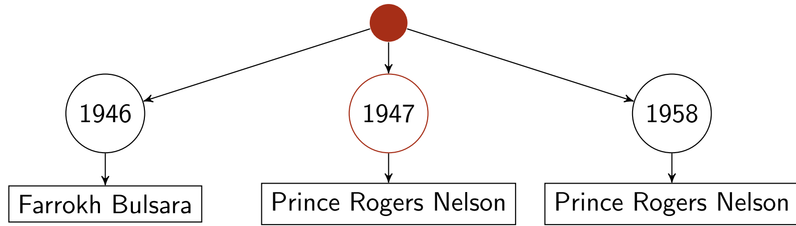
Revived Revived records are those we failed to add but were read.

Recovered Recovered records are those we were not sure about and that were read.

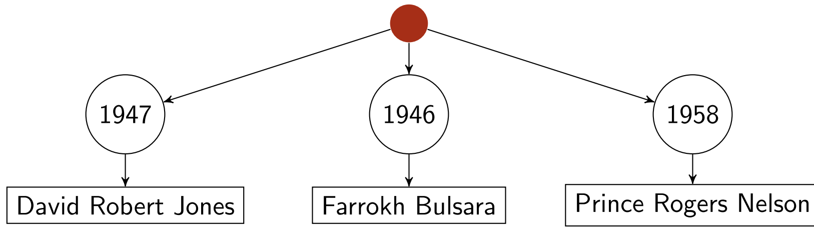
Reorders Reordered records are those not retrieved in order.



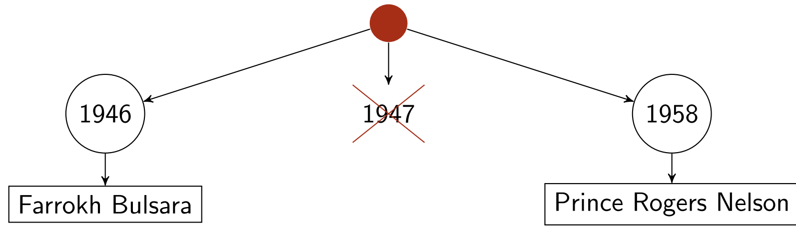
(a) Index tree under normal conditions.



(b) Index tree with corrupt pointer. The corrupt pointer refers to the wrong data.



(c) Index tree with wrong order.



(d) Index tree with missing data.

Figure 3.2: Collection of index trees for artist table with different possible failures.

3.3 Fault injection

We use fault injection to simulate silent data corruption. This is done by flipping bits inside the files used by the database engine. Faults need to be injected into the engine's file system. For this a Python script is created that will flip bits in designated files. This script accepts two parameters: the number of bits to flip, and the file location.

Firstly, the files used by the database engine must be identified. This is done with the help of lsof, a Linux command that lists open files. When provided with a PID (process identifier) the lsof command returns all the open files for that particular process. By passing the PID of the database process we are able to get the files the database engine is using at that particular moment, and feed those file to the fault injection script. Not all files returned by lsof are useful, so certain files are filtered. These files are files like the 'COCKROACHDB_VERSION' file, which does not have any influence on the functioning of the engine.

Next, the fault injection script has to come into action. As mentioned before the script accepts the location of a file to be injected as a parameter. Furthermore the script also accepts a second parameter which indicates how many bits have to be flipped inside that particular file. We use various amounts of injections per experiment: 1, 2, 50 and 1000. An overview of all the files injected by the script in both experiments is shown in Table 6.1. These files are of different types, all with their own function.

Firstly there are SST (Static Sorted Table File) files. These are files used by RocksDB to store the table data [14]. Secondly there are option files. Option files specify options used by RocksDB, such as the buffer size and allocation sizes [16]. Thirdly there are manifest files that help recover RocksDB in the event of a system failure. As file systems are not atomic, all transactions executed by RocksDB are stored in a transactional log as a manifest file. When the operating system or the database system crashes, this manifest log is used to restore the database to its last known consistent state [15]. Finally there are log files, referred to as WAL (Write Ahead Log) files. WAL files contain a serialised version of the in-memory table RocksDB is using [17]. This WAL file is also used to recover the database to a consistent state.

When a file is chosen, the script will inject n faults using the following function:

```
def insert_bit_flips(file_path, n_flips):
  f = io.open(file_path, 'rb+')
  file_end = f.seek(0, os.SEEK_END) - 1
  for _ in range(n_flips):
    # Find and read a random byte in the file
    random_pos = random.randint(0, file_end)
    f.seek(random_pos)
    f_data = f.read(1)

    # XOR this byte with a random number 0 <= x <= 7
    n_data = xor(ord(f_data), 2 ** random.randint(0, 8 - 1))

    # Write the modified bit
    f.seek(random_pos)
    f.write(chr(n_data))
    f.close()
```

For each iteration this function seeks a random byte in the file, performs a XOR operation and writes the result. One example of such an XOR operation is shown below, where one bit is flipped:

```
1011 0010
0000 0010
------------- XOR
1011 0000
```

What is reported back by the script (and logged in the experiment log) is shown below. Where the counter indicates this is the nth injection performed so far. The script is distributed to all nodes and is called from the master node.

```
{
  :file_path "/home/maxgrim/cockroach-data/000003.log",
  :file_bits 4608,
  :injected_bits 1,
  :ratio 2.17013888889E-4,
  :counter 4
}
```

Fault injection events are called nemesis events and reported in Tables 6.3, 6.4, 6.8 and 6.9 as "info start". One such an event calls the Python script and is mixed with the other events in the table. In other words, a test with 50 fault injections does not mean that 50 faults are injected once, but every time the "info start" event is triggered, which occurs just as often as the other events in the test.

CHAPTER 4

Results

This chapter will describe the results. In total 976 tests are executed. The experiments are time limited by 300 seconds. The total test duration averaged to 306 seconds. The 6 seconds overhead is caused by initialising the experiments and collecting the results.

4.1 Banking test

Tables 4.1 and 4.2 show the number of iterations done for both the single table method and multi table method banking tests. The number of iterations varies per test due to several reasons. Firstly, if the connection from the master to one or multiple nodes is lost, it is impossible for the master to download the results from the nodes. This may happen when the OS crashes, or when other events happen that break this connection. OS crashes were never the focus of this research and are therefore not measured, so exact reasons as to why the connection is lost are unknown. In the framework logs several errors are reported, including closed sockets, closed streams and connection resets. Secondly, it seems some SSH packets get corrupted. The framework is not able to cope with these errors and aborts the connection. Finally, in some cases a deadlock seems to occur. In this case the framework seems to the framework sometimes seems to run into a deadlock. The reasons as to why this happened are not investigated, and solved by killing the test after 1200 seconds, which four times the average time needed to complete a test.

Considering the test result without fault injections we can affirm the database system operates as it should. This supports the tests previously performed by the developers at Cockroach Labs [48]. Tables 4.1 and 4.2 mark zero invalid tests without fault injections. Moreover, panics measured during the tests were not existent for these tests (Figure 4.4). Panics are cases where the database system on a node crashes.

These tests seem valid, yet transfers still fail. Tables 6.3 and 6.4 show failure rates of 11.1% for the single table method and 13.5% for the multi table method. This is also illustrated in Figures 4.1 and 4.2a. Transfers are randomly generated by the framework. In the case a transfer would result in a negative balance, the transfer is also marked as a failure. While marked as failure, we are not interested in these types of failures. Inspecting Figure 4.3 confirms 100% of

injections	tests	total length (s)	invalid
0	59	17999	0
1	27	8230	0
2	22	6703	0
50	14	4270	0
1000	31	9888	0

Table 4.1: Bank test with the single table method.

injections	tests	total length (s)	invalid
0	62	18904	0
1	45	13743	0
2	25	7635	1
50	14	4693	0
1000	33	10120	1

Table 4.2: Bank test with the multi table method.

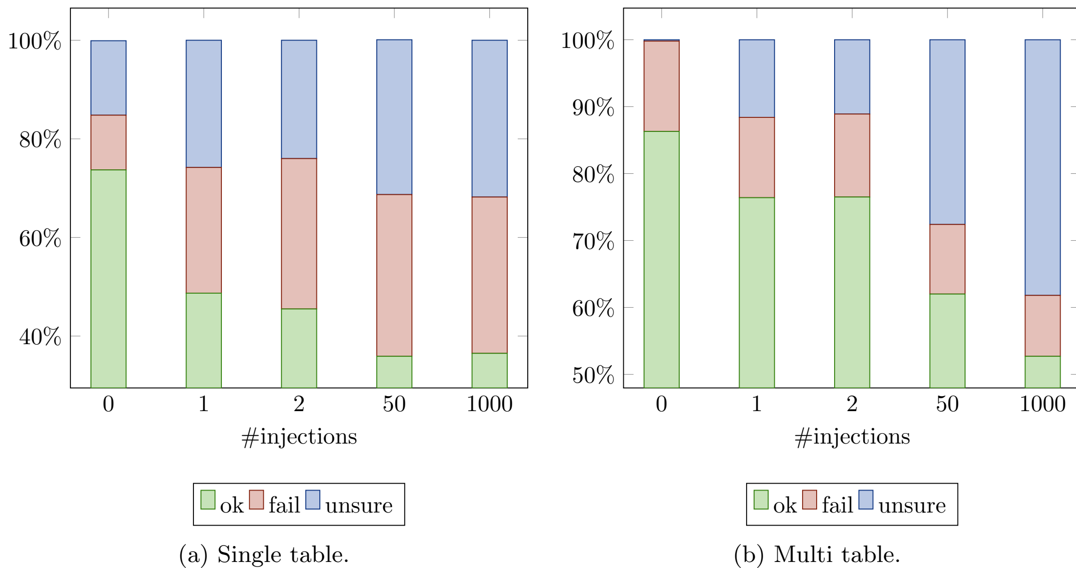


Figure 4.1: Transfer events.

the transfer failure rates for both single- and multi table methods are caused by transactions that would otherwise result in a negative balance. Therefore the tests without fault injections are considered successful.

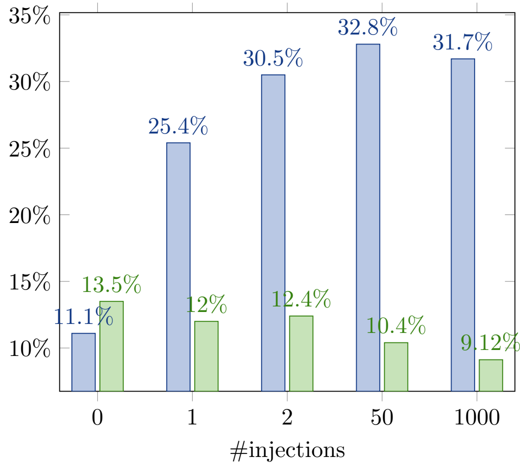
Interestingly, when looking at the failures rates for multi table tests with fault injections, we do not see an increase in failed transfers (Figure 4.1b and Table 6.4). In contrast, the transfer failure rate for the single table method does seem to increase initially (Figure 4.1a and Table 6.3). Furthermore, we observe a shift in the reason as to why these transfers fail. Figure 4.3 shows increasing reports of checksum errors or even database losses under fault injections. Where with zero injections all failures are caused by a resulting negative balance, checksum failures are present in all other experiments. It is unclear why the number of checksum failures is particularly high with 50 injections. We also see the changes of a lost database increase with the number of injections. In some cases CockroachDB reports that either the "system" or "jepsen" (which is the test database) do not exist. This indicates the database and/or the pointers to it are that corrupted that the engine can not find them anymore. We would expect some sort of corruption warning here as well, as this is not apparent.

Additionally the number of unsure transfers increases as more faults are injected. Transfers are marked unsure if no apparent error is returned by the server. There are various reasons for unsure transfers, including timeout, a closed connection or an I/O error. Timeouts that do occur can be caused by the server being confused or even crashed under fault injections. Measured over all banking tests the reasons of unsure transactions are shown in Table 6.5.

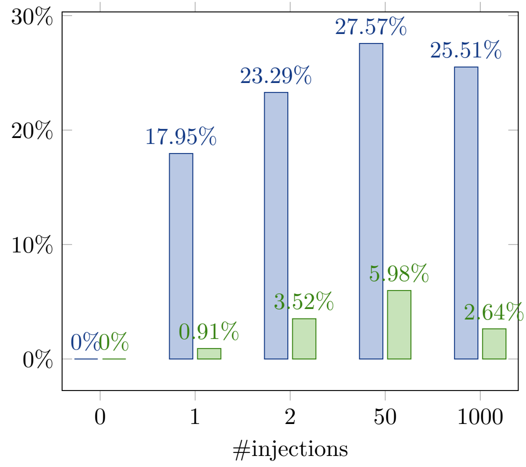
Not only transfer but also read events fail. Read events request the transactions for every account from the database. While without fault injections 0% fails, this increases with the number of fault injections for both single table and multi table methods (see Figure 4.2b). In the case a read event fails, we can not determine the balance of an account anymore. Clearly these failure rates are significantly higher for the single table method. In other words, the read reliability is higher for tests using the multi table method.

To begin with, it is positive that CockroachDB does prevent returning corrupt data by reporting a checksum mismatch. However, this means that the client is unable to determine the balance. Consequently the application has "lost" the balance for the account. By default CockroachDB stores three replicas of its data distributed over nodes. In the event all three replicas are damaged it would be unavoidable for the database to not return any data. Yet, when one or two replicas are damaged the database should still be able to return the balance using the unharmed replica. However, read failures still occur relatively often when only injecting into N3 (see Tables 6.7 and 6.6). This indicates CockroachDB does not perform this kind of repairs yet.

When considering errors they can be divided into two types: server errors and client errors.

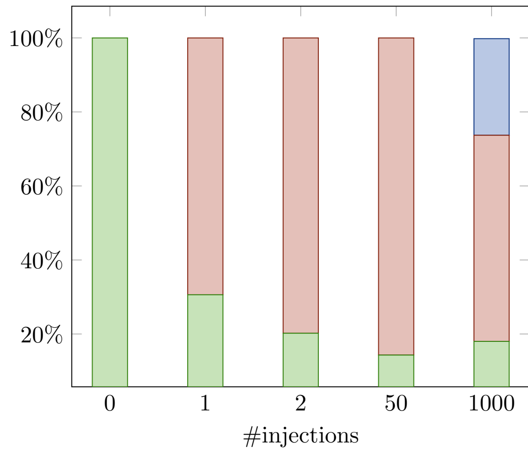


(a) Transfer failure rates.

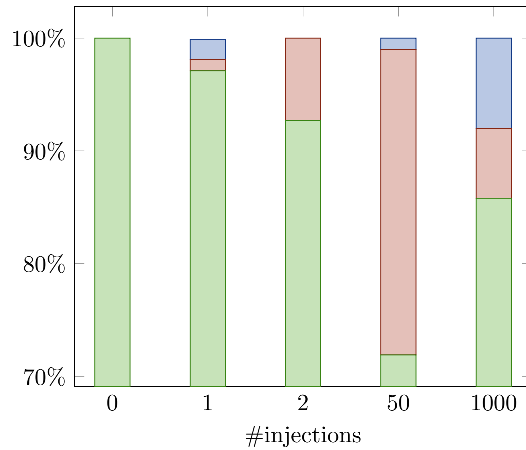


(b) Read failure rates.

Figure 4.2: Transfer and read failure rates for bank tests using the single- and multi table method.



(a) Single table.



(b) Multi table.

Figure 4.3: Bank test transfer failures categorised by reason.

injections	tests	total test length (s)	invalid
1	43	13116	0
2	41	12507	0
50	44	13415	0
1000	41	12497	0

Table 4.3: Bank test with the single table method, where faults are only injected on N3.

injections	tests	total length (s)	invalid
1	42	12808	0
2	41	12501	0
50	40	12193	0
1000	42	12800	0

Table 4.4: Bank test with the multi table method, where faults are only detected on N3.

Server errors are written in an error file by the database system on the nodes. These logs are collected at the end of the experiment for analysis. Examples of server errors are checksum mismatches during internal replication or other forms of communication not visible by the client. On the other hand, client errors are collected by the framework during the execution of queries. Examples of client errors are the failed read and transfer events mentioned before. Server errors and client errors are visualised in Figure 4.5. Client errors generally occur more frequent than server errors. Furthermore client errors tend to increase with the amount of fault injections except with 1000 fault injections. As 1000 fault injections is huge amount, probably other errors arise earlier, all in all lowering the amount of client errors.

As noted before, the bank tests are also executed with fault injections only performed on node N3. Tables 4.3 and 4.4 show these tests had zero invalid tests. The number of iterations for each test varies less compared to tests injecting on all nodes. This indicates these tests are more stable. Moreover, on average less server- and client errors occur in tests where faults are only injected in node N3 (Figure 4.5).

4.1.1 Invalid tests

Two tests were marked as failed by the checker. Both these tests used the multi table method, and during the test faults were injected on all nodes (see Table 4.2). To put it in numbers, 0.3% of the total 666 tests returned a failure. If we only consider tests using the multi table method, the invalid percentage rises to 1.12%. The results of both these experiments are shown in sections 6.1 and 6.2.

As stated before, a test is considered invalid in case one of the constraints is not met. In the first invalid test, shown in section 6.1, the total balance is not constant over time. While the balance starts at €225, the final balance is €221. So €4 is lost in the process, while the delta check seems to be fine. Over time, the total balance takes the values €225, €229, €221, and €230. Note that for this test the transaction records were not logged, as this is only added during later executed tests. Because of these missing records we can only speculate as to why this total balance varies. Nonetheless we can conclude that these balance inconsistencies are caused by the data corruption and should not be visible to the client.

The second failed tests however did log the transactional records during the test. These results are shown in section 6.2. Same as with the first failed test the balance was not constant over time. In this case, the starting balance of €225 spikes to a whopping €9007199254741217. This is where probably a high order bit in the 64-bit integer is flipped without the database noticing. We can exemplify this with the help of the transaction details shown in section 6.2.1. Using the tuple representation introduced in Chapter 3 we can show the last three transactions performed on account 0:

```
T68     (2, 22)                = 24
T79     (-5, 24)               = 19
T114    (-4, 9007199254741011) = 9007199254741007
```

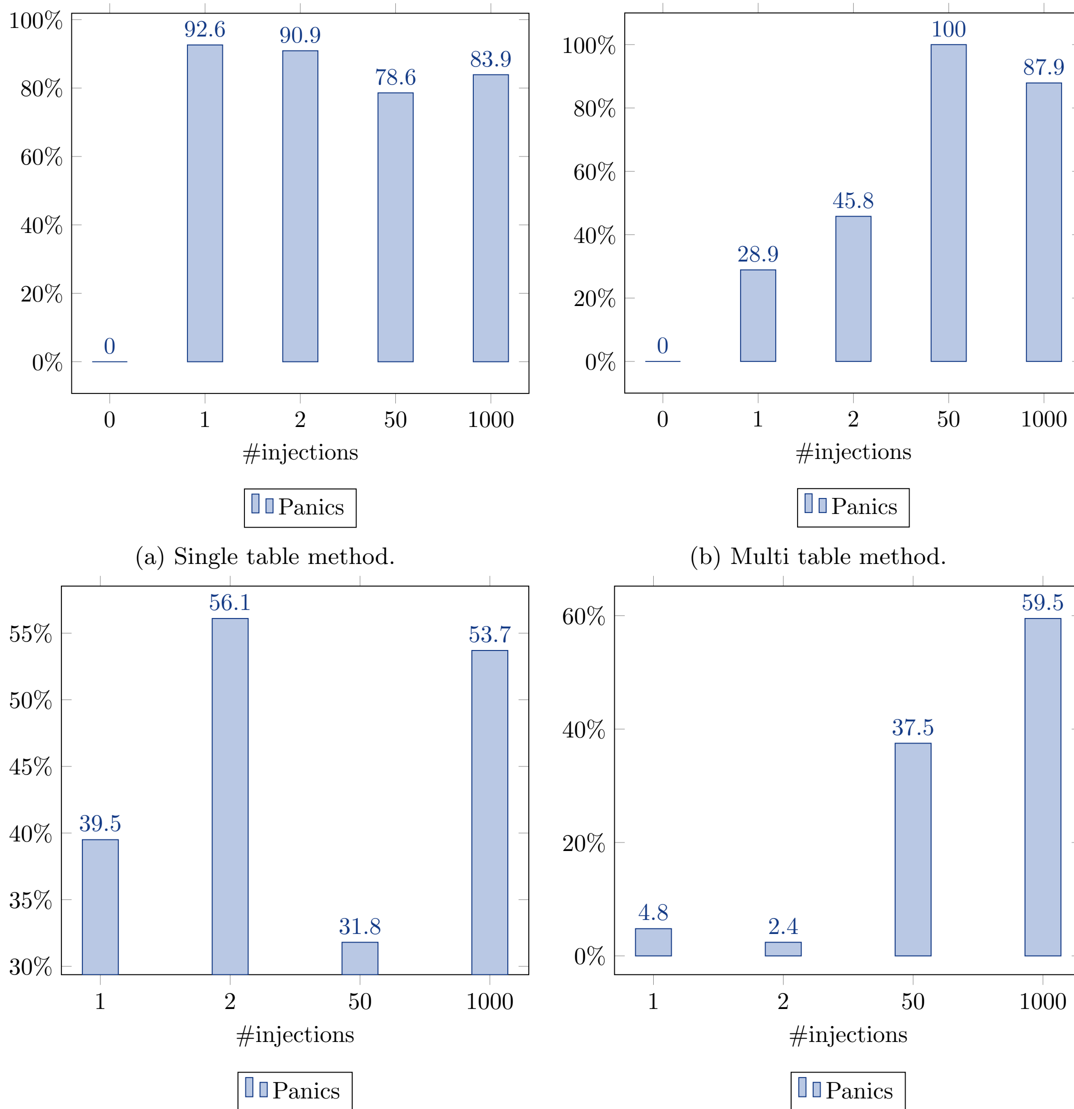
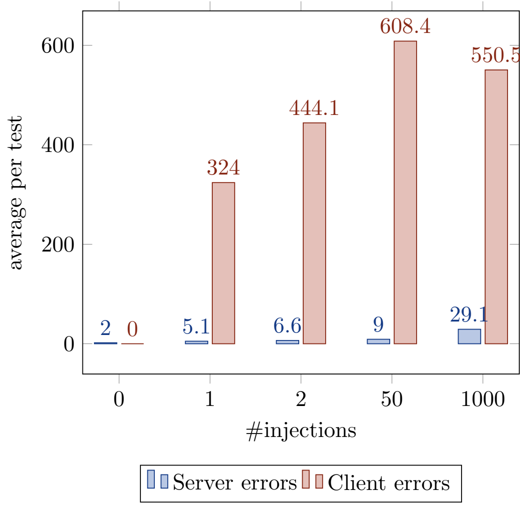
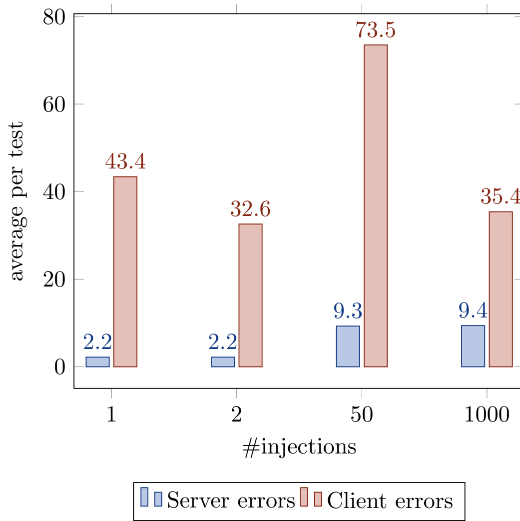


Figure 4.4: Panics measured during bank test experiments. All nodes have the possibility to panic. In these measurements, if one of the five nodes crashes with a panic, this is counted as one panic for that particular test. Next, all panics for all experiments are counted and normalised, resulting in the graph above. For example, in the case of 50 injections there is a panic rate of 100%. This does not necessarily means that all nodes have paniced, but that in every experiment at least one node has paniced.



(a) Single table method.

(b) Multi table method.



(c) Single table method, injections only performed on N3.

(d) Multi table method, injections only performed on N3.

Figure 4.5: Server and client errors encountered during the bank tests.

This implies somewhere between T79 and T114 a data corruption modified the balance field. The enormous increase in the balance value can be justified by flipping a high order bit in the 64-bit signed integer value stored in the database system:

```
19 =
    ...0000000000000000000000000000000000000000000000000000010011
9007199254741011 =
    ...0000100000000000000000000000000000000000000000000000010011
```

After this corruption has taken place the balance restores to its original value of €225 at the following read, as can be seen in section 6.2. However, even further down the line, the balance seems to suffer from a second corruption, this time rising the total balance to €2097377 twice in time.

One of the possible explanations for the first balance recovery is that the corrupted node containing the high value €9007199254741217 crashes, with another node thereafter taking action into showing the total balance, restoring it back to its original value of €225.

4.2 Index test

The number of index tests using the monotonic function is shown in tables 4.5 and 4.6. The number of tests varies less than with banking tests, suggesting the tests are more stable. Furthermore the amount of invalid tests is much higher compared to the bank tests.

As described in the methodology section the index test only emits two read events at the end of the test, one with the index used and one without. Would this final read for some reason fail, the checker can not verify whether the test is valid. This explains why the invalid rate is much higher for these tests. Tables 6.8 and 6.9 do confirm this, as read failures match the number of invalid tests. In a few cases this does not hold, these are discussed later in this section.

The reasons as to why reads fail are shown in Figures 4.9 and 4.10. This indicates there is a higher chance of checksum mismatches when indexes are not in place.

Considering the test result without fault injections we can for this test also conclude the database system operates as it should. There are no panics detected (see Figure 4.6). Furthermore no client errors were observed (see Figure 4.7 and Tables 6.8 / 6.9). On average two server errors were monitored during the tests. On closer inspection these errors should be interpreted as warnings instead of errors, and were not relevant to our research.

During the tests one case is detected where there are inconsistencies between the read using an index and the read not using an index, specifically the value of “fail read-withindex” in Table 6.8 is equal 12 where “fail read” is equal to 13. This indicates that the read using the index did succeed while the one without failed. What is interesting though is another type of error, found in an index vs. no index test. One test executed with 50 bit flips indicated a failure of the non-index read, whilst the index read did succeed. This indicates the table data is corrupted, but when the index is used the data is returned fine. You could turn this around and say, when the index is corrupt, the data used by the client coming form the index might thereafter corrupt the actual table data, which is obviously bad.

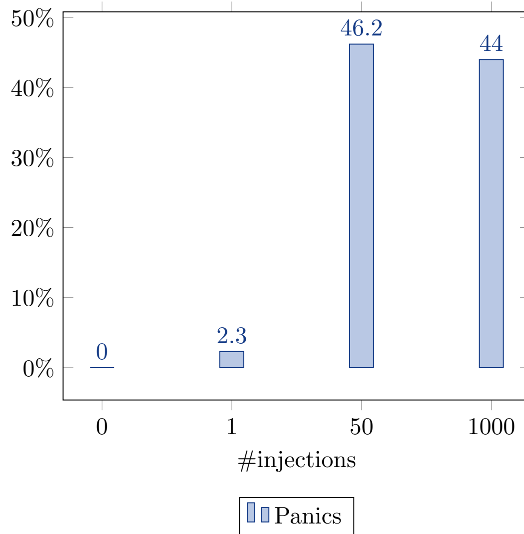
Figure 4.8 indicates the number of adds is relatively lower compared to bank tests. This is probably due to the fact that less data is used for add operations, lowering the odds of encountering corrupted data. Just as with the bank tests also the number of unsure responses increases with the number of fault injections.

injections	tests	total length (s)	invalid
0	47	14285	0
1	43	13084	30
50	26	7917	21
1000	25	7612	16

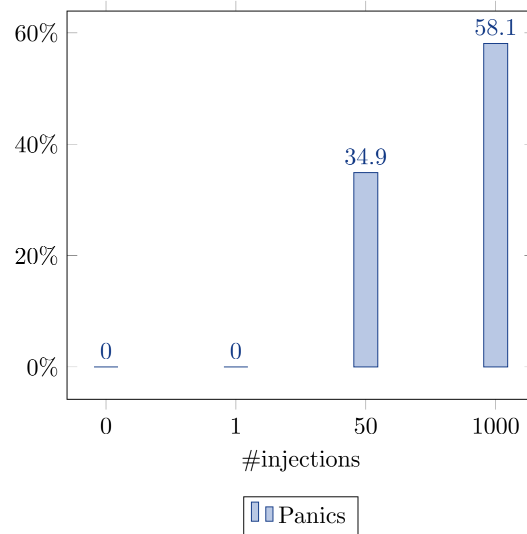
Table 4.5: Monotonic tests with an index in place.

injections	tests	total length (s)	invalid
0	51	15669	0
1	44	13532	4
50	43	13280	14
1000	31	9552	14

Table 4.6: Monotonic tests without an index in place.



(a) Tests executed without an index in place.



(b) Test executed with an index in place.

Figure 4.6: Panics measured during index corruption experiments. All nodes have the possibility to panic. In these measurements, if one of the five nodes crashes with a panic, this is counted as one panic for that particular test. Next, all panics for all experiments are counted and normalised, resulting in the graph above.

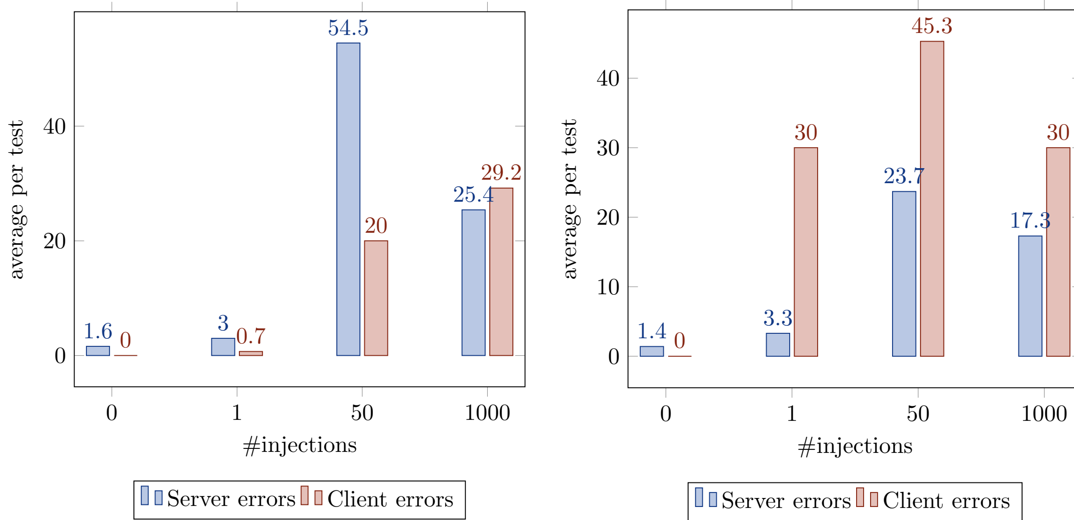
4.2.1 Invalid tests

In this phase we found an error in the framework. The results indicated “revived” errors, while they actually were false positives. Take for example this output, taken from the monotonic test without indexes with 50 bit flips:

```
232843795233 7 :invoke :add 1089
232848348646 7 :fail :add 1089
    PSQLException: ERROR: database "system" does not exist
237094759706 9 :ok :add (1089 1464913460344570640000000000N 1)
272108902530 13 :invoke :add 1291
```

Values are marked as revived by the framework if the value failed to add yet later were read by the client. In the example output above, the add is firstly marked as failed, but later still marked as successful. This makes the framework believe the value was not added, while it actually was.

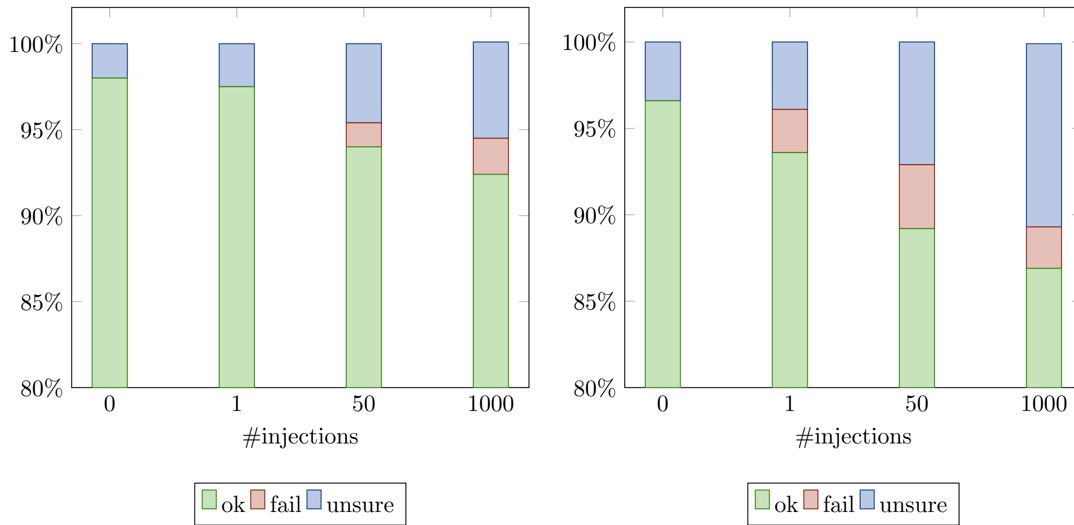
Furthermore, no faults were found with the reorder test, implying that none of the data corruptions led to a wrong order in the tree traversals.



(a) Tests executed without an index in place.

(b) Tests executed with an index in place.

Figure 4.7: Server and client errors encountered during the index corruption tests.



(a) Without an index in place.

(b) With an index in place.

Figure 4.8: Add events.

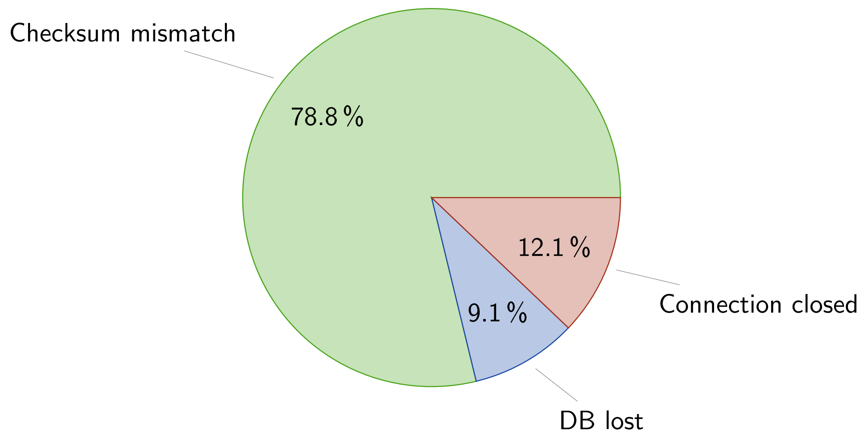


Figure 4.9: Reason for failed reads without indexes in place.

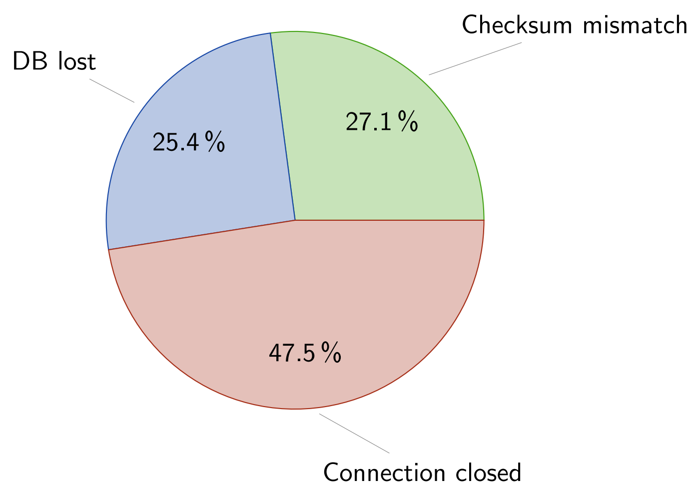


Figure 4.10: Reason for failed reads with indexes in place.

CHAPTER 5

Conclusion

We as humans collect more data every year. Databases are often used for storing important data, and they rely on storage systems functioning properly. However, data corruptions do occur and are caused by a variety of reasons [2][23]. Storage systems are expected to report errors in the event data corruptions occur, but not all corruptions are detected. Lot of effort has been put into preventing these problems, and because of these efforts the chances of data corruptions have been reduced [33]. Yet, it is still possible for these silent data corruptions to occur, and there is little research on the effects of these silent data corruptions on database systems.

Databases promise to be robust, highly consistent, fault-tolerant, survivable and durable. Although this sounds promising, these claims are all based upon the assumption that storage systems are fault free. Scientific research of these claims in combination with (simulated) data corruptions is sparse. In the hope of setting a stepping stone towards new research on database robustness, this project extended an existing database testing framework called Jepsen with functionality simulating silent data corruptions. The database system tested by this project is CockroachDB, though with some effort it is possible to use the framework for other databases as well. CockroachDB is an open-source lock-free distributed SQL database and currently in beta stage.

To test the distributed database a test environment is used consisting of five node instances and a master instance. The five nodes are used for a CockroachDB cluster whereas the the master node is used for controlling the experiments. Two workloads are devised, one simulating money transfers inside a bank, and another simulating a monotonic function. The framework runs these experiments with different parameters and analyses the consequences of simulated data corruptions.

Analysing these results we can confirm that, as promised, under normal conditions the CockroachDB engine functions properly. During the tests executed no inconsistencies were encountered, the system did not panic, and no client errors were reported.

Moreover, CockroachDB does detect corruptions in its file system with the help of checksum techniques. Under fault injection the client often reported checksum mismatches, which prevent the client from observing invalid data. Results indicated significant difference between the single- and multi table methods used for the bank test, indicating users should take their data structures into consideration when designing an application.

Panics were observed relatively often during the experiments, but because the rate of simulated fault injections in these experiments was relatively high, we can not blame the database engine to crash.

However, we observed two cases where the tests resulted in serious inconsistencies. In both tests the sum of all balances varied through the experiment while this should remain constant at all times. Where in one experiment €4 was lost, the other experiment showed an enormous temporary increase in the balance of one account.

Additionally, the test result indicated that CockroachDB does not use its stored replicas for recovery in case of an detected data corruption. This is unfortunate, as this implies corrupted data is lost inevitably, while the data might still be valid in the stored replicas. CockroachDB

uses replication for availability but not yet for error recovery.

The work in this thesis was performed on a database system still in beta stage, and it is therefore remarkable that this database engine was significantly reliable in the presence of silent data corruptions despite its relatively young age. Meanwhile, the methodology presented in this thesis is relatively independent from the specific database engine as it is based on SQL, and could thus readily be extended to other relational database management systems.

5.1 Discussion

The experiments done in this project inject many faults simultaneously, way more than usually would occur in natural situations. Therefore we can not blame the database engine to crash relatively often under these conditions, as in ordinary situations this rate of data corruption would not occur. Nevertheless we have chosen to do so, as flipping bits in the order of one bit every week would be too slow for testing. If we'd chosen a lower rate of errors the odds of encountering an error lower drastically. Therefore the simulated fault injections have been speed up to quickly produce results. Despite the high rate of fault injections, this still illustrates that a bit flip at the right time and place could lead to severe inconsistencies in the database. Therefore database developers should not blindly trust the data received from the storage system and verify all data received.

5.2 Future work

To start with, this framework can be extended to test other database engines as well. In these tests additional workloads could be defined that test other interesting properties of the engine. Furthermore, steps could be taken into analysing the probability distribution of database system inconsistencies under increasing simulated fault injection, rather than the deterministic approach.

CHAPTER 6

Appendix

6.1 Failed test 1

```
{:perf {:valid? true},
  :details
  {:valid? false,
  :start-balance 225,
  :end-balance 221,
  :valid-balances false,
  :bad-deltas (),
  :test-length (600),
  :balances (225, 225, 225, 225, 225, 225, 225, 225, 225, 225, 225, 225,
    ...
    225, 225, 225, 225, 225, 225, 225, 225, 229, 225, 225, 225, 225, 225,
    225, 225, 225, 229, 225, 225, 221, 221, 221, 225, 225, 221, 230, 230,
    230, 230, 230, 230, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221,
    221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221,
    221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221,
    221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221,
    221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221, 221,
    ...
    221)
} :valid? false}
```

6.2 Failed test 2

```
{:perf
 {:latency-graph {:valid? true},
  :rate-graph {:valid? true},
  :valid? true},
 :details
 {:valid-balances false,
  :valid-account-transaction-info false,
  :valid? false,
  :start-balance 225,
  :bad-deltas (),
  :valid-bad-deltas true,
  :test-length 300,
  :bad-balances
  (225
  ....
  9007199254741217
  225
  225
  225
  2097377
  225
  225
  2097377
  225
  ...
  225),
  :bad-account-transaction-info (...),
 :valid? false}
```

6.2.1 Transaction details

```
{:account 12, :delta 2097151,
:last-balance 39, :first-balance 2097190,
  :account-transactions
  ({:account 12,: logicaltime 1463948026732284106,
    :timestamp 132, :balance 36,
    :delta 2097154}
  {:account 12, :logicaltime 1463948026732284106,
    :timestamp 126, :balance 39,
    :delta -3}
  {:account 12, :logicaltime 1463948026732284106,
    :timestamp 121, :balance 41,
    :delta -2})}

{:account 9, :delta 2097154,
  :last-balance 4, :first-balance 2097158,
  :account-transactions
  ({:account 9, :logicaltime 1463948029469462306,
    :timestamp 128, :balance 5,
    :delta 2097153}
  {:account 9, :logicaltime 1463948029469462306,
    :timestamp 119, :balance 4,
    :delta 1}
  {:account 9, :logicaltime 1463948029469462306,
    :timestamp 77, :balance 0,
    :delta 4})}

{:account 0, :delta -9,
  :last-balance 24, :first-balance 9007199254741007,
  :account-transactions
  ({:account 0, :logicaltime 1463948024660475877,
    :timestamp 114, :balance 9007199254741011,
    :delta -4}
  {:account 0, :logicaltime 1463948024660475877,
    :timestamp 79, :balance 24,
    :delta -5}
  {:account 0, :logicaltime 1463948024660475877,
    :timestamp 68, :balance 22,
    :delta 2})}
```

file	injections
OPTIONS-000005	328999
000003.log	255220
MANIFEST-000001	254546
000006.log	104112
MANIFEST-000009	40376
000008.sst	40163
000007.log	39081
OPTIONS-000008	33843
MANIFEST-000005	33722
000004.sst	32576
000010.log	6610
000009.log	5880
000012.sst	3549
000011.log	1429
000013.sst	1284
000011.sst	1062
000013.log	34
000009.dbtmp	1

Table 6.1: Fault injections per file name.

filetype	injections	percentage
log	412366	34.8728%
options	362842	30.6847%
manifest	328644	27.7926%
sst	78634	6.6499%
dbtmp	1	0.0001%

Table 6.2: Fault injections per filetype.

Table 6.3: Events triggered in bank tests using the single table method.

	0 injections		1 injection		2 injections		50 injections		1000 injections	
type	total	events/s	total	events/s	total	events/s	total	events/s	total	events/s
fail delete	0	0	2859	0.347	3204	0.478	2804	0.657	5504	0.557
fail read	0	0	2982	0.362	3232	0.482	2836	0.664	5692	0.576
fail transfer	3063	0.170	4190	0.509	4179	0.623	3360	0.787	7149	0.723
info delete	827	0.046	2838	0.345	2166	0.323	2622	0.614	5989	0.606
info read	890	0.049	2679	0.326	2291	0.342	2609	0.611	6188	0.626
info start	0	0	24402	2.965	19550	2.916	12552	2.939	25460	2.575
info transfer	4158	0.231	4245	0.516	3295	0.492	3216	0.753	7182	0.726
invoke delete	27419	1.523	16443	1.998	13838	2.064	10229	2.395	21940	2.219
invoke read	27312	1.517	16611	2.018	13877	2.070	10287	2.409	22314	2.257
invoke transfer	27471	1.526	16435	1.997	13723	2.047	10255	2.402	22578	2.283
ok delete	26592	1.477	10746	1.306	8468	1.263	4803	1.125	10447	1.056
ok read	26422	1.468	10950	1.331	8354	1.246	4842	1.134	10434	1.055
ok transfer	20250	1.125	8000	0.972	6249	0.932	3679	0.862	8247	0.834

Table 6.4: Events triggered in bank tests using the multi table method.

	0 injections		1 injection		2 injections		50 injections		1000 injections	
type	total	events/s	total	events/s	total	events/s	total	events/s	total	events/s
fail delete	0	0	308	0.022	657	0.090	713	0.152	715	0.071
fail read	0	0	300	0.022	630	0.086	685	0.146	751	0.074
fail transfer	6180	0.327	3910	0.285	2222	0.303	1185	0.252	2563	0.253
info delete	292	0.015	4164	0.303	2318	0.316	3295	0.702	11355	1.122
info read	750	0.040	4754	0.346	2358	0.321	3376	0.719	11612	1.147
info start	0	0	35192	2.560	20036	2.732	12144	2.587	27338	2.701
info transfer	84	0.004	3761	0.274	2003	0.273	3145	0.670	10732	1.060
invoke delete	45392	2.401	32307	2.351	18360	2.503	11704	2.494	28262	2.793
invoke read	45594	2.412	32849	2.390	17912	2.442	11575	2.466	28481	2.814
invoke transfer	45820	2.424	32454	2.361	17970	2.450	11403	2.430	28098	2.777
ok delete	45100	2.386	27835	2.025	15385	2.098	7696	1.640	16192	1.600
ok read	44844	2.372	27795	2.022	14924	2.035	7514	1.601	16118	1.593
ok transfer	39556	2.092	24783	1.803	13745	1.874	7073	1.507	14803	1.463

Percentage	Type
74.437%	Connection closed
25.368%	Timeout
0.105%	I/O error sending backend
0.037%	Unknown reason
0.021%	Socket closed
0.009%	Server is not accepting clients
0.008%	Connection reset
0.006%	Connection refused
0.005%	Database “system” does not exist
0.002%	Checksum mismatch
0.001%	Context deadline exceeded
0.001%	Stream closed

Table 6.5: Reasons for unsure transfers in banking tests.

Table 6.6: Events triggered in bank tests using the multi table method, only injecting on node N3.

	1 injection		2 injections		50 injections		1000 injections	
type	total	events/s	total	events/s	total	events/s	total	events/s
fail delete	172	0.0134	72	0.006	16	0.001	253	0.020
fail read	123	0.010	76	0.006	26	0.002	173	0.014
fail transfer	4082	0.319	3902	0.312	3835	0.315	4055	0.317
info delete	748	0.0584	726	0.058	2340	0.192	3258	0.255
info read	951	0.074	823	0.066	2608	0.214	3631	0.284
info start	75258	5.876	73778	5.902	71902	5.897	76140	5.948
info transfer	512	0.040	417	0.033	2103	0.172	3208	0.251
invoke delete	31207	2.436	30579	2.446	31196	2.559	32907	2.571
invoke read	31161	2.433	30838	2.467	31321	2.569	33542	2.620
invoke transfer	31772	2.481	30628	2.450	31536	2.587	33765	2.638
ok delete	30287	2.365	29781	2.382	28840	2.365	29396	2.297
ok read	30087	2.349	29939	2.395	28687	2.353	29738	2.323
ok transfer	27178	2.122	26309	2.105	25598	2.099	26502	2.070

Table 6.7: Events triggered in bank tests using the single table method, only injecting on node N3.

	1 injection		2 injections		50 injections		1000 injections	
type	total	events/s	total	events/s	total	events/s	total	events/s
fail delete	601	0.046	473	0.0378	1083	0.081	474	0.0379
fail read	643	0.049	442	0.035	1030	0.0768	499	0.040
fail transfer	3053	0.233	2732	0.218	3418	0.255	2842	0.227
info delete	2940	0.224	1593	0.127	5108	0.381	3083	0.247
info read	2924	0.223	1529	0.122	5087	0.379	3115	0.249
info start	79742	6.080	76274	6.099	83250	6.206	77084	6.168
info transfer	5425	0.414	3882	0.310	7342	0.547	5310	0.425
invoke delete	22809	1.739	21158	1.692	25837	1.926	22453	1.797
invoke read	22928	1.748	20683	1.654	25939	1.934	22898	1.832
invoke transfer	22999	1.754	20541	1.642	26237	1.956	22444	1.796
ok delete	19268	1.469	19092	1.527	19646	1.464	18896	1.512
ok read	19361	1.476	18712	1.496	19822	1.478	19284	1.543
ok transfer	14521	1.107	13927	1.114	15477	1.154	14292	1.144

Table 6.8: Events triggered in the monotonic tests using indexes.

	0 injections		1 injection		50 injections		1000 injections	
type	total	events/s	total	events/s	total	events/s	total	events/s
fail add	0	0	1314	0.097	1940	0.146	926	0.097
fail read	0	0	4	0.001	13	0.001	13	0.001
fail read-withindex	0	0	4	0.000	12	0.001	13	0.001
info add	2182	0.139	2085	0.154	3704	0.279	4032	0.422
info start	0	0	43844	3.240	42166	3.175	30796	3.224
invoke add	63477	4.051	53469	3.951	52421	3.948	37885	3.966
invoke read	51	0.003	44	0.003	43	0.003	31	0.003
invoke read-withindex	51	0.003	44	0.003	43	0.003	31	0.003
ok add	61295	3.912	50070	3.700	46777	3.522	32927	3.447
ok read	51	0.003	40	0.003	30	0.002	18	0.002
ok read-withindex	51	0.003	40	0.003	31	0.002	18	0.002

Table 6.9: Events triggered in the monotonic tests without indexes.

	0 injections		1 injection		50 injections		1000 injections	
type	total	events/s	total	events/s	total	events/s	total	events/s
fail add	0	0	0	0	503	0.064	719	0.095
fail read	0	0	30	0.002	20	0.003	16	0.002
info add	1303	0.091	1502	0.115	1663	0.210	1938	0.255
info start	0	0	43084	3.293	26016	3.286	25124	3.300
invoke add	66036	4.623	59946	4.582	36385	4.596	34877	4.582
invoke read	47	0.003	43	0.00	26	0.003	25	0.003
ok add	64733	4.532	58444	4.467	34219	4.322	32220	4.233
ok read	47	0.003	13	0.001	6	0.001	9	0.001

Bibliography

[1] Amazon. *Elastic Compute Cloud (EC2) Cloud Server & Hosting, AWS.* `https://aws.amazon.com/ec2/`. Accessed: 2016-05-09.

[2] Lakshmi N Bairavasundaram et al. "An analysis of data corruption in the storage stack". In: *ACM Transactions on Storage (TOS)* 4.3 (2008), p. 8.

[3] Hal Berenson et al. "A critique of ANSI SQL isolation levels". In: *ACM SIGMOD Record.* Vol. 24. ACM. 1995, pp. 1–10.

[4] Julie Bort. *Google apologizes for cloud outage that one person describes as a 'comedy of errors'.* `http://uk.businessinsider.com/google-apologizes-for-cloud-outage-2016-4`. Accessed: 2016-05-30.

[5] Eric A Brewer. "Towards robust distributed systems". In: *PODC.* Vol. 7. 2000.

[6] Aaron Brown. "Availability benchmarking of a database system". In: *EECS Computer Science Division, University of California at Berkley* (2002).

[7] Joao Carreira, Henrique Madeira, João Gabriel Silva, et al. "Xception: Software fault injection and monitoring in processor functional units". In: *Dependable Computing and Fault Tolerant Systems* 10 (1998), pp. 245–266.

[8] Cockroach. *CockroachDB architecture.* `https://github.com/cockroachdb/cockroach/blob/master/docs/design.md`. Accessed: 2016-06-04.

[9] CockroachDB. *Index selection in CockroachDB.* `https://www.cockroachlabs.com/blog/index-selection-cockroachdb-2/`. Accessed: 2016-05-10.

[10] Edgar F Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[11] DigitalOcean. *Simple Cloud Computing, Built for Developers.* `https://www.digitalocean.com/`. Accessed: 2016-05-09.

[12] Thomas W. Doeppner. *Operating Systems In Depth.* ISBN 978-0-471-68723-8. John Wiley & Sons, Inc., 2010.

[13] Prof. Dr. Stefan Edlich. *NOSQL Databases.* `http://nosql-database.org/`. Accessed: 2016-05-30.

[14] Facebook. *A Tutorial of RocksDB SST formats.* `https://github.com/facebook/rocksdb/wiki/A-Tutorial-of-RocksDB-SST-formats`. Accessed: 2016-06-04.

[15] Facebook. *RocksDB MANIFEST file.* `https://github.com/facebook/rocksdb/wiki/MANIFEST`. Accessed: 2016-06-04.

[16] Facebook. *RocksDB Options File.* `https://github.com/facebook/rocksdb/wiki/RocksDB-Options-File`. Accessed: 2016-06-04.

[17] Facebook. *Write Ahead Log File Format.*
https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-File-Format. Accessed: 2016-06-04.

[18] Kelly Fiveash. *AWS outage knocks Amazon, Netflix, Tinder and IMDb in MEGA data collapse.*
http://www.theregister.co.uk/2015/09/20/aws_database_outage/. Accessed: 2016-05-30.

[19] Leading Edge Forum. *Data rEVOLUTION.*
http://assets1.csc.com/innovation/downloads/LEF_2011Data_rEvolution.pdf. Accessed: 2016-05-30.

[20] PC freak. *What is Vertical scaling and Horizontal scaling Vertical and Horizontal hardware / services scaling.*
http://www.pc-freak.net/blog/vertical-horizontal-server-services-scaling-vertical-horizontal-hardware-scaling/. Accessed: 2016-06-04.

[21] Hector Garcia-Molina. *Database systems: the complete book.* Pearson Education India, 2008.

[22] Geoff Gasior. *Behind the scenes with Intelś SSD division. A look at reliability, validation, and frickinṕarticle accelerators.*
http://techreport.com/review/26269/behind-the-scenes-with-intel-ssd-division/. Accessed: 2016-06-04.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google file system”. In: *ACM SIGOPS operating systems review.* Vol. 37. 5. ACM. 2003, pp. 29–43.

[24] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *ACM SIGACT News* 33.2 (2002), pp. 51–59.

[25] Google. *Google Cloud Platform.*
https://cloud.google.com/. Accessed: 2016-05-09.

[26] Theo Haerder and Andreas Reuter. “Principles of transaction-oriented database recovery”. In: *ACM Computing Surveys (CSUR)* 15.4 (1983), pp. 287–317.

[27] Wikimedia Commons Hankwang. *Hard drive capacity over time.*
https://commons.wikimedia.org/wiki/File:Hard_drive_capacity_over_time.png. Accessed: 2016-05-03.

[28] Robin Harris. *Seagate’s visible firmware problem.*
http://www.zdnet.com/article/seagates-visible-firmware-problem/. Accessed: 2016-06-03.

[29] Rich Hickey. *The Clojure Programming Language.*
https://clojure.org/. Accessed: 2016-06-07.

[30] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. “Fault injection techniques and tools”. In: *Computer* 30.4 (1997), pp. 75–82.

[31] Kyle Kingsbury. *Call Me Maybe: simulating network partitions in DBs.*
https://github.com/aphyr/jepsen. Accessed: 2016-04-30.

[32] Kyle Kingsbury. *Distributed Systems Safety Analysis.*
http://jepsen.io/. Accessed: 2016-04-30.

[33] Andrew Krioukov et al. “Parity Lost and Parity Regained.” In: *FAST.* Vol. 8. 2008, pp. 127–141.

[34] Dependable Systems Lab. *LFI: A Tool Suite for Library-Level Fault Injection.*
http://dslab.epfl.ch/proj/lfi/. Accessed: 2016-04-06.

[35] Cockroach Labs. *CockroachDB. The scalable, survivable, strongly consistent, SQL database.*
https://www.cockroachlabs.com/. Accessed: 2016-05-30.

[36] Paul D Marinescu and George Candea. “LFI: A practical and general library-level fault injector”. In: *Dependable Systems & Networks, 2009. DSN’09. IEEE/IFIP International Conference on.* IEEE. 2009, pp. 379–388.

[37] Paul Dan Marinescu, Radu Banabic, and George Candea. "An Extensible Technique for High-Precision Testing of Recovery Code." In: *USENIX Annual Technical Conference.* 2010.

[38] Microsoft. *Microsoft Azure: Cloud Computing Platform & Services.* `https://azure.microsoft.com/`. Accessed: 2016-05-09.

[39] Microsoft. *TransactSQL Reference (Database Engine), DBCC CHECKDB (TransactSQL).* `https://msdn.microsoft.com/en-us/library/ms176064.aspx`. Accessed: 2016-06-03.

[40] MySQL. *MySQL 5.7 Reference Manual, CHECK TABLE Syntax.* `http://dev.mysql.com/doc/refman/5.7/en/mysqlcheck.html`. Accessed: 2016-06-03.

[41] NEC. *Silent data corruption in disk arrays: A solution.* `https://docs.oracle.com/cd/B19306_01/server.102/b14231/repair.htm`. Accessed: 2016-06-03.

[42] Wee Teck Ng and Peter M Chen. "Integrating reliable memory in databases". In: *The VLDB JournalThe International Journal on Very Large Data Bases* 7.3 (1998), pp. 194–204.

[43] Regina Lucia Oliveira Mores and Eliane Martins. "A strategy for validating an ODBMS component using a high-level Software Fault Innjection Tool". In: *Dependable Computing.* Springer, 2003, pp. 56–68.

[44] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14).* 2014, pp. 305–319.

[45] Oracle. *Database Administrator's Guide, Using DBMS_REPAIR to Repair Data Block Corruption.* `https://docs.oracle.com/cd/B19306_01/server.102/b14231/repair.htm`. Accessed: 2016-06-03.

[46] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID).* Vol. 17. ACM, 1988.

[47] Tamir Duberstein Peter Mattis. *SQL in CockroachDB: Mapping Table Data to Key-Value Storage.* `https://www.cockroachlabs.com/blog/sql-in-cockroachdb-mapping-table-data-to-key-value-storage/`. Accessed: 2016-05-30.

[48] Raphael Poss. *DIY Jepsen Testing CockroachDB.* `https://www.cockroachlabs.com/blog/diy-jepsen-testing-cockroachdb/`. Accessed: 2016-04-25.

[49] PostgreSQL. *History.* `https://www.postgresql.org/about/history/`. Accessed: 2016-05-30.

[50] Gavin Powell. *Beginning database design.* John Wiley & Sons, 2006.

[51] RocksDB. *A persistent key-value store for fast storage environments.* `http://rocksdb.org/`. Accessed: 2016-05-30.

[52] Pramod J Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence.* Pearson Education, 2012.

[53] Jerome H Saltzer, David P Reed, and David D Clark. "End-to-end arguments in system design". In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 277–288.

[54] Bianca Schroeder and Garth A Gibson. "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" In: *FAST.* Vol. 7. 2007, pp. 1–16.

[55] Sandeep Shah and Jon G Elerath. "Reliability analysis of disk drive failure mechanisms". In: *Proceedings of the Annual Symposium on Reliability and Maintainability.* Citeseer. 2005, pp. 226–231.

[56] David T Stott et al. "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors". In: *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International.* IEEE. 2000, pp. 91–100.

[57] Sriram Subramanian et al. “Impact of disk corruption on open-source DBMS”. In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE. 2010, pp. 509–520.

[58] Ars Technica. *Half-Day Outage Darkens Microsofts Azure Cloud.* http://www.wired.com/2012/02/azure_outage/. Accessed: 2016-05-30.

[59] Matt Tracy. *How CockroachDB Does Distributed, Atomic Transactions.* https://www.cockroachlabs.com/blog/how-cockroachdb-distributes-atomic-transactions/. Accessed: 2016-05-13.

[60] Matt Tracy. *Serializable, Lockless, Distributed: Isolation in CockroachDB.* https://www.cockroachlabs.com/blog/serializable-lockless-distributed-isolation-cockroachdb/. Accessed: 2016-05-13.

[61] European Union. *Towards a thriving data-driven economy.* https://ec.europa.eu/digital-single-market/en/towards-thriving-data-driven-economy. Accessed: 2016-04-05.

[62] Mike Volpi. *Databases Will Inherit the Earth.* https://www.indexventures.com/news-room/index-insight/databases-will-inherit-the-earth. Accessed: 2016-06-02.

[63] Mai Zheng et al. “Torturing databases for fun and profit”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 449–464.

Further Reading

[64] Inc. Aerospike. *ACID & CAP: Clearing CAP Confusion and Why C In CAP C in ACID*. `http://www.slideshare.net/AerospikeDB/acid-cap-aerospike`. Accessed: 2016-05-02.

[65] Wendy Bartlett and Lisa Spainhower. "Commercial fault tolerance: A tale of two systems". In: *Dependable and Secure Computing, IEEE Transactions on* 1.1 (2004), pp. 87–96.

[66] Persi Diaconis and Frederick Mosteller. *Methods for studying coincidences*. Springer, 2006.

[67] Benjamin Erb. "Concurrent Programming for Scalable Web Architectures". Diploma Thesis. Institute of Distributed Systems, Ulm University, Apr. 2012. URL: `http://www.benjamin-erb.de/thesis`.

[68] Richard Hartmann. *RAID sucks*. `http://richardhartmann.de/blog/posts/2012/02/RAID-sucks/`. Accessed: 2016-06-03.

[69] Michael Le and Yuval Tamir. "Fault Injection in Virtualized SystemsChallenges and Applications". In: *Dependable and Secure Computing, IEEE Transactions on* 12.3 (2015), pp. 284–297.

[70] Thomas Naughton et al. "Fault injection framework for system resilience evaluation: fake faults for finding future failures". In: *Proceedings of the 2009 workshop on Resiliency in high performance*. ACM. 2009, pp. 23–28.

[71] Dan RK Ports et al. "Transactional Consistency and Automatic Management in an Application Data Cache." In: *OSDI*. Vol. 10. 2010, pp. 1–15.

[72] Maitrayi Sabaratnam, Oystein Torbjornsen, and Svein-Olaf Hvasshovd. "Cost of ensuring safety in distributed database management systems". In: *Dependable Computing, 1999. Proceedings. 1999 Pacific Rim International Symposium on*. IEEE. 1999, pp. 193–200.

[73] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. "Understanding latent sector errors and how to protect against them". In: *ACM Transactions on storage (TOS)* 6.3 (2010), p. 9.

[74] Christopher A Stein, John H Howard, and Margo I Seltzer. "Unifying File System Protection." In: *USENIX Annual Technical Conference, General Track*. 2001, pp. 79–90.

[75] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. "NoSQL databases". In: *Lecture Notes, Stuttgart Media University* (2011).

[76] Timothy K Tsai, Ravishankar K Iyer, and Doug Jewitt. "An approach towards benchmarking of fault-tolerant commercial systems". In: *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*. IEEE. 1996, pp. 314–323.