• U C •

Universidade de Coimbra

Master in Informatics Engineering
2015-2016

Dependability Assessment of NoSQL Engines

Final Dissertation

Student:
Luís Filipe André Ventura
lfav@student.dei.uc.pt

Supervisor:
Prof. Dr. Nuno Antunes
DEI–UC

September 2, 2016

• U C •

Universidade de Coimbra

Master in Informatics Engineering
2015-2016

Dependability Assessment of NoSQL Engines

Final Dissertation

Student:
Luís Filipe André Ventura
lfav@student.dei.uc.pt

Supervisor:
Prof. Dr. Nuno Antunes
DEI–UC

Jury:
Prof. Dr. Vasco Pereira
DEI–UC

Jury:
Prof. Dr. Raul Barbosa
DEI–UC

September 2, 2016

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Acknowledgements

During the course of not only this dissertation, but my entire academic life I have met many amazing individuals which have helped me grow, not only academically but also into a better person.

First I would like to thank the teachers who have given me most of the knowledge I possess. This knowledge has not only allowed my to finish this work, it will be the foundation for my future. Among the many talented professors I have met during my years at this university there are two to whom I would like to give special thanks. To professor Alberto Cardoso I give my thanks for all the guidance and assistance he provided while I was facing a difficult period of my life. To professor Marco Vieira I give my thanks for both the opportunity to work as research student, as well as all his guidance. He always urges his students to strive to be the best they can, not only by being kind and supportive, but also harsh when needed.

Next I want to thank my family, especially my grandparents. They have raised me the best they could, always striving to provide me with whatever I needed. I have never lacked for anything and I own everything I have accomplished to them.

Thirdly, I would like to thank my close group of friends. Not only have they helped me with with much of my academic work, they have provided me with much needed mental support whenever I was feeling down. In alphabetical order: Ana, André, Claudio, David, Diogo, Ivo, Pedro, and Pratas. I love you guys, thank you for being part of my life.

Last, but certainly not least, I want to say a huge thank you to my supervisor, Nuno Antunes. During this last year, he was more than a supervisor or a friend, he was a mentor. I have never learned so much from someone else in a single year. Your constant friendly presence, desire to always improve yourself each day, and incredible work ethic, make me think I can only hope to be someone as great as you someday.

To everyone present on this list, thank you for everything, without you I would not be where I am today.

Abstract

NoSQL databases are the response to the sheer volume of data being generated, stored, and analyzed by modern users and applications. They see great usage in big data and real-time web applications as they are capable of both: extreme horizontal scaling, and storing data with less rigid structures than the ones used in relational DBMSes. NoSQL databases are known to sometimes compromise consistency in favor of availability, partition tolerance, and performance. Several studies have evaluated the performance of these databases, but their users also need to understand how they behave in the presence of faults and how to quantify the impact of those faults.

As such, the main goal of this dissertation is to perform an experimental assessment of NoSQL databases. This work proposes an experimental methodology based on fault injection, an experimental procedure to implement the methodology, and three metrics based on dependability attributes, which can be used to evaluate the databases. A prototype for a tool, which implements the experimental procedure and automates the testing process, developed during the work is also presented. Lastly, an experimental campaign assessing three popular NoSQL engines was performed to demonstrate both the methodology and tool.

The results of the experimental campaign show that many times, even in the presence of simple faults, the integrity of the data is affected. They also show that each database handles the workloads and faults differently, evidencing that users need to carefully select which solution to use in their systems.

Keywords

NoSQL Databases, Dependability Assessment, Fault Injection

List of Publications

This dissertation is partially based on the work presented in the following publication:

- Luís Ventura, Nuno Antunes, "Experimental Assessment of NoSQL Engines Dependability", *12th European Dependable Computing Conference (EDCC 2016)*, Gothenburg, Sweden, September 5-9, 2016.
  - ***Abstract:*** *NoSQL databases are the response to the sheer volume of data being generated, stored, and analysed by modern users and applications. They are extremely scalable horizontally and store data with less rigid structures than the relational ones. NoSQL databases are known to compromise consistency in favour of availability, partition tolerance, and performance. Several studies evaluated the performance of these databases, but the users also need to understand how they behave in the presence of faults and quantify the impact of those faults. This paper presents an experimental evaluation of NoSQL databases' dependability using fault injection, which compares three widely used NoSQL engines based on how they perform in the presence of operator faults. The results show clearly that many times the integrity of the data is affected, even in the presence of simple faults. It is also shown how different databases handle the workloads and the faults differently, evidencing that users must carefully select the solution to use in their systems.*

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

CQL Cassandra Query Language. 5

DI Data Integrity. 31, 32

IT Impact on the Throughput. 33

NYI Not Yet Implemented. 30

OS Operating System. 29, 30

SPOF Single Points Of Failure. 6

SQL Structured Query Language. 5

SUT System Under Test. 21, 25, 30, 31

YCSB Yahoo! Cloud Serving Benchmark. 1, 11–13, 17, 20, 28, 29, 35

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

1. Introduction

For over forty years, relational databases have been the leading model for data storage, retrieval and management. However, the need to handle data in web-scale systems, in particular Big Data systems, have led to the creation of numerous **NoSQL databases** [37]. Unlike relational databases, NoSQL databases embrace schemaless data, run on clusters, and have the ability to trade-off traditional consistency for other useful properties. Advocates of NoSQL databases claim that they can build systems that are more "performant", scale better, and are easier to program with [42].

Several studies assessing the performance of NoSQL of databases have been performed [20, 26, 28, 44]. There is even a well-established benchmarking tool which has been used in most of these studies, Yahoo! Cloud Serving Benchmark (YCSB) [26]. YCSB allows its users to evaluate the performance of different NoSQL engines by generating and executing workloads against the databases, while collecting measures related to performance such as throughput and latencies. While it is true that performance is very important for users looking for a new database solution for their system, other non-functional attributes such as data integrity or availability can be even more important.

To achieve their levels of performance, NoSQL databases make some tradeoffs. Each engine usually documents these tradeoffs, and there is even a study which uses this documentation alongside existing performance evaluation to try and rate these databases from a quality attribute point of view [37]. There is also one work which tries to evaluate the availability of these databases by simulating cloud anomalies [43]. However, the amount of information regarding the dependability of these databases is still quite limited, and with critical systems starting to include NoSQL databases, there is a need to **understand how each different database behaves under the effect of faults**, as well as the impact of those faults.

Over the last decades, **fault injection** has been the premier technique for the dependability assessment of various systems [33, 47]. It is used to validate the fault tolerance mechanisms of systems and to understand their behavior in the presence of faults. For this evaluation to be accurate, the faults used must be representative of the faults the system face during regular operation.

Fault injection techniques have been used in many studies which assess the dependability of various systems, including dependability benchmarks for Relational Database Management Systems (RDBMSes) [47]. However, as stated before, the number of studies performed on the experimental evaluation of NoSQL database's quality attributes is rather limited. The fact that most of the work is focused on performance evaluation [26] leaves a vacuum in the assessment of attributes such as integrity, availability and reliability. This information may be very useful for users that are planning to use NoSQL databases to handle their data.

This work proposes a methodology to **assess and compare NoSQL databases in terms of dependability attributes**. This methodology is inspired in measurement-based techniques [34] and uses fault injection to characterize the behavior of the NoSQL databases in the presence of faults, with particular interest on the **integrity of the data**, the **availability of the engine**, and on the **impact of the fault in the throughput**.

A tool – **NoSDep** – was developed to implement the proposed methodology. NoSDep is a prototype tool capable of performing automated dependability evaluations on NoSQL databases. It includes components to manage the workload, the injection of the faults and also to collect the measurements relevant to compute the dependability metrics. A third-party widely used workload generator is used (YCSB) [26] to exercise the database during the experiments. Also, until now, the tool implements a fault model with 6 classes of faults.

An experimental campaign was designed to **demonstrate the applicability of methodology** and the **usage of the tool**. It consisted in the evaluation of three popular NoSQL databases. These evaluations used the faults of the proposed fault load, and three different workload types. The campaign also served to observe how the different NoSQL databases behaved in the presence of the faults. The engines have different recovery mechanisms which lead to very different recovery times and data integrity problems. It is interesting to notice that some engines had problems even in the presence of very simple faults.

A key concern in this kind of evaluation is related to the representativeness, which is mainly influenced by the workloads executed and faults injected. Regarding the workload, the option of using a an existing tool, which is accepted by the industry and academia, provides us with several different predefined configurations that represent typical big data usage scenarios. The fault model must emulate a set of real faults experienced by NoSQL databases during their activity. With hardware achieving high levels of reliability, software faults and operator faults are nowadays considered the most frequent causes of computer failures [38, 47]. Moreover, while the current implementation of the tool and the experimental campaign performed use only operator faults, the approach was designed to be generic and therefore can easily be extended with other types of faults.

This work was able to successfully evaluate the dependability of three different databases. The methodology was validated by performing an experimental campaign. The engines were evaluated according to a previously defined set of metrics, which were calculated with the measurements collected during over 810 test slots. The results show that the **integrity of the data is not always guaranteed**. In fact, records were lost even when the fault injected was simple and apparently harmless (e.g. clean restart of the engine), resulting in 403 data integrity issues in a total of 810 tests ($\approx$ 49.75%). In 55 cases, the engine also finished the test slot with one extraneous record that was never confirmed to the client.

1.1. Contributions

The main goal of this work was to research techniques to assess the dependability of NoSQL databases. The main contributions are listed below:

- An experimental methodology to evaluate and compare NoSQL databases. While many studies have evaluated these databases' performance, work on other quality attributes is limited. After researching existing literature, only one experimental methodology that focuses on anomaly prediction to characterize availability was found. The methodology is presented in Section 3.

- A prototype tool – NoSDep – that implements the proposed methodology. To validate the methodology and evaluate the databases, NoSDep was developed to automate the experimentation process made the experiments take much less time. The tool is presented in Section 4.

- An experimental campaign and the analysis of its results. An experimental campaign using 3 different databases, 3 workloads and 6 fault types was performed. The campaigned showed several problems on the tested databases. The campaign is presented in Section 5.

1.2. Thesis Structure

This document is divided in chapters, organized as follows:

- Chapter 2 contains the background and related work. It introduces the fundamental concepts of this work, such as NoSQL, Dependability, and Performance and Dependability assessment. Regarding the assessment part, the most prominent techniques, as well as some works which make use of them are also presented.

- Chapter 3 presents the main reserach objectives, as well as the approach used to reach those objectives. The approach section introduces the methodology that is proposed to assess NoSQL databases, explain every component of the methodology, and presents the metrics that are proposed to rate the evaluated databases.

- Chapter 4 introduces the tool that was developed to automate the evaluation of NoSQL databases. It begins by providing a broad view of the components which compose the tool, followed by an explanation of each component. This chapter also contains a list of the necessary prerequisites to run the tool.

- Chapter 5 presents the experimental campaign that was performed in order to demonstrate the methodology and tool. The chapter begins by presenting the main research questions the campaign was attempting to answer. Next, it presents the experimental setup that was used during the campaign, including the rationale behind the selection of the engines. Lastly, the chapter presents the results obtained the results obtained during the campaign, as well as a discussion where the research questions are answered based on these results.

- Finally, chapter 6 contains the main conclusions and the future work. The conclusions cover the strong and the weak points of this work. The future works present the next steps that will be taken in furthering this research.

This page is intentionally left blank.

2. Background and related work

This chapter introduces the concepts most important to this dissertation. It contains an introduction to each technology involved, as well as review of some of the most important related work.

Section 2.1 explains why NoSQL databases were created and why they have recently been gaining traction in the market. Section 2.2 discusses some studies performed on evaluating the performance of these databases. Section 2.4 introduces the concept and importance of dependability and dependable systems, and it is followed by section 2.5 where the most prominent techniques used to evaluate the dependability of a system are introduced. This last section also presents some of the existing studies which make use of these techniques and a review of existing dependability assessment studies targeting NoSQL databases.

2.1. NoSQL

"The term NoSQL was first coined in 1988 to name a relational database that did not have a Structured Query Language (SQL) interface" [37]. It reappeared in 2009 to name an event which highlighted new non-relational databases, such as Google's BigTable [25] and Amazon's DynamoDB [28].

While the term initially stood for "No SQL", a few NoSQL engines do have query languages, such as Cassandra's Cassandra Query Language (CQL). As such, "non-relational" or "Not Only SQL" are more popular (and correct) definitions.

There are four general types of NoSQL databases, each with their own specific characteristics and attributes [37, 44]:

- **Key-value stores** – The least complex databases. They are designed for storing data in a schema-less way. In key-value stores, all data consists of an indexed key paired with a value. A few examples of key-value stores are: Redis [2], Aerospike [4], and DynamoDB [10]
- **Column stores** – While relational databases serialize the rows together, column databases serialize the columns. Row databases are optimized for operations that target a specific record, usually involving several of the record's columns. On the other hand, column databases excel in operations that target only a subset of a row's columns. A few example of column stores are: Cassandra [5], HBase [11], and Accumulo [3]
- **Document stores** – These databases store their records as documents. Documents can be seen as a grouping of key-value pairs. Unlike relational DBMSes, in a document store each document has its own schema. A few examples of document stores are: MongoDB [1], OrientDB [16], and CouchDB [8]
- **Graph stores** – These databases excel at dealing with interconnected data. They consist of connections between nodes. Each node can hold any number of key-value pairs and each edge contains a semantically relevant label. A few examples are: Neo4j [14], OrientDB [16], and Titan [19]

In addition to the four previously mentioned types, there are also multi-model NoSQL databases, such as OrientDB which appeared on the previous list in two different database types, which support multiple data models [39].

In the next subsections the importance of NoSQL databases is explained as well as their main use cases, the trade-offs they make to achieve their features and the most popular databases of each type.

2.1.1. Purpose and importance

NoSQL databases were created to deal with some of the traditional relational-database's limitations, particularly, handling Big Data. The almost limitless amount of data collection technologies ranging from point of sale systems, to GPS tools, or social networks connecting millions of users and their information, all act as force multipliers for data growth [27].

To understand how NoSQL databases help deal with the problem of Big Data, it is important to analyse the properties of a Big Data project. Big Data projects are usually typified by [27]:

- **High data velocity** - lots of data coming in very quickly, possibly from different locations.
- **Data variety** - storage of data that is structured, semi-structured, and/or unstructured.
- **Data volume** - data that involves many tera- or petabytes in size.
- **Data complexity** - data that is stored and managed in different locations or data centers.

Based on these needs, to solve a Big Data project a system should have high availability, be highly scalable and be very fast. Those are the selling points of many NoSQL solutions:

- **Horizontal scalability**: One of the most common points mentioned when talking about NoSQL databases is their inherent support of horizontal scalability (i.e. scaling by increasing the number of computers). Horizontal scalability is not only cheaper than vertical scalability (i.e. scaling by increasing the power of the computer being used), since the upgrading consists of adding additional machines to the pool, it typically involves no system downtime [36].

  In fact, while relational databases are also capable of scaling horizontally, the relational nature of their data structure makes the joining of their tables in a distributed system a difficult task to manage.

- **High availability**: We are on the Information Age. Nowadays almost everyone is connected to the web, and downtime can be a deadly blow to a company's reputation. NoSQL databases were designed to be used in a distributed architecture, as such, there are no Single Points Of Failure (SPOF) and there is built-in data replication. [37, 44]

  These properties mean that even if one node goes down there should be no downtime or data loss in the system.

- **Performance**: NoSQL databases were designed to be used in a distributed architecture. The fact that the data is replicated among several nodes means that not only is availability very high, read performance will also be higher due to load balancing. Likewise, write performance is also very high since data partitioning allows for parallel writes [37].

2.1.2. Trade offs

When talking about trade-offs in NoSQL databases, it is important to mention the CAP (**C**onsistency, **A**vailability, and **P**artition-tolerance) theorem. Introduced by Eric Brewer in 2000, the CAP theorem states that any networked shared-data system can have at most two of these three desirable properties [24]:

- **C**onsistency: equivalent to having a single up-to-date copy of the data;
- **A**vailability : high availability of the data;
- **P**artition tolerance : tolerance to network partitions.

For many years, relational databases have been stamped as being consistent and available (CA), and having trouble with partitions. In the context of distributed databases however, since a database can't be called a reliable distributed database if it is not partition tolerant, the behaviour of the database usually falls on two one of these two types: if a network split happens, should the database keep responding to requests with possibly old/wrong data? (AP) or should it stop responding unless it is capable of ensuring it has access to the latest copy of the data? (CP). [30]

When they have to ensure partition tolerance, distributed relational databases typically fall into the category of CP (favouring consistency and partition tolerance), i.e., in the case of a network partition happening, they favor data consistency over availability (e.g. If the communication between two datacenters is broken, either only one of them will keep replying to requests or both will stop, in order to ensure data is consistent). [30]

In truth, even though many relational databases favor consistency, there are some eventually consistent (databases where update operations executed on one node eventually execute on all nodes) databases, like MariaDB, which are starting to appear. According to the CAP theorem, these databases would be classified as AP (favouring availability and partition tolerance) [23, 35].

When NoSQL databases first appeared, a common misconception regarding their classification using the CAP theorem was that they are all AP (favouring availability and partition tolerance), sacrificing data consistency in the process. Nowadays, its understood that while some NoSQL databases do favor availability and others favor consistency, they are not fully (i.e. only) consistent or fully available. Instead, these databases offer fine-tuning of consistency and availability levels, allowing their users to configure the database to suit their needs [37].

Image 1 contains a visual representation of the cap theorem, as well as the CAP properties favoured by a few NoSQL databases. Like stated before, in the case of NoSQL databases, these properties can usually be fine-tuned. Cassandra for example is represented as AP, since by design it uses eventual consistency (meaning all nodes will eventually have the same data) to ensure high availability. However, it is also possible to configure Cassandra to use strong consistency (ensuring tha data is consistent across all nodes before accepting new operations),i.e., favouring CP in the CAP theorem [5].

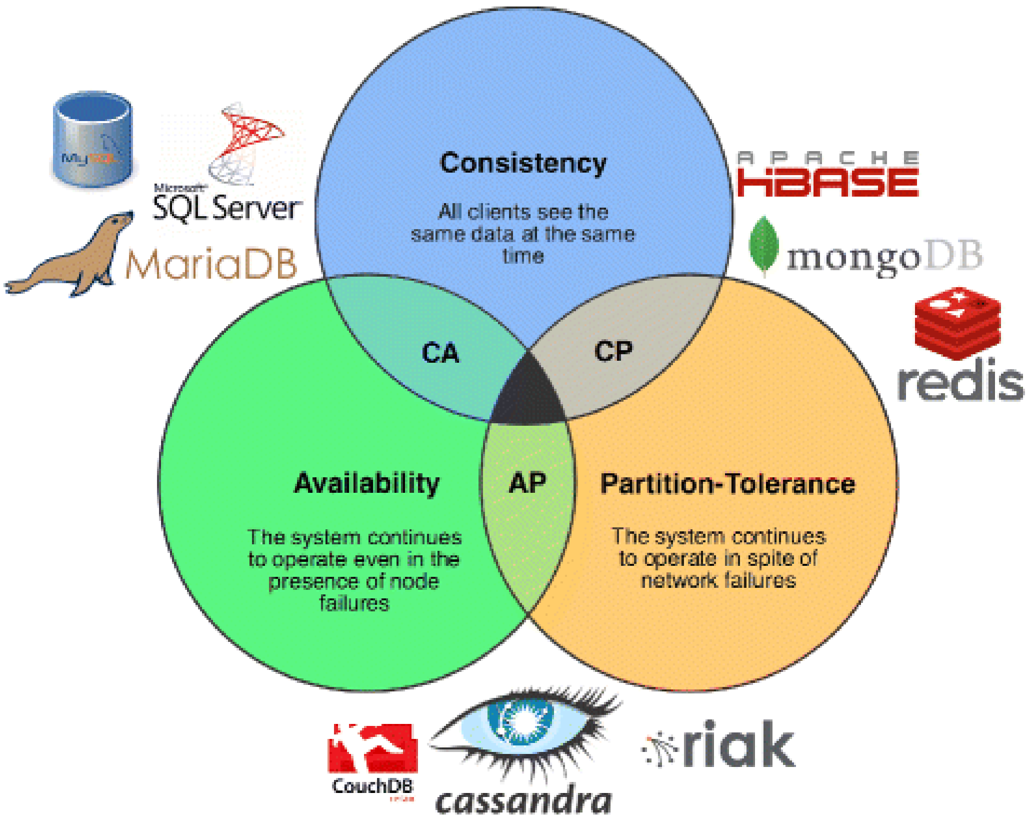
Consistency
All clients see the same data at the same time
CA
CP
Availability
The system continues to operate even in the presence of node failures
AP
Partition-Tolerance
The system continues to operate in spite of network failures
MySQL
SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

Figure 1: CAP theorem and properties favoured by each database (from [37]).

Another important point when talking about trade-offs in NoSQL databases is ACID transactions. ACID is a set of properties that guarantee that database transactions are executed reliably.

ACID stands for [29]:

- **A**tomicity - during transactions, if any one part of the transactions fails, the entire transactions fails and the database state is left unchanged.
- **C**onsistency - even though it shares the same name, it has nothing to do with the consistency from CAP's theorem. In ACID, consistency means that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.
- **I**solation - during the execution of concurrent transactions, the resulting system state must be the same as if the transactions were executed serially, i.e., one after the other.
- **D**urability - after a transaction has been committed, it will remain so, even in the event of power loss, crashes, and/or errors.

When the first NoSQL databases appeared, they sacrificed ACID transactions for the sake of performance, and used what are called BASE transactions. Like ACID, BASE is a set of properties of database transactions.

BASE stands for [40]:

- **B**asically **A**vailable - means that the system guarantees availability, in terms of the CAP theorem.
- **S**oft state - indicates that the system may change over time, even without input. This property derives from the eventual consistency model of the system.
- **E**ventually Consistent - indicates that the system will become consistent over time, given that it doesn't receive new input during that time.

There is one more important trade-off made by NoSQL databases. **Table joins**.

Relational databases are usually normalized to eliminate duplication of information such as when objects have one-to-many relationships. What this means is that the information can be kept in different tables and when there is the need to combine information from both tables a join can be performed.

Unlike relational databases, NoSQL databases are usually denormalized, i.e., information is not kept in self-sustained entities. A good example to understand normalized and denormalized data is the storage of people and their phone numbers. In a normalized environment, people and phone numbers are kept in self-contained entities and a join operation is performed when needed. In denormalized environments, however, instead of two tables we would have only the person table, with a list of phone numbers inside.

Normalization is better to ensure data integrity, as there is no data duplication, however since joins can be heavy operations, denormalized data is typically faster to access.

While most NoSQL databases don't support either ACID transactions or table joins, there are some that actually do. Table 1 contains a list of the NoSQL databases that support ACID transactions and joins.

`Database`	`ACID transactions`	`Joins`
Aerospike	Yes	No
ArangoDB	Yes	Yes
CouchDB	Yes	Yes
c-treeACE	Yes	Yes
HyperDex	Yes	Yes
InfinityDB	Yes	No
LMDB	Yes	No
MarkLogic	Yes	Yes
OrientDB	Yes	Yes

Table 1: NoSQL databases supporting ACID transactions and joins

2.1.3. NoSQL databases usage

As was mentioned before, NoSQL engines are very popular. This subsection presents some use cases where NoSQL databases are being used.

Cassandra is being used by instagram to store auditing information related to security and site integrity purposes. Before changing to cassandra, instagram was using Redis (an in-memory NoSQL database) to store this information. Since using disks is much cheaper than using memory, and this data was mostly about writes, instagram did not lose any performance and was able to save 75% of the costs to store the same amount of information [6].

Ebay is building an elastic, highly scalable, and highly available metadata store [13], using mongodb as its storage engine.

Redis is an in-memory key-value pair database, commonly used as an in-memory database, message queue, or cache. The fact that redis is the third most deployed container in docker is testament to its popularity [17].

Neo4j is a widely used graph database. Gamesys is using Neo4j on its new social network. Graph databases are well suited to handle social networks, due to the massive connectedness of the data. Neo4j was chosen because Gamesys judged it to be the most mature graph database [15].

Following, we present a list with the most popular databases for each database type [9].

- **Key-value stores**:
    1. Redis [2]
    2. Memcached [12]
    3. RIAK KV [18]
- **Column stores**:
    1. Cassandra [5]
    2. HBase [11]
    3. Accumulo [3]
- **Document stores**:
    1. MongoDB [1]
    2. CouchBase [7]
    3. DynamoDB [10]
- **Graph stores**:
    1. Neo4j [14]
    2. OrientDB [16]
    3. Titan [19]

Relational databases are still the most popular, however, it should be noted that among the 12 databases listed in this section, three of them are in the top10 of the most popular databases (MongoDB,Cassandra, and Redis)[1].

Additional information about any of these databases can be found in their respective websites. Additionally, in section 5.1, the ones selected for the experimental campaign are explored in detail.

[1]Ranking obtained from db-engines (http://db-engines.com/en/ranking), on August 2016

2.2. Performance Evaluation of NoSQL databases

Nowadays, the scaling and speed performances of NoSQL databases [26], the trade offs made by them in order to achieve their levels of performance [28, 37, 44], and the main use cases where NoSQL databases can be used [31], all have a lot of research performed on them.

Some NosSQL databases, like Amazon's DynamoDB in 2007 [28], have had evaluations performed before they were even called NoSQL databases.

While DynamoDB's evaluation is based on field measurements, i.e. data collected from the system performing in a real scenario, most of the recent NoSQL database's performance evaluations have made use of Yahoo! Cloud Serving Benchmark (YCSB).

YCSB was used to compare six different databases in [41]. This work used workloads focusing on write, read, and scan operations, and measured the throughput, read/write latencies of each engine and how much space each engine required to store the same data on disk.

Also using YCSB, the work in [20] focused on comparing the reading and update performances of MongoDB and Cassandra. In this work, the workloads chosen were comprised of mostly read or update operations. The authors ran several workloads, with varying amounts of data and in the end compared the baseline performance of each database.

The study performed in [44] evaluated a couple of NoSQL databases, alongside a relational database(MySQL), from both a quantitative and a qualitative point of view. The qualitative analysis consisted in checking whether a set of features was available for the NoSQL databases taken into account. The quantitative analysis consisted in using YCSB to run workloads of 120 million operations against the databases, and collecting information regarding the throughputs and latencies.

2.3. YCSB

As was stated before, most of the performance evaluations performed on NoSQL databases make use of YCSB [26].

YCSB is a framework for benchmarking systems, which was released in 2010 by workers in the research division of Yahoo!, with the goal of "facilitating performance comparisons of the new generations of cloud data serving systems" [26].

The framework consists of a workload generating client and a package of standard workloads that cover interesting parts of the performance space(read-heavy workloads, write-heave workloads, etc.) [26].

Besides the workload generator, to run YCSB additional code to interface with a data serving system is needed. The creators of YCSB also developed a tool called the YCSB client which is used to execute the YCSB benchmarks. This tool was developed with with extensibility in mind, as such it is possible to add both new interfaces to connect to databases and different workloads to it.

When YCSB was proposed, the authors also proposed six different workloads, each exploring different parts of the performance space. These workloads, which come bundled with the YCSB client, are considered representative of real-world applications and have been used in many performance evaluations [26][44][28][20]. The list presented below contains the core set of workloads, with information regarding the type of operations, the performance space it is evaluating, and an application example of the workload.

- **workloadA** (50% reads, 50% updates) is an update-heavy workload and "an application example is a session store recording recent actions" [26].
- **workloadB** (95% reads, 5% updates) is a read-heavy workload and "an application example is photo tagging; adding a tag is an update, but most operations are to read tags" [26].
- **workloadC** (100% reads) is a read-only workload and "an application example is a user profile cache, where profiles are constructed elsewhere (e.g. Hadoop" [26].
- **workloadD** (95% reads, 5% updates) is a read-heavy workload and "an application example is user status updates; people want to read the latest" [26].
- **workloadE** (95% scans, 5% inserts) is a read-heavy (bulk) workload and "an application example is threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)" [26].
- **workloadF** (50% reads, 50% read-modify-writes) is a read and update-heavy workload and "an application example is a user database, where user records are read and modified by the user or to record user activity" [26].

To run a workload using YCSB there are six steps:

1. Set up the database system to test
2. Choose the appropriate DB interface layer
3. Choose the appropriate workload
4. Choose the appropriate runtime parameters.
5. Load the data.
6. Execute the workload.

Setting up the database system:

This step consists in setting up the database system which is going to be evaluated. Besides the installation and normal configurations of the database system, it may also be necessary to create tables/keyspaces/storage buckets to store records.

Choosing the appropriate DB interface layer:

The interface layer is a Java class that will execute the operations generated by the YCSB client against the database's APIs. The YCSB client already comes packed with interfaces to run the workloads against some of the most popular NoSQL databases (e.g. MongoDB, Cassandra, Redis, etc) in the market, however, new interfaces can be created if needed.

Choosing the appropriate workload:

The workload defines the data that will be loaded into the database during the loading phase, as well as the operations that will be executed against the dataset during the transaction phase.

The workloads are a combination of :

- A parameter file containing the properties of the dataset.
- A workload Java class which uses the properties specified in the parameter file to insert records (during the loading phase) or execute transactions (during the transaction phase).

Choosing the appropriate runtime parameters:

Besides the specific types of workload and properties of data sets, there are additional settings which can be specified when running the benchmark. Some of these settings are:

- *threads n* : **the number of client threads**. By default, YCSB uses a single worker thread, but additional threads can be specified to increase the load against the database.
- *target n* : **target number of operations per second**. By default, YCSB will attempt to do as many operations as it can. This parameter can be used to throttle the operation throughput if needed.
- *s* : **status**. Setting this parameter will make the client report its status every *10* seconds.

Loading the data:

Like stated before, workloads have a loading step and a transaction step. To execute the loading phase a command like the one below is used:

```
#run loading phase
python bin/ycsb load mongodb -P workloads/workloada -P properties/mongodb >
    loading_output.out
```

Listing 1: Running the loading phase.

If running from the root folder of ycsb, what this will do is call the YCSB client, tell it to execute the loading phase using the MongoDB interface, specify the workload and property files to be used and log the output to a file called output.out.

The loading phase is meant to populate the database, and consists only of insert operations. As such, independently of the workload specified, YCSB's behaviour will be the same: generate data and insert it into the database.

Executing the workload:

Once the data is loaded, the workload can be executed. This is done by telling the client to run the transaction section of the workload.

```
#run transaction phase
python bin/ycsb run mongodb -P workloads/workloada -P properties/mongodb >
    transactions_output.out
```

Listing 2: Running the transaction phase.

As can be seen from the snippets, for YCSB users, the only difference in running the loading and transactions phases is changing the *load* to *run*.

The transaction phase of the workload will execute operations against the dataset inserted during the loading phase. The operations to be performed (i.e. updates, reads, etc) are generated according to the workload file.

After the execution of the transaction phase, the client will direct the performance statistics to stdout. These statistics include minimum and maximum latencies, number of successful operations and throughput(ops/sec), to name a few.

2.4. Dependability

The dependability of a system is the system's capacity to deliver service which can be trusted. Another definition provided by Avižienis et al. is: "the dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable." [22]

Image 2 contains the dependability tree [22], which is a visual representation of the attributes, threats and means of systems.

A dependable system must be able to satisfy a set of attributes, in order to be capable of resisting the existing threats. To attain the required attributes, a system has many means which have been developed over the course of the past 60 years.

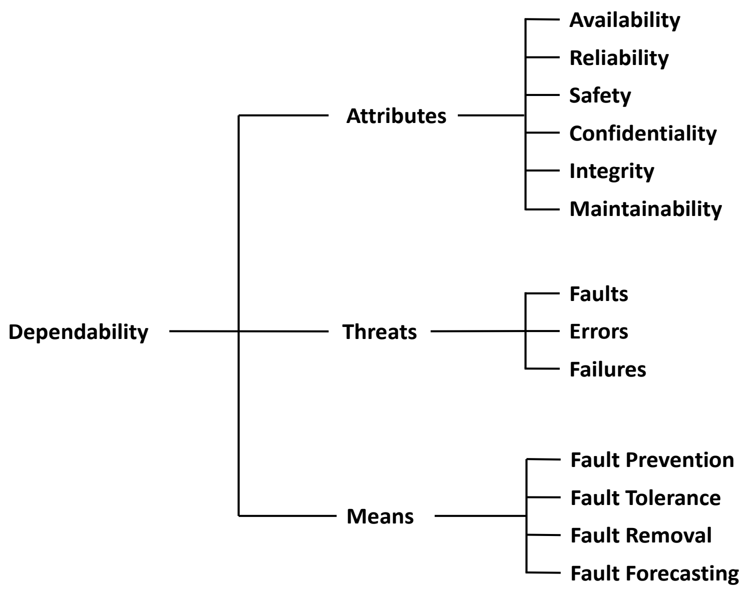

Figure 2: Dependability tree (from [21])

The next subsections describe each element of the dependability tree.

2.4.1. Attributes

A system is said to be dependable if it can satisfy the following quality attributes [22, 47]:

- **Availability**: readiness for correct service.
- **Reliability**: continuity of correct service.
- **Safety**: absence of catastrophic consequences on the user(s) and the environment.
- **Confidentiality**: absence of unauthorized disclosure of information.
- **Integrity**: absence of improper system alterations.

- **Maintainability**: ability to undergo modifications and repairs.

2.4.2. Threats

During normal execution, a system may be affected by threats which cause a drop in dependability. The three types of threat are:

- **Faults**: defects in the system. The presence of a fault may not always lead to a failure, as depending on the system's inputs and state, the fault may never be executed. When producing errors faults are called active, otherwise they are called dormant.
- **Errors**: discrepancies between the intended and actual behaviours of the system. They occur at runtime, whenever a fault gets activated.
- **Failures**: instances in time when the system displays behaviour that is contrary to its specification. It should be noted that not all errors give way to failures. e.g. If an exceptions gets thrown but is correctly handled, the system will not enter a failure state.

Figure 3 gives a visualization of the fault-error-failure chain [21]. Faults originate errors which in turn give way to failures, unless they are correctly handled.

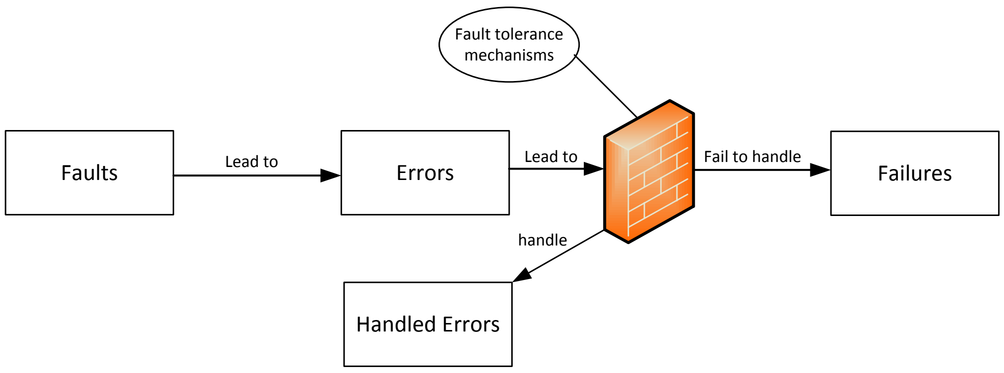

Figure 3: Fault-error-failure chain, with fault tolerance mechanisms in place (adapted from [22]).

Failures can be categorized in two different levels, according to the relation between the benefit of having the system perform in the absence of a failure, and the consequences of a failure happen [21]:

- **minor failures**: where the harmful consequences of failures are of similar or lower cost than the benefits provided by correct service delivery.
- **catastrophic failures**: where the harmful consequences of failures are orders of magnitude, or even incommensurably higher than the benefits provided by correct service delivery.

To consider a system dependable in the presence of failures there are several criteria that can be used to determine the class of the failure's severity. A few examples of such criteria are:

- for availability, the downtime of the service.
- for safety, the possibility of having endangered human lives.
- for confidentiality, the type of information that may have been unduly disclosed.
- for integrity, the extent of the corruption of data, as well as the system's capacity to recover from the corruption.

2.4.3. Means

Having mentioned the dependability attributes and the threats they are subjected to, it is important to talk about the means to break the fault-error-failure chains.

In figure 3 one of the possible means of improving software dependability is shown, Fault Tolerance.

The four means to ensure software dependability are the following [21, 22]:

- **Fault Prevention** - technique to prevent the occurrence or introduction of faults. This technique consists in the usage of quality control techniques during the design and manufacturing phases of the system.
- **Fault Tolerance** - technique to preserve correct delivery of service in the presence of faults. Fault tolerance is comprised of two main techniques: **Error detection**, which consists in identifying the presence of an error, and **Recovery**, which consists in turning a system state with the presence of errors into a state without errors or faults that may be activated again.
- **Fault Removal** - technique to reduce the number or severity of faults. It can occur during system development or system operation.
  - **Removal during development** - Consists of three steps : the **verification** step which is always executed and consists in checking if system adheres to a set of given properties; the **diagnosis** step which is executed only if the verification fails, and consists in diagnosing the faults that prevent the verification properties from being fulfilled; the **correction** step, which is executed after the diagnosis steps and consists in the necessary corrections to the system.
  - **Removal during operation** - Consists in the use of corrective or preventive maintenance (patches, updates, etc.). Corrective maintenance aims to remove faults that have produced one or more errors and have already been previously reported, whereas preventive maintenance is aimed at uncovering and removing faults before they cause errors during normal operation.
- **Fault Forecasting** - technique to predict the likelihood of faults, so they can be removed or have their effects circumvented.

2.5. Dependability assessment

As dependability is a 'meta-attribute', which encompasses multiple quality attributes, the techniques available for dependability assessment are very diverse. Following, some of the existing techniques used for the quantitative assessment of dependability are introduced, focusing on the ones more related to the work that was performed:

- **Fault injection** – used to evaluate the system's fault tolerance mechanisms. Consists in introducing artificial, but representative, faults in components to evaluate specific fault handling mechanisms and to assess the impact of faults in actual software components [33];

- **Robustness testing** – used to evaluate the system's ability to operate despite abnormalities in its input. It stimulates the software in such a way that triggers internal errors, exposing both programming and design errors [46];

- **Benchmarks** – standard tools that allow the evaluation and comparison of different systems according to certain characteristics. A benchmark defines all the steps and components needed to obtain the values of interest [34];

- **Field measurements** – observations of components in the 'wild' or in production. These observations have the advantage of being realistic [34];

An example of a concrete dependability benchmark based on fault injection was proposed in [47], which targets OLTP systems (DBench-OLTP). The goal of this benchmark was to provide a practical way to assess the performance and dependability features of OLTP systems, focusing on the availability of the system.

DBench-OLTP was defined in five main components: a workload, faultload, measures, procedures and rules, and experimental setup. The workload and some of the measures are inherited from the TPC-C benchmark. The remaining components are explained in the paper. The procedures are simple as a driver system guides all the experiment. The fault load is constituted by a set of operational faults, because of their importance in this environment. The new measures characterize the systems in terms of availability.

To demonstrate the usefulness of the technique, the authors applied the benchmark to a set of ten different systems and compared the results. The results shown the importance of the ideas introduced by the benchmarking as they show that some of the systems with lower performance can present higher availability, providing to the users of the benchmark the opportunity to choose the most adequate metric. The paper concludes with the discussion of the effort necessary to use the benchmark.

2.6. Dependability Assessment of NoSQL databases

When YCSB was developed it was intended to be used to evaluate cloud system's aspects other than performance, such as availability and replication(which directly affect dependability attributes) [26]. However, apart from performance evaluations, there is limited work on evaluating quality attributes of NoSQL databases.

Recently, there has been worked published on the evaluation of NoSQL engines from a quality attribute point of view [37]. However, that is a literature review work, which gathers information about some NoSQL engines' quality attributes from sources such as the engines' websites or works performing evaluations on the databases.

An experimental evaluation of Scale-out Storage Systems (SoSS) based on fault injection is presented in [43]. SoSS scale horizontally and include key-value and document stores (NoSQL databases) among other systems, and are likely to run in the cloud. This work focuses on the impact that anomalies have on data availability. A cluster of VMs running MongoDB was exposed to memory and network faults to simulate cloud anomalies, and their predictability was analysed. The work shows that the injected faults adversely impacted the performance of these systems and proposes an anomaly detection approach to enhance the dependability of SoSS through predictions of cloud anomalies.

3. Research Objectives and Approach

The main goal of this dissertation was to evaluate and compare different NoSQL databases. To evaluate this kind of databases, an experimental methodology based on fault injection was defined. While running a first experimental campaign in order to validate the methodology, the existence of a well-defined experimental procedure that did not change between experiments made evident the need for the creation of a tool – **NoSDep** – to automate the process of assessing the databases.

This chapter presents the main research objectives of this work, as well as the approach used to reach them. Section 3.1 presents the main research goals as well as the reasoning behind them. Section 3.2 explains the proposed approach to reach these objectives, introducing the methodology and experimental approach.

3.1. Research Objectives

As stated before, the main research objective of this dissertation was to perform an experimental assessment of NoSQL databases' dependability. To achieve this main objective, the following three goals were set and achieved:

- *Definition of a methodology*

  A measurement based methodology was defined, which required careful selection of the measures to characterize the databases. To understand how these databases behave under the presence of faults, the methodology was based on fault injection as it has been the premier technique for dependability assessment of various system over the last decades. To ensure the fault-based evaluation is accurate it was necessary to use a representative fault model, as such the fault model consists only of faults the systems may face during normal execution.

- *Development of a tool to implement and automate the methodology*

  After defining the methodology and the experimental procedure, a few experiments were run to validate them. The running of these *ad-hoc* experiments made evident the need for a tool to automate the experimentation process. As stated before, the experimental procedure is the same for every experiment, as such what the tool does is implement this procedure in an automated way. The development of the tool was the second objective to achieve, as the tool is what made running the experimental campaign possible.

- *Demonstration of the tool and methodology*

  The last objective that was achieved in this dissertation was the demonstration of the tool and methodology by running an experimental campaign. This campaign validated the methodology and proved that the developed tool was correctly implementing the experimental procedure. To perform this demonstration a set of representative engines was selected, and a set of experiments was defined. The ensuing step was the performance of the experimental campaign and analysis of the results.

3.2. Approach

As was shown in subsection 2.4.1, dependability is a meta-attribute which encompasses several attributes. In the context of NoSQL databases, integrity and availability assume a special importance [45].Thus, the proposed methodology presented in Figure 4 uses fault injection to assess the dependability of YCSB databases, and is focused on availability and data integrity.

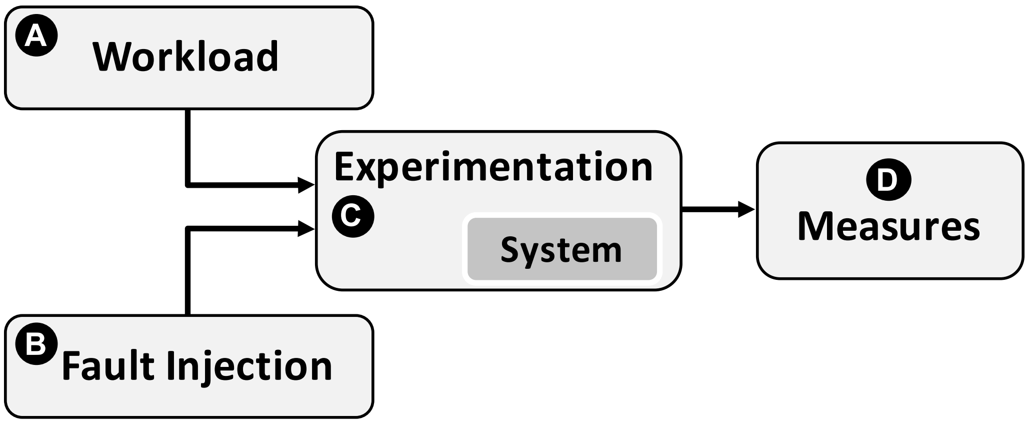

Figure 4: Proposed methodology

The proposed methodology consists in having a prototype of a real system running a NoSQL database which will have a representative workload be generated and executed against it. At a defined point in the execution of the workload, faults will be injected onto the system. To understand how well the databases are capable of tolerating these faults, several measurements are taken during the execution of the workload and analysed after the execution finishes.

In the ensuing subsections each element of the methodology is explained in detail.

3.2.1. Workload

To have a representative evaluation, it is necessary to have the engines working under a realistic load. To ensure this, YCSB was used as the workload generator. As was seen in section 2.3, YCSB already comes bundled with a set of different workloads which can be considered representative of real world applications. However, none of the core workloads is insert-mostly.

It was also explained that YCSB is an extensible workload generator, as such, the following workload was defined to be used in the evaluations of the databases:

- **workloadL** (100% inserts) is an insert-only workload and an application example is a data loading process.

The reasoning behind the creation of this workload is that data integrity can only be verified in workloads that contain either inserts or updates, therefore an insert-only workload is likely to be more prone to data integrity problems when faults affect the system.

Due to the need to use representative workloads in the experiments, no other workloads were added to YCSB during this dissertation. While workloads can easily be added, it should be kept in mind that workloads without insert or update operations are unlikely to detect data integrity problems.

3.2.2. Fault Injection

As was explained in section 2.5, fault injection is a technique used to evaluate a system's fault tolerance mechanisms. For these faults to be considered representative, they must emulate real faults the system may face during normal operation.

The proposed approach was designed to be generic in terms of the faults to be used, thus being able to include faults from one or more of the following types:

- **Operator faults:** correspond to administrator mistakes [38, 47]. the complexity and frequency of management and tuning tasks leads to the prevalence of this type of faults.
- **Software faults:** also known as software bugs, are one of the main causes of system failures [34]. Several techniques allow to emulate software faults in a representative fashion [34].
- **Hardware faults:** although the reliability of hardware increased a lot, high level hardware failures such disk failures can have catastrophic consequences on data management systems [38, 47].

With the current levels of hardware reliability, operator and software faults are nowadays considered the most frequent causes of computer failures [33, 47].

The functioning of NoSQL databases and their underlying machines is ensured by system administrators. It is a human activity, and thus subject to mistakes. System administrator mistakes are basically operator faults, and they can be easily emulated by reproducing those mistakes [45].

3.2.3. Experimentation

The experimentation process always follows the following procedure:

1. **Start** the System Under Test (SUT);
2. Execute the **loading phase**
3. Start the **transaction phase**
4. **Inject the fault** at a predetermined instant
5. Wait for a **detection period** before restarting the engine
6. Wait for the transaction phase to conclude
7. Analyse the data.

Each assessment always starts with a clean start of the database to evaluate. The next step is the execution of a loading phase which is used to both populate and warm up the database. After the loading, the transaction phase begins. This transaction phase is where the fault is injected into the system, at a predetermined time. After injection of the fault there is a detection period, followed by a restart of the database if needed. Lastly, the data regarding the experiment is analysed as soon as the transaction phase finishes its execution.

3.2.4. Measures

During the experimentation process several measurements are performed in order to evaluate the database. Each of the collected measures is explained below:

- **Matching records** – Number of records confirmed by the database still present after injection of the fault
- **Missing records** – Number of records confirmed by the database that went missing after injection of the fault
- **Outdated records** – Number of records confirmed by the database that are missing update operations
- **Extraneous records** – Number of records present in the database whose insert operation was reported as having failed.
- **Recovery Time** – Time the database requires to recover after detection of the fault
- **Pre-fault Throughput** – Number of operations per second before injection of the fault
- **Post-fault Throughput** – Number of operations per second after injection of the fault

Chapter 4 contains more information on how the measures are collected in the current implmentation.

These measures are combined in the metrics presented next to characterize three different attributes: **Data integrity**, **Availability**, **Throughput**.

3.2.5. Metrics

Three metrics were computed from the measures present in the previous subsection.

The first metric portrays the impact of the faults in the **Data Integrity (DI)**. It is defined as seen in (1). The metric is a ratio between the number of legitimate records and the number of expected records. It focuses on the engine's ability to recover from faults, with the minimum amount of data loss. The `expected records` correspond to all the records that were confirmed by the NoSQL engine. The `legitimate records` correspond to the number of matching records minus the extraneous records. The extraneous records are subtracted from the matching records because they were never confirmed to the client. Thus, they should penalize the engine. However, they cannot be put on the denominator as they can not be considered expected records.

The measurements necessary to calculate this metric are the most complex to collect.

Subsection 4.6.1 explains the data integrity verification process in-depth.

$$DI = \frac{\#Legitimate\_records}{\#Expected\_records} \equiv \tag{1a}$$

$$DI = \frac{matching - extraneous}{matching + outdated + missing} \tag{1b}$$

The second metric that was defined is the **Recovery Time (RT)** and it is related to the system's availability. The availability of a system can be given by the formula present in (2a) [34]. Following the methodology proposed in this dissertation it is not possible to calculate the availability of the database, as there is no way to calculate the Mean Time Between Failures (MTBF). However, data to compute the **Recovery Time (RT)** is available and it influences availability according to (2b).

$$availability = \frac{MTBF}{MTBF + MTTR} \tag{2a}$$

$$MTTR = DT + RT \tag{2b}$$

Lastly, even though the work is focused on dependability assessment, it is still important to understand how the presence of faults impact the performance of NoSQL databases. As such, to portray the impact the faults have on the throughput of these engines, the metric **Impact on the Throughput (IT)** that is defined based on the throughput values before and after the injection of the fault is proposed:

$$IT = \frac{TP(pre\_fault)}{TP(post\_fault)} \tag{3}$$

This metric is a ratio between the throughputs pre- and post-fault and a value greater than one implies a degradation on the throughput value after the injection of the fault.

This page is intentionally left blank.

4. NoSDep - A Dependability Assessment Tool for NoSQL

NoSDep (NoSQL Dependability) is a tool to automate the evaluation of NoSQL databases. This tool consists of several components, each responsible for implementing a different step of the methodology presented in section 3.2. Figure 5 presents a diagram with all of these components. It should be noted that while the diagram is based on UML's component diagram, it does not follow UML's conventions. As such, to properly read the diagram the following information should be taken into account:

- A component box indicates a component
- A component box inside another component means that component is part of the larger one
- A **dashed line** indicates that the component towards which the arrow is pointing is used by the first component
- A **solid line** indicates communication between the involved components

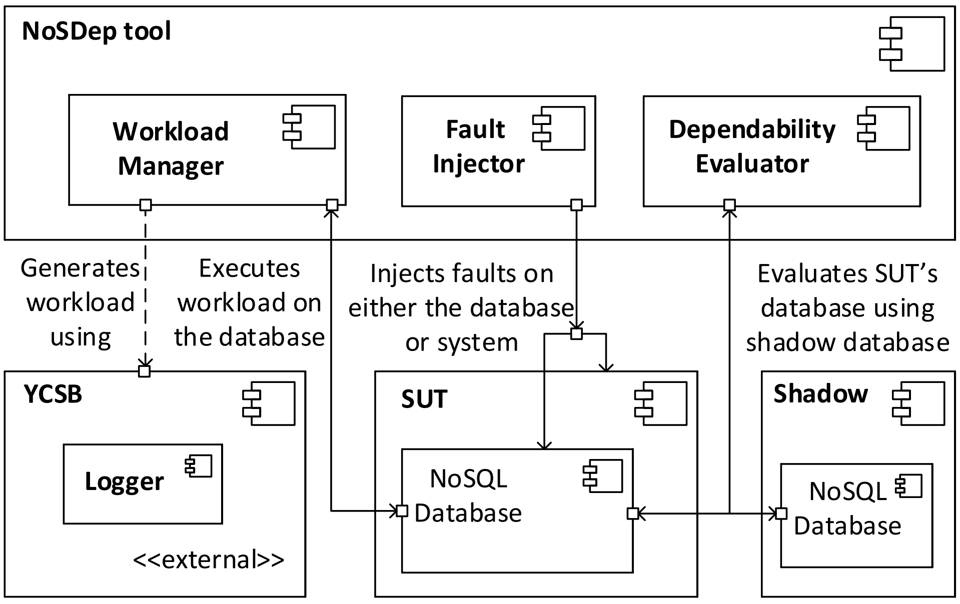

Figure 5: NoSDep's components.

Looking at figure 5 it is possible to observe that the main component, the NoSDep tool, has three smaller components. These smaller components can be directly matched with the proposed methodology:

- The **workload manager** uses an external component (YCSB) to generate and execute a workload against theSUT database
- The **fault injector** is responsible for implementing and coordinating the injection of the faults targeting either the SUT database or the machine it is running on

- The **dependability evaluator** is the component responsible for evaluating the dependability of the database being tested. It uses the measurements collected during the workload execution, in order to compute the metrics defined in 3.2.5

Besides the Main component and its three subcomponents, the tool also uses three external components:

- **YCSB** is the component responsible for generating and executing the workloads. It contains a subcomponent, the logger
  - The **logger** is responsible for storing all information related to the execution of the workloads in a log file
- The **SUT** represents the system being tested. It contains the NoSQL database which is being evaluated
- The **Shadow** component represents the shadow database, which is built from the successful operations present in the log file, and used to evaluate the integrity of the SUT database

To better understand the functioning of the tool, each of the components is explained in the ensuing sections. Section 4.6 is particularly important, as it explains how the component calculates each of the metrics used to evaluate the databases. Following the explanation of the components, the last section contains the list of prerequisites necessary to run the NoSDep tool.

4.1. NoSDep

In subsection 3.2.3 the experimental procedure proposed to evaluate NoSQL databases was presented. What NoSDep does is provide an automated implementation of this procedure.

Figure 6 contains a graphical representation of this implementation.

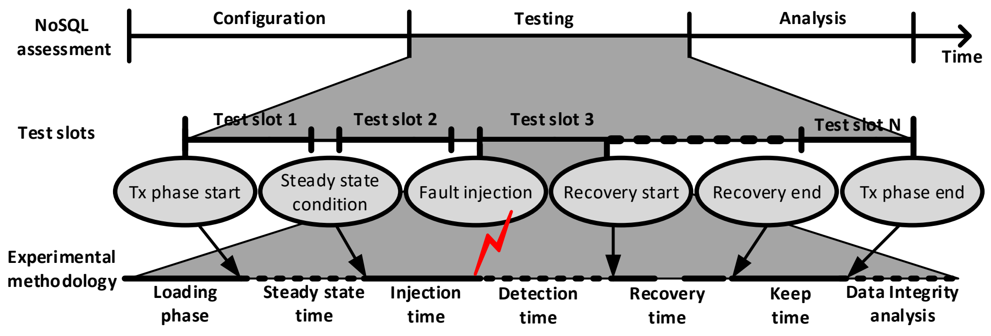

Figure 6: NoSDep's experimental procedure.

In the figure it is possible to see how NoSDep handles the evaluation of the databases.

The user begins by introducing the necessary configurations, such as:

- Database to assess, i.e. select which database from the list of supported databases is going to be assessed

- Information required to connect to the database, i.e. ip and port of the machine where the database is running
- Information about the workload to execute, i.e. which workload to execute and any additional workload configurations (4.2)

The configurations are followed by the testing period, which is composed by several test slots. Each test slot is an implementation of the experimental procedure presented in 3.2.3. For the results of the evaluation to be considered representative, it is necessary to test each engine several times, as such, the higher the amount of test slots, the better the results will be.

Each test slot begins with a clean state of the database, i.e. the database starts only with the necessary configurations, with no data either in memory or disk. NoSDep ensures this clean state by performing the following actions between test slots:

1. Deleting all data and tables in the databases and shutting down the databases
2. Deleting all of the database's data files still present in the disk
3. Creating the tables again
4. Restarting the machines running the database
5. Starting the database

The last step of the evaluation is the analysis of the results. At the end of the execution of all the test slots, the user can check all the information regarding the evaluations, including the values of each of the metrics presented in subsection 3.2.5. These values take into consideration the average results of all test slots.

The NoSDep main component is responsible for controlling the execution of the tests. It receives the necessary configurations from the user, and makes use of its several components to implement the experimental procedure:

1. It **starts the System Under Test (SUT)** by connecting to the machines running the databases and starting them. This connection is done via *ssh*
2. It executes the loading phase using the Workload Manager (WM) component, and starts the transaction phase using the same component
3. It times the right instant to **inject the fault** and uses the **Fault Injector** component to inject it
4. After the injection, it **waits for a detection period** before restarting the engine (if needed)
5. After the transaction finishes executing, it uses the Dependability Evaluator (DE) component to analyse the results
6. It assures the next test slot begins with a clean slate, using the steps previously enumerated
7. Finally, after the last test slot, it presents the results of the assessment to the user

4.2. Workload Manager

The Workload Manager (WM) is the component responsible for managing the generation and execution of the workload, which is performed by the YCSB component. This component contains all the property and workload configuration files required by the YCSB. It also contains a script which receives three parameters:

- The database to evaluate
- The workload to be executed
- The property file to be used

While some databases have more than 1 driver available, at this moment the tool is defaulting to the most popular synchronous driver available for that database. In the future specifying the driver will also be possible.

The available workloads have already been presented in sections 2.3 and 3.2.1. The WM keeps a a different parameter file for each workload, which is used by YCSB in order to generate it. This property file contains the necessary information for YCSB to connect to the database, as well as some additional (optional) runtime parameters. Below is a list of the tool's currently supported parameters:

- **hosts** – IP address of the machine(s) hosting the target database(s)
- **port** – Port to be used by YCSB
- **recordcount** – Number of records to be insert into the database during the loading phase (default:500000)
- **operationcount** – Number of operations to be performed during the transaction phase (default:500000)
- **threads** – number of YCSB client threads (default:10)

It should be noted that the hosts and port parameters have to be defined to evaluate the databases, whereas the other parameters have default values which will be used if the NoSDep user does not specify them.

In order to store the desired configurations in the property files this component exposes a simple Java API to the main component, which can be used to access and/or change the values of the parameters.

4.3. YCSB and Logger

The YCSB component is an external component which is used to generate and execute the workloads. YCSB was introduced in subsection 2.3, and the previous section discussed the necessary configurations to run it. The Logger component is an addition to YCSB responsible for logging all operations executed against the evaluated databases.

Apart from the addition of a new workload (workloadL, presented in 3.2.1)there was only one change to the base YCSB tool: the addition of the Logger module.

The LG module is a simple Java module which was implemented on top of YCSB and is responsible for logging the following information for each operation being executed against the database:

- The timestamp (in nanoseconds) of the operation
- The success status of the operation: either SUCCESS or FAILURE
- The key of the record being affected by the operation
- The fields associated with the record, if the operation is either an insert or update

To ensure YCSB's representativeness as a workload generator is not affected, the LG module does not interfere with the way YCSB generates or its executes the workload. The only thing it does is log the information related to the execution of each operation **immediately after** a response is received from the database.

The timestamps are calculate using Java's "*System.nanoTime()*" method. The nanoTime method returns a "free-running" time in nanoseconds, which is useful for comparison with other nanoTime values. NanoTime uses the highest resolution clock available on the platform, and returns values in nanoseconds [32].

4.4. Fault Injector

The fault injector is the component responsible for the injection of the faults.

Table 2 presents the current fault model proposed for the evaluation of NoSQL databases. The fault model is composed of operator faults, however it could be expanded to include other types of fault as well.

Table 2: Proposed fault model

`CRE`	Clean Restart Engine
`FRE`	Force Restart Engine
`CRO`	Clean Restart Operating System
`FRO`	Force Restart Operating System
`PRM`	Power Restart Machine
`UNC`	Unplug network cable
`DDW`	Delete Data Working
`DDI`	Delete Data Idle

As can be observed, most of the faults are based on unintentional shutdowns of the NoSQL engine, Operating System (OS) or machine running the database. The shutdowns of the engine and OS can be either forced or clean. These faults allow for the evaluation of the database's fault tolerance mechanisms responsible for recovering from abrupt termination.

The Unplug Network Cable (UNC) is meant to evaluate how the databases tolerate sudden Internet shortages.

Lastly, the Delete Data (DD) faults are meant to evaluate if the databases have any mechanisms (and how well they work) in place to detect and recover from unexpected removal of the data files on disk, either while working (running the workload) or in an idle state (after execution of the workload).

Regarding the injection of the faults there are two types of fault: the ones that need human intervention, and the fully automated ones. Fully automated faults are ones which don't require the tool's user to do anything after the test has started. On the contrary, the ones requiring human intervention need the user to perform some sort of action during the test (such as physically restarting the machine or unplugging the network cable).

The current version of the tool only supports the Linux OS. However, there are plans to port the tool to other popular operating systems. As such, the list below provides both generic and OS-specific (Linux) instructions on how the tool implements the faults.

- **CRE** – Perform a clean shutdown of the engine – Send SIGTERM signal to the database process
- **CRO** – Perform a clean shutdown of the Operating System – Use the command "*shutdown -r 0* " Which sends a SIGTERM signal to all running processes and is expected to give them enough time to flush all dirty data to disk before restarting.
- **FRE** – Perform a forced shutdown of the engine – Send SIGKILL signal to the database process
- **FRO** – Perform a forced restart of the Operating System – Use the command "reboot -f -n" which forces immediate reboot of the system and doesnt not contact the init sytem
- **PRM** – Perform a physical restart of the machine running the database – Press the restart button on the computer at the right time
- **UNC** – Unplug the network cable from the computer – Unplug the network cable from the computer at the right time and wait 30 seconds before plugging it back in.
- **DDW** – Delete data files while working – (Not Yet Implemented (NYI)) Goes to the folder containing the data files of the database and deletes some of them **while** the workload is executing
- **DDI** – Delete data files while idle – (NYI) Goes to the folder containing the data files of the database and deletes some of them **after** the workload executes

During tests with the PRM or UNC faults, the fault injector component doesn't do anything (as it is the user that physically injects the faults). From the tool's standpoint, those tests are almost the same as performing clean runs, with the only difference being that the tool gives a sound alert 10 seconds before the faults need to be injected (and 10 seconds before the network cable needs to be plugged back in in the case of the UNC fault).

4.5. SUT

The System Under Test (SUT) consists in a NoSQL database, fully configured to receive requests of a workload and whose dependability is being evaluated, running on a single computer or cluster. During our experiments, the NoSQL database is the only software running on the machine, to provide us with a controled environment.

During the execution of the tests the database starts automatically with the booting of the OS and stays up until the end of the test (with the only downtime being due to the injected faults). During this same period, the computer(s) running the database are only running the database, with no other processes running.

In this work, building this component consisted in formatting a computer and clean installing an OS, installing three different NoSQL engines in it and configuring each database to be able to run YCSB's workload.

4.6. Dependability Evaluator

The Dependability Evaluator (DE) is the component responsible for using the measurements available in the log file, which was created by the LG component, to compute the necessary metrics in order to assess the databases.

The component was implemented in Java and the implementation details regarding the computation of each metric are presented and explained in the following subsections.

4.6.1. Data Integrity

Not only is the Data Integrity (DI) metric the most important of the three, its measurements are the hardest to collect. The enumeration below presents the procedure to collect the measurements required to compute this metric.

1. During the workload, the **client** stores all operations in a safe **log**, **immediately after** confirmation by the server;
2. After the workload, the log is used to build a **shadow database** which contains all successful operations confirmed by the SUT database;
3. The shadow and SUT databases are **compared**, counting the number of:

matching – key-value pairs that are equal in both databases;

outdated – keys in both databases, but with different values;

missing – keys present only in the shadow database;

extraneous – keys present only in the SUT database.

After collecting these measures, the formula (1) presented in subsection 3.2.5 is used to calculate the DI value.

A few more implementation details are presented below:

- When building the shadow database, the DE component goes through each operation in the log file, from first to last. It performs the ones marked as success and skips the ones marked as fail
- The DE component uses the same Java driver used by YCSB to connect to the databases. However, unlike YCSB which was configured to use synchronous drivers/methods in the tests, it uses asynchronous methods to create the shadow database. This was designed like this for two reasons:
  - **Performance** – Asynchronous methods of data insertion are generally faster as they don't block waiting for a response from the database
  - **Controlled environment** – Asynchronous data transfer can be used because in a controlled environment all operations are expected to succeed
- While comparing the databases, the DE goes through every single record on the shadow database and searches for its key in the SUT database
- If it finds matching keys, it compares every single field of the records to see if they all match. Only 100% matches count towards matching records

- After comparing the two databases and collecting the necessary measures, the DE component calculates the DI value for the database using the previously defined metric

4.6.2. Recovery Time

Subsection 3.2.5 presented the formula to calculate a system's availability in equation (2a). It also explained that with the data which can be collected using the proposed methodology only the Recovery Time (RT) can be calculated.

Figure 7 shows the highlights procedure's part related to the recovery of the database.

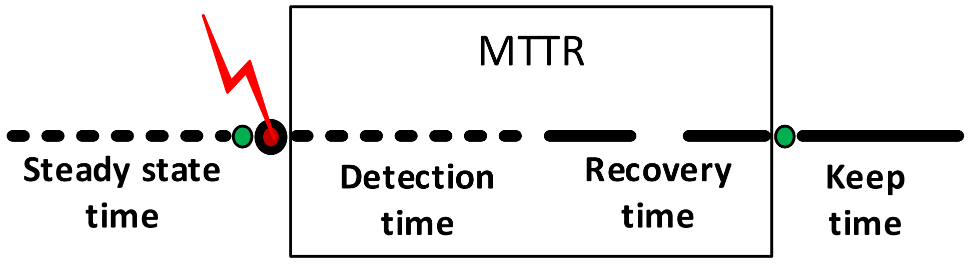

Figure 7: Mean time to recover

As can be seen in the figure, at some point in time a fault is injected into the system. After the injection of the fault there is a detection time and a recovery time before the keep time which represent normal database functioning. The balls filled with green represent the last correct operation before the injection of the fault and the first correct operations after recovery of the database.

The process to calculate the database's RT is pretty simple. The time that goes from the last confirmed operation before the first failure (first green ball) to the first confirmed operation after the last failure (second green ball) is equal to the database's downtime.

With this information, it is possible to observe that (DT) + RT is equal to the difference in time between the first correct operation after failure and the last correct operation before failure. As was shown before, the log file contains the timestamp of each operation. Not only that, the detection time is a fixed and known value in each experiment. As such, the following formula represents how the RT value is calculated:

$$RT = Timestamp\_first\_correct - Timestamp\_last\_correct - DT \tag{4}$$

The validity of the formula is dependent on there being only one fault injection point and problems only occurring inside the MTTR period. As such, the following steps were taken to ensure this validity:

- The number of fault injection points is controlled by the tool.
- The tool looks for problems outside of the MTTR during each experiment.

During the experimental campaign performed during this work no problems were found outside the MTTR period.

4.6.3. Impact on the Throughput

The simplest metric to implement was the Impact on the Throughput (IT). To calculate the throughput of a database the procedure is as simple as counting the number of operations performed during a period of time and dividing that number by the time period.

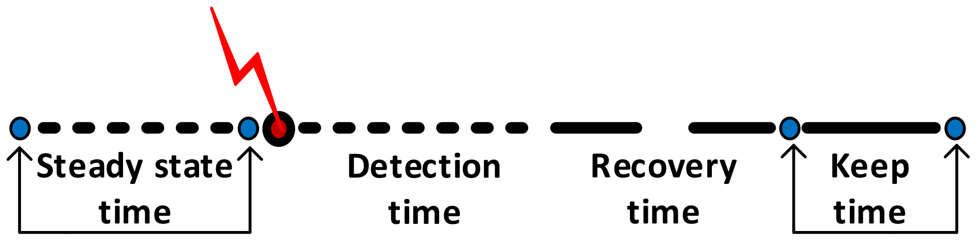

Figure 8: Pre- and Post-Fault throughputs

As such, as can be seen with the help of figure 8, calculating the throughput value pre-fault consists in counting the number of operations between the first confirmed operation and the last confirmed operations before the injection of the fault. Similarly, the post-fault throughput consists in calculating the throughput between the first correct operation after the injection of the fault and the last operation of the workload.

Having these two throughput values, to calculate the IT metric the tool only has to divide them.

It should be noted that to calculate the pre-fault throughput only the operations performed during the transaction phase were considered. The reasoning behind this is that the loading phase is to be used as a warm-up phase to the database. As such, throughput calculations only begin at this steady state time. When using the NoSDep tool The loading phase throughput is still shown to the user, however it is calculated by YCSB, not NoSDep.

This page is intentionally left blank.

5. Experimental Campaign

The main purpose of this experimental campaign was the demonstration of both the methodology which was defined to assess NoSQL databases, and the tool which provides an automated implementation of the methodology.

This way, during the experimental campaign the developed tool was used to follow the methodology in order to evaluate three of the most used NoSQL engines on how well they behave in the presence of faults. Particular focus was given to the attributes: **Availability** and **Integrity**.

In practice, the experimental campaign tried to answer the following research questions:

- What is the impact of the injected faults in the integrity of the data?
- What is the amount of time required by the engine to recover from a failure caused by the faults?
- What is the impact of the faults on the throughput of the engines?
- How heavily does the type of workload being executed affect the availability and data integrity of the engines? Does it also affect the engine's performance?
- How heavily does the injection point of the fault affect its impact?
- What are the differences between faults targeting the Operating System (OS) or the machine?

5.1. Experimental Setup

As can be seen in figure 9, three machines were used during the experimental campaign.

NoSDep represents a machine running the NoSDep tool, which is responsible for controlling the execution of the tests. Even though it is running on a dual core machine with 4GB of RAM, it does not create a bottleneck during the tests, as it is only responsible for coordinating the other machines and injecting the faults, which is not computationally intensive.

The YCSB machine is running a single instance of the YCSB client. It is responsible for emulating the client applications using the system, and also for taking the necessary measurements. It is running on a dual core machine with 8 GB of RAM

The SUT represents a NoSQL database fully configured to run the workload and whose dependability is being evaluated. All of the databases were running on a Ubuntu Server 14.04.3 LTS 64bit installation, on a machine with a 3.50GHz Dual-core CPU, 16GB of RAM, and a solid-state disk (SSD) for the OS and database installations, as well as a 7200 rpm Hard Disk Drive (HDD) to store the data.

The three machines are connected to each other through fast Ethernet.

When selecting which databases to use on the test bed, several criteria was taken into acount:

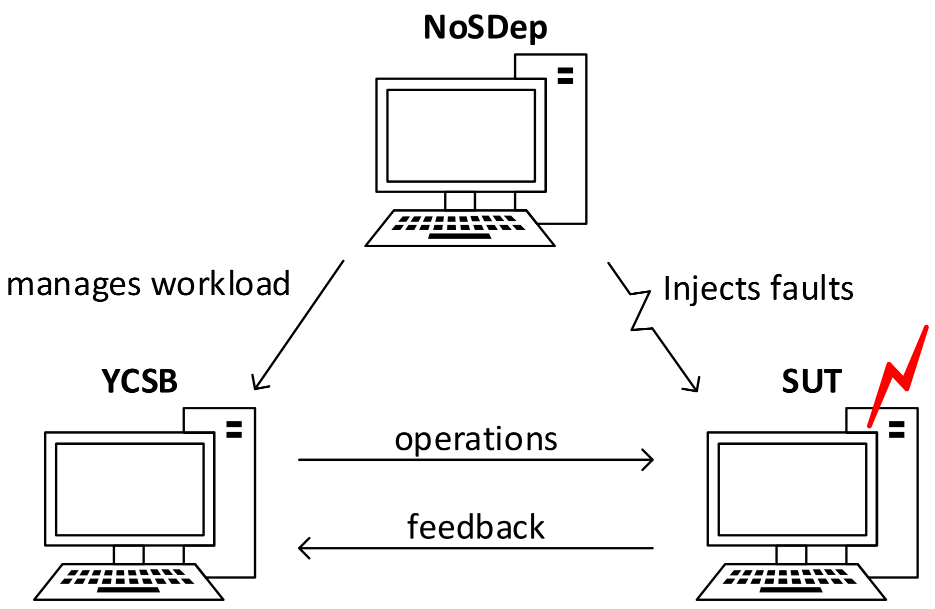

Figure 9: Setup used in the experimental campaign.

- Popularity of the database – The popularity of the database was one of the most important criterion. If a database is not popular, interest in a study evaluating it will generally be quite low
- Existence of a YCSB driver – Even though YCSB is extensible, priority was given to databases which already have a driver developed. This was done ir order to avoid having to spend time implmenting a new database driver
- Type of the database – As was seen in section 2.1, NoSQL databases can be divided in four main types. The selected databases are all of different types.
- Known trade-offs – The CAP theorem was presented before. When selecting which databases to use, priority was given to databases favouring different attributes

The databases which were evaluated during the experimental campaign are presented in Table 3. They all have different *types*, favour different CAP attributes, and all of them had YCSB interfaces already available. This diversity adds value to their evaluation.

Regarding configuration, all the databases were configured to run in a single node setup. Default configurations were used whenever it was possible, and the following dependability-friendly options were adopted:

- In **MongoDB**, journaling was used with its default sync value of 100ms. The journal is an append-only file where all operations performed are written, annotating which were already flushed to disk. In the case of a crash, the journal is used to replay operations that were not flushed yet
- **Cassandra** uses a commitlog which was configured with its default *periodic* mode. In this mode, the commitlog is fsync'd to disk every sync_period (the default 10000ms was used). As such, all data since the last fsync can be lost in the case of a crash

- In **Redis**, its Append-only File (AOF) was used with an every second policy (fsync's the AOF approx. every 1000ms). The AOF logs every write operation received by the server, which will be played again at server startup to rebuild the dataset.

Table 3: Details of the evaluated NoSQL databases.

Name	**MongoDB** [1]	**Cassandra** [5]	**Redis** [2]
Version	v3.2.1. with `MMAPv1 storage`	v2.2.4	v3.0.5
Type	Doc. Oriented	Column Based	Key-Value
CAP	CP	AP/CP	CP
YCSB driver	mongo-java-driver v3.0.3	cassandra-driver-core v2.1.8	jedis v2.0.0
Fault tolerance	Journaling with 100ms commit Interval	Periodic commitlog sync 10000ms period	AOF enabled everysec (1000ms) configuration

5.2. Experimental procedure

The assessment procedure is composed by several test slots, as depicted in Fig. 6. A test slot is a measurement interval during which a workload is executed in a database and one fault is injected.

`3` different instants were defined to inject the fault: 25%, 50% and 75% of the workload duration. This allows us to study the different impacts on the data integrity and the time necessary to recover. The time corresponding to each of these instants was calculated earlier, using profiling runs.

In the case of faults targeting the engine, there is a fixed detection period of `30s`, corresponding to an approximation of the time necessary for the OS to restart. Thus, it is enough time for the OS to stabilize after the engine shutdowns. For faults targeting the OS, since the engines are always started with the OS, the detection time was considered `0s`. The detection time was always subtracted from the analysis of the recovery time, thus, it was left out of the analysis.

The total number of test slots corresponds to: `w*f*n*p*5`, where w is the number of different workloads used, f is to the number of faults, n is the number of databases evaluated and p is the number of injection instants. Each test slot was repeated `5` times, for statistical reasons.

This way, the experimental campaign consisted of a total of `810` test slots: 3 workloads*6 faults*3 databases*3 instants*5 repetitions.

5.3. Results

This section provides an overview of the results. The full results from the experimental campaign can be checked at: `https://eden.dei.uc.pt/~lfav/nosdep/`

Table 4 presents an overview of the results obtained during the campaign for all the faults excluding the Unplug Network Cable (UNC). The results for the injection instants and repetitions were aggregated, so each of the presented rows corresponds to a sum or an average of 15 independent test slots.

Table 4: Overview of the results obtained during the experimental campaign.

W_Fault	Engine	#Issues	#Recs	matching	outdated	missing	extra.	Total Time	RT	TP-Pre	TP-Post
	MongoDB	1 (of 15)	500000	499999.93	0.07	0.00	0.00	00:14:57	0.46	937.04	901.85
A_CRE	Cassandra	4 (of 15)	500000	499999.73	0.27	0.00	0.00	00:17:31	33.15	903.33	859.68
	Redis	1 (of 15)	500000	499999.93	0.07	0.00	0.00	00:12:26	5.19	1,353.31	1,356.26
	MongoDB	13 (of 15)	500000	499983.60	16.40	0.00	0.00	00:15:26	24.75	949.46	880.35
A_FRE	Cassandra	15 (of 15)	500000	498912.13	1087.87	0.00	0.00	00:17:34	33.43	901.02	856.94
	Redis	2 (of 15)	500000	499999.87	0.13	0.00	0.00	00:12:17	4.39	1,361.19	1,370.50
	MongoDB	0 (of 15)	500000	500000.00	0.00	0.00	0.00	00:17:30	31.42	908.80	901.85
A_CRO	Cassandra	1 (of 15)	500000	499999.93	0.07	0.00	0.00	00:18:09	46.22	896.85	791.00
	Redis	0 (of 15)	500000	500000.00	0.00	0.00	0.00	00:13:13	33.92	1,341.61	1,408.51
	MongoDB	15 (of 15)	500000	499978.73	21.27	0.00	0.00	00:18:34	109.47	920.98	582.23
A_FRO	Cassandra	15 (of 15)	500000	499001.60	998.40	0.00	0.00	00:18:19	41.33	894.26	789.67
	Redis	15 (of 15)	500000	499844.87	155.13	0.00	0.00	00:12:58	30.58	1,363.80	1,437.41
	MongoDB	15 (of 15)	500000	499984.67	15.33	0.00	0.00	00:20:09	188.14	937.09	552.70
A_PRM	Cassandra	15 (of 15)	500000	498745.07	1254.93	0.00	0.00	00:18:00	43.82	904.48	801.34
	Redis	15 (of 15)	500000	499841.40	158.60	0.00	0.00	00:13:04	31.53	1,351.32	1,419.92
	MongoDB	0 (of 15)	1000000	999998.00	0.00	0.00	0.00	00:11:11	0.56	1,533.71	1,541.01
L_CRE	Cassandra	12 (of 15)	1000000	981018.93	0.00	0.00	0.80	00:16:09	33.15	1,101.20	1,074.39
	Redis	7 (of 15)	1000000	929495.93	0.00	0.00	0.47	00:13:03	6.59	1,228.62	1,243.19
	MongoDB	14 (of 15)	1000000	999913.00	0.00	85.00	0.00	00:11:33	28.07	1,542.61	1,595.73
L_FRE	Cassandra	15 (of 15)	1000000	975838.27	0.00	4616.27	0.00	00:16:15	33.43	1,093.34	1,069.23
	Redis	6 (of 15)	1000000	930181.93	0.00	0.00	0.40	00:12:48	6.40	1,254.77	1,270.36
	MongoDB	0 (of 15)	1000000	999998.87	0.00	0.00	0.00	00:10:57	28.43	1,544.02	1,541.01
L_CRO	Cassandra	10 (of 15)	1000000	987223.40	0.00	0.00	0.67	00:15:47	48.34	1,104.53	1,129.38
	Redis	6 (of 15)	1000000	991955.53	0.00	0.00	0.40	00:13:25	34.06	1,183.13	1,324.25
	MongoDB	15 (of 15)	1000000	999899.00	0.00	99.47	0.00	00:11:27	57.06	1,538.07	1,651.01
L_FRO	Cassandra	15 (of 15)	1000000	994758.13	0.00	2008.60	0.00	00:15:49	41.97	1,103.76	1,127.80
	Redis	15 (of 15)	1000000	994345.67	0.00	659.87	0.00	00:13:25	31.37	1,259.93	1,323.61
	MongoDB	15 (of 15)	1000000	999936.27	0.00	62.13	0.00	00:11:41	57.83	1507.52	1604.25
L_PRM	Cassandra	15 (of 15)	1000000	993448.47	0.00	5105.67	0.00	00:15:49	42.53	1104.50	1133.75
	Redis	15 (of 15)	1000000	996127.20	0.00	659.07	0.00	00:13:41	33.21	1,242.41	1,287.85
	MongoDB	0 (of 15)	500000	500000.00	0.00	0.00	0.00	00:19:39	3.68	942.59	898.59
F_CRE	Cassandra	2 (of 15)	500000	499999.87	0.13	0.00	0.00	00:22:27	33.13	868.24	822.36
	Redis	1 (of 15)	500000	499999.93	0.07	0.00	0.00	00:14:51	4.72	1,476.95	1,481.69
	MongoDB	15 (of 15)	500000	499987.33	12.67	0.00	0.00	00:20:02	25.61	952.65	872.72
F_FRE	Cassandra	15 (of 15)	500000	499271.07	728.93	0.00	0.00	00:22:32	33.45	868.06	819.11
	Redis	0 (of 15)	500000	500000.00	0.00	0.00	0.00	00:14:44	4.76	1,483.23	1,488.84
	MongoDB	0 (of 15)	500000	500000	0.00	0.00	0.00	00:21:28	30.07	932.86	668.27
F_CRO	Cassandra	2 (of 15)	500000	499999.87	0.13	0.00	0.00	00:22:50	44.63	868.39	776.79
	Redis	1 (of 15)	500000	499999.93	0.07	0.00	0.00	00:15:27	32.19	1,478.01	1,527.64
	MongoDB	14 (of 15)	500000	499987.80	12.20	0.00	0.00	00:23:16	125.53	942.19	652.91
F_FRO	Cassandra	15 (of 15)	500000	499492.93	507.07	0.00	0.00	00:23:28	42.78	868.62	776.08
	Redis	15 (of 15)	500000	499842.40	157.60	0.00	0.00	00:15:31	29.83	1473.22	1,521.25
	MongoDB	15 (of 15)	500000	499988.53	11.47	0.00	0.00	00:23:15	139.492	947.95	675.75
F_PRM	Cassandra	15 (of 15)	500000	499313.13	686.87	0.00	0.00	00:23:04	40.91	878.96	792.47
	Redis	15 (of 15)	500000	499911.20	11.47	0.00	0.00	00:15:42	30.40	1,472.71	1,469.70

To properly read the table the following information should taken into account:

- The first two columns show the workload executed, the fault injected, and the database being evaluated
- The #issues column indicates how many tests slots (out of the 15) had data integrity problems
- The #Recs column indicates the number of records which were introduced into the database
- The matching, outdated, missing, and extra. (extraneous) represent the **average** counting for each of these measures which were previously introduced in 4.6.1
- "Total Time" presents the average time necessary to run both the load and transaction phases, while RT presents the average time to recover
- Lastly, TP-pre and TP-post present the throughput value before and after the injection of the fault

Observing these results, the first relevant point is that a large number of issues related to data integrity were identified during the experiments. In fact, from the total of 675 test slots executed, 377 led to at least one problem of data integrity, which corresponds to more than half of the cases ($\approx 55.85\%$).

As expected, the number of problems is much smaller in the "clean" faults than in the "forced" ones. Nevertheless, all of the engines had at least one outdated or missing issue, even during a clean restart of the engine (CRE). In some cases, Cassandra and Redis also finished the process with one extra record that was never confirmed to the client (extraneous).

All of the experiments with the FRM fault, 135 in total, resulted in data integrity issues and in longer times to recover. The experiments with the FRO fault had similar results, with 134 out of 135 also having data integrity issues (only MongoDB was able to avoid losing data in one of the test slots).

These numbers demonstrate that NoSQL databases are not prepared to tolerate this kind of problems. One of the reasons is that "by design" the database's developers trade guarantees of data integrity for higher performance and scalability.

In the experiments with workloadA, no cases of missing or extraneous records were observed. Likewise, in the experiments with workloadL no cases of outdated information were observed. Although workloadA has no inserts in the transaction phase and workloadL has no updates, this was verified to check for any possibility of unexpected problems with the data.

The experiments with workloadF had similar results to the ones using workloadA, the only difference being that on average experiments using workloadF resulted in a lower number of integrity issues. This is due to the fact that while the two workloads have similar operations, both consisting of reads and inserts, workloadF has a lower number of updates.

The results related to the UNC fault can be seen in table 5. The reason for presenting these results in a separate table is that the UNC fault never caused a failed operation on both MongoDB and Cassandra. Without failed operations it is not possible to calculate the Impact on the Throughput (IT) metric, as such, while analysing the data for this fault, a different approach was used to measure the impact of the faults on the throughput:

- While analysing the log file, the timestamps of the operations corresponding to 25%, 50% and 75% of the transaction phase were stored.
- Using these timestamps, alongside the already calculated first and last operations, it was possible to calculate four different throughputs:
  - 0 to 25 % of the transaction phase
  - 25 to 50 % of the transaction phase
  - 50 to 75 % of the transaction phase
  - 75 to 100 % of the transaction phase

Looking at the throughputs in table 5, it is possible to find a correlation between the time period with the lowest throughput and the instant that the fault was injected. For example, looking at the first test slot, where the fault was injected at 25% of the workload, the time period with the lowest throughput was from 25 to 50%.

The most relevant finding regarding this fault type is that, while this fault had one of the lowest amounts of data integrity issues out of all the faults, in the experiments with

Redis and workloadL there was always an extraneous record. It is not clear whether this problem is caused by the client not waiting for an answer or the engine not verifying if the client received its response.

The reason for the Recovery Time (RT) not being present in table 5 is that since the Unplug Network Cable (UNC) fault never led to a system failure, the RT value during these tests was always 0.

The UNC fault had data integrity issues in 26 out of 135 test slots (≈ 19.26%). Moreover, the average number of wrong records (i.e. either outdated, missing, or extraneous) in every test slot using this fault never surpassed 1.

Table 5: Overview of the UNC fault's results obtained during the experimental campaign.

W_Fault	Engine	Inj. Point	#Issues	#Recs	matching	outdated	missing	extra.	TP 0–25	TP 25–50	TP 50–75	TP 75–100
		25%	0 (of 5)	500000	500000	0.00	0.00	0.00	985.85	672.17	875.65	900.85
	MongoDB	50%	1 (of 5)	500000	1000000.00	0.20	0.00	0.00	993.13	888.67	711.40	863.80
		75%	0 (of 5)	500000	499999.93	0.00	0.00	0.00	1000.93	877.61	866.68	746.61
		25%	0 (of 5)	500000	1000000	0.00	0.00	0.00	1135.59	751.21	1093.92	1097.52
A_UNC	Cassandra	50%	1 (of 5)	500000	1000000.00	0.20	0.00	0.00	1135.52	1091.15	749.46	1090.51
		75%	1 (of 5)	500000	499999.93	0.20	0.00	0.00	1134.54	1085.17	749.46	1090.51
		25%	2 (of 5)	500000	1000000	0.40	0.00	0.00	1057.31	1218.71	1139.40	1374.39
	Redis	50%	2 (of 5)	500000	1000000.00	0.40	0.00	0.00	1367.57	1218.71	1139.40	1374.39
		75%	1 (of 5)	500000	499999.93	0.20	0.00	0.00	1358.52	1365.81	979.64	1368.95
		25%	0 (of 5)	500000	1000000	0.00	0.00	0.00	669.16	518.47	603.87	602.81
	MongoDB	50%	0 (of 5)	500000	1000000.00	0.00	0.00	0.00	683.68	605.29	484.70	613.01
		75%	0 (of 5)	500000	499999.93	0.00	0.00	0.00	647.40	617.16	607.62	496.45
		25%	0 (of 5)	500000	1000000	0.00	0.00	0.00	599.62	465.76	578.31	563.39
F_UNC	Cassandra	50%	0 (of 5)	500000	1000000.00	0.00	0.00	0.00	599.13	581.47	467.30	564.43
		75%	0 (of 5)	500000	499999.93	0.00	0.00	0.00	602.24	579.14	580.24	456.33
		25%	0 (of 5)	500000	1000000	0.00	0.00	0.00	1018.86	788.42	1020.36	1024.45
	Redis	50%	2 (of 5)	500000	1000000.00	0.40	0.00	0.00	1014.04	1018.22	785.19	1018.55
		75%	1 (of 5)	500000	499999.93	0.20	0.00	0.00	1028.70	1039.90	1030.94	789.28
		25%	0 (of 5)	1000000	1000000	0.00	0.00	0.00	1052.97	1429.27	1541.69	1542.11
	MongoDB	50%	0 (of 5)	1000000	1000000	0.00	0.00	0.00	1538.24	1184.99	1309.87	1543.37
		75%	0 (of 5)	1000000	1000000	0.00	0.00	0.00	1533.77	1540.38	936.95	1538.37
		25%	0 (of 5)	1000000	1000000	0.00	0.00	0.00	1135.59	751.21	1093.92	1097.52
L_UNC	Cassandra	50%	0 (of 5)	1000000	1000000	0.00	0.00	0.00	1135.52	1091.15	749.46	1090.51
		75%	0 (of 5)	1000000	1000000	0.00	0.00	0.00	1134.54	1085.17	1091.52	834.20
		25%	5 (of 5)	1000000	999991.00	0.00	0.00	0.40	1131.66	1079.62	1293.23	1310.94
	Redis	50%	5 (of 5)	1000000	999990.00	0.00	0.00	0.40	1283.06	1307.70	947.12	1320.67
		75%	5 (of 5)	1000000	999990.80	0.00	0.00	0.20	1264.92	1294.19	1290.55	941.71

5.4. Discussion

Considering the large number of data integrity issues observed, it was imperative to perform a more detailed analysis of the data. No influence of the injection instant (25%, 50%, or 75% of the workload) was detected in the results for data integrity.

Considering the Data Integrity (DI) metric presented in 4.6.1, the summary of the results is presented in Fig. 10. Each set of three columns represent the results achieved by the databases in one workload and one fault type (again, each column corresponds to the average of 15 measurements). As we can observe, the results obtained are in general, very close to 1, which means that the issues are related to small chunks of data. Still, only in 6 cases did the databases achieve perfect integrity (5 times with MongoDB and 1 with Redis).

The results are fairly worse in "forced" faults, for all of the three databases. This is justified by the fact that the database does not have the chance to flush the dirty memory pages to the disk, losing data that represents operations that were already confirmed to the client.

The results show that during these experiments, and in terms of DI, Cassandra achieved the worst results for all of the fault types. In one configuration (L FRE), it averaged only (≈ 0.995%) data integrity. This means that almost 0.5% of the data was either lost or missing updates. Cassandra uses a commit log, that by default is configured for a "periodic sync" of 10sec. This means that up to 10 seconds of operations can be lost. The reasoning behind this is that lower values for the periodic sync may hurt the performance. Cassandra also relies on replication, to mitigate this while keeping good performances.

MongoDB and Redis perform very similarly, except for the "forced" faults. In the case of FRE, Redis performs slightly better, while in the case of FRO, it presents worse results, losing more data. MongoDB relies on journaling while Redis relies on a AOF, which in essence have the same objectives of the commit log. From the results, it is possible to understand that Redis is more vulnerable to failures of the OS than MongoDB. In this case, it is not clear which is the best database, as it depends on the faults that the target system will be exposed to.

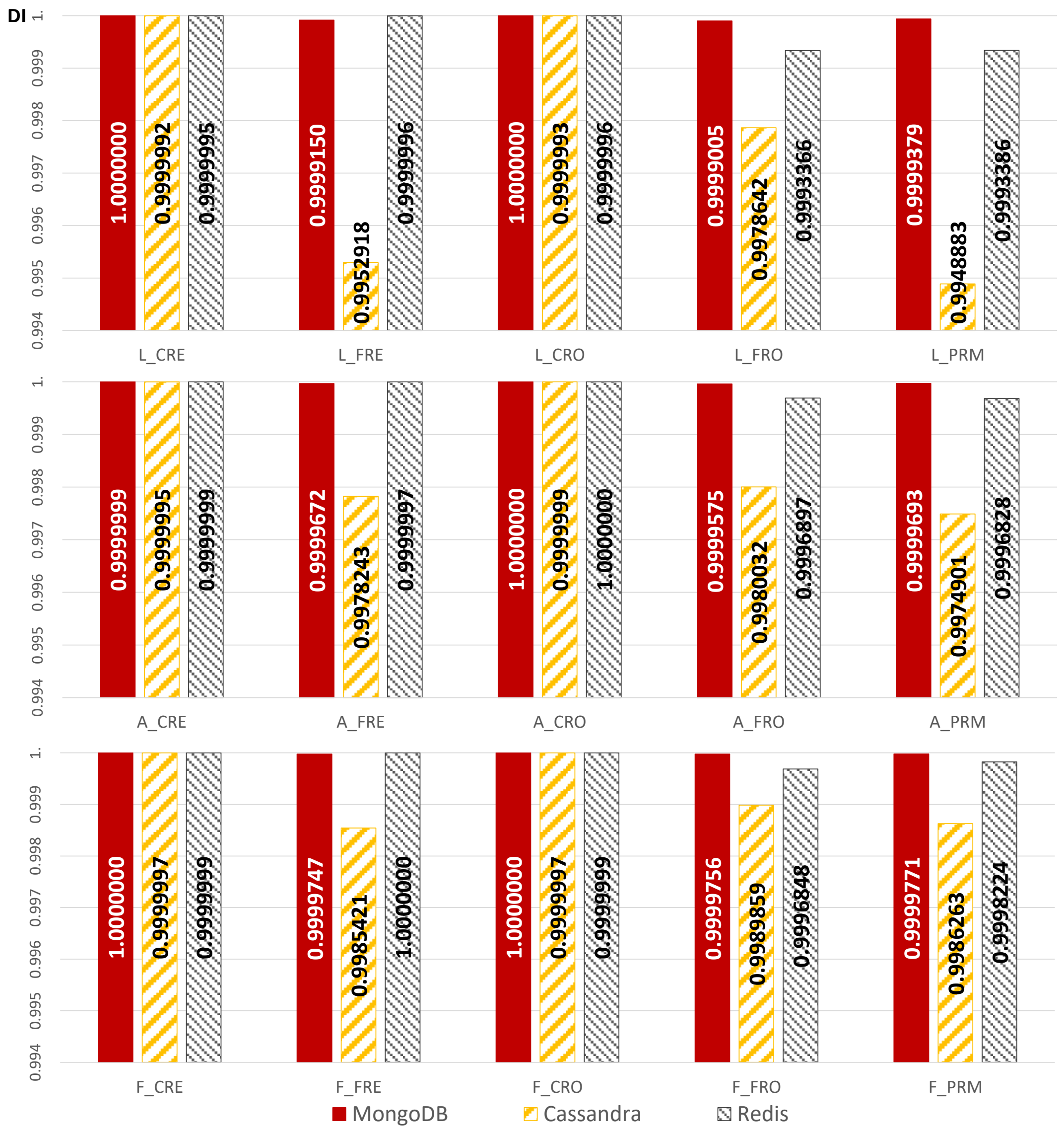

Figure 10: Data Integrity (DI) values for each Workload_FaultType.

The Recovery Time (RT) influences the availability of the system as explained in Section 3.2.4. It is measured by computing the difference between: the instant of the first successful operation after the fault, and the last successful operation before the fault. As mentioned in section 3.2.4 the detection time was left out of the analysis. Table 6 presents the summary of the results for the recovery time of each database. As we can observe, faults affecting the OS have a much higher impact on the time to recover, as expected since the times measured also include the time necessary to boot the OS.

It is interesting to notice that after a clean shutdown of the database, MongoDB is extremely fast to recover. However, after a forced engine or OS shutdown, it takes much longer. This shows that in the cases of clean shutdowns, it is able to store all the information in the disk, in a ready to use format. The results for A_FRO differ from those of the other faults. This workload, together with the force restart, stress the fact that MongoDB is not optimized for reads and updates.

Redis recovers slightly faster than the other engines in the cases of the "forced faults". In the case of "clean" faults, the engine takes some time to shutdown, which influences the results. To recover, Redis always needs to load everything from the AOF. For the experiments with workloadL (prefixed with "L "), it is possible to observe that the time to recover consistently increases as the instant of the fault injection advances. Redis works in-memory, and it is reconstructing the database from its AOF on start. Thus, as there are more records in the database, it takes more time. This would grow even further if workloads with more elements were used. This does not happen with workloadA, because the transaction phase consists only of reads and updates, so no further data is added.

Although Cassandra takes a long to time to recover, it also displayed the most constant times with a very low standard deviation in engine faults.

In general, Redis took the lowest amount of time to recover and Cassandra took the longest. It should be noted though, that unlike in MongoDB and Cassandra, Redis showed a correlation between injection time and time to recover, as such, if we were using a much larger dataset, we believe that Redis would have taken longer to recover whereas the other databases would probably display similar times.

Fig. 11 presents the summary of the results for the impact of the faults in the throughput. Before looking at the results, one might assume that the throughput presents some degradation, mainly in machine restarts.

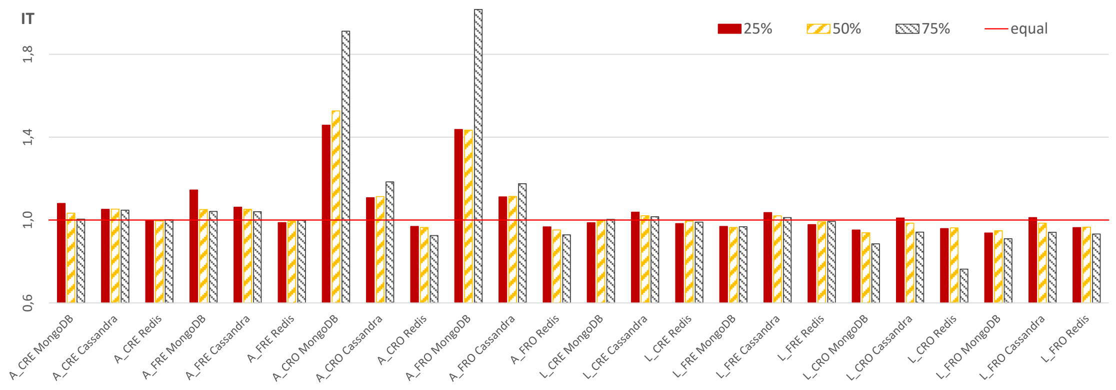

Figure 11: Impact on the throughput (IT): ratio between the throughput before and after the fault injection for each workload, fault, and engine.

As we can observe, two configurations clearly stand out from the remaining: `A_CRO MongoDB` and `A_FRO MongoDB`. In these cases, the throughput becomes much worse after the injection of the fault. They have the particularity of being executed with MongoDB and the faults injected being the ones with the worst impact in that database. Furthermore, the workload used is also the one in which MongoDB is not optimized for, thus explaining such a degradation in performance.

Table 6: Summary of the results for the recovery time.

Workload_Fault	Fault Inj. instant	MongoDB	Cassandra	Redis
L_CRE	25%	0.54	33.15	5.37
	50%	0.59	33.15	6.73
	75%	0.54	33.15	7.67
L_FRE	25%	26.97	33.42	5.29
	50%	28.64	33.42	6.44
	75%	28.59	33.44	7.48
L_CRO	25%	29.60	46.20	33.24
	50%	28.10	49.20	34.36
	75%	27.58	49.92	34.59
L_FRO	25%	57.82	41.14	29.99
	50%	54.64	42.71	31.10
	75%	58.73	42.05	33.00
L_PRM	25%	55.02	44.75	31.44
	50%	58.13	38.72	33.05
	75%	60.36	44.11	35.13
A_CRE	25%	0.54	33.14	5.06
	50%	0.54	33.14	5.48
	75%	0.31	33.17	5.03
A_FRE	25%	18.03	33.43	4.15
	50%	18.87	33.43	4.34
	75%	37.35	33.43	4.66
A_CRO	25%	30.68	43.01	33.52
	50%	31.79	46.22	33.95
	75%	37.79	49.42	34.29
A_FRO	25%	124.95	38.91	30.21
	50%	87.24	41.33	30.48
	75%	116.23	43.76	31.04
A_PRM	25%	169.14	43.76	33.87
	50%	191.49	41.44	29.91
	75%	203.80	46.26	30.81
F_CRE	25%	0.54	33.13	4.26
	50%	9.95	33.13	4.82
	75%	0.54	33.13	5.08
F_FRE	25%	20.75	33.45	4.32
	50%	22.40	33.44	4.86
	75%	33.67	33.46	5.09
F_CRO	25%	30.20	41.41	30.88
	50%	30.50	46.25	33.48
	75%	29.50	46.23	32.21
F_FRO	25%	108.63	42.15	29.34
	50%	140.59	42.99	30.07
	75%	127.37	43.20	30.10
F_PRM	25%	151.55	39.75	29.88
	50%	131.50	40.60	30.20
	75%	136.72	42.39	31.11

Finally, Redis never presents a worse throughput than before the fault. This is due to the AOF based mechanisms of recovery of Redis. This mechanism allows restoring the database without degradation and also has a "warm-up" effect (i.e. the data loaded during the recovery process makes an exploration of the spatial and temporal localities possible). Finally, No patterns regarding the relation between the injection instant and the impact on the throughput were observable.

5.5. Threats to the Validity of the Experiment

There are some threats to the validity of the experimental campaign that must be discussed:

- ***T1*** **Size of the workload** – developed to handle big data, 1 Million operations may be too small to be a representative workload. However, we believe that the size of the workload has limited impact on most of the observations made in this experiment.
- ***T2*** **Single node deployment** – although NoSQL databases were designed for horizontal scaling, these experiments with a single node provided insight on the behaviour of the databases and served to validate the methodology. Furthermore, many organizations use single node deployments of these databases.
- ***T3*** **Representativeness of the fault model** – due to the time required for the experiments, it was not possible to cover a very large range of faults. However, operator faults are still very important in the context of data management, and the experiments showed that they can lead to dangerous failures.

6. Conclusions and Future Work

This work proposes a methodology based on fault injection to assess and compare NoSQL databases in terms of dependability attributes, with particular focus on availability and data integrity. A tool, named **NoSDep** was developed to implement the methodology and ease its use.

The tool and the methodology were applied in an experimental campaign that evaluated 3 leading NoSQL database solutions according to their dependability attributes. The results showed the applicability of the methodology and also confirmed the usefulness of such a tool.

From the point of view of NoSQL databases, the **results clearly showed that the benefits of NoSQL databases do not come without an associated cost**. In fact, the evaluated databases are very limited in guaranteeing the integrity of the data. During the experiments, more than half of the tests executed resulted in a state that was inconsistent with what was confirmed to the client. Additionally, the **recovery times were in many cases quite significant**, particularly in faults based on forced restarts. Finally, the results show the importance of the users carefully selecting the database that better fits the objectives of their systems.

Future work includes expanding the fault model and performing new experimental campaigns with larger (distributed) setups and larger workloads. It is the author's belief that after these tasks are performed, this work can give way to the **proposal of dependability benchmarks for NoSQL databases**. Such benchmarks will allow the comparison, in a standard way, of different systems and configurations according to dependability attributes. They will also allow the analysis of the trade-offs between performance and dependability.

References

[1] Mongodb. URL https://www.mongodb.org/. Accessed: 2016-08-31.

[2] Redis. URL http://redis.io/. Accessed: 2016-08-31.

[3] Accumulo. URL https://accumulo.apache.org/. Accessed: 2016-08-31.

[4] Aerospike. URL http://www.aerospike.com/. Accessed: 2016-08-31.

[5] Apache cassandra, . URL http://cassandra.apache.org/. Accessed: 2016-08-31.

[6] Facebook's instagram: Making the switch to cassandra from redis, a 75% "insta" savings, . URL http://planetcassandra.org/blog/interview/facebooks-instagram-making-the-switch-to-cassandra-from-redis-a-75-insta-savings/. Accessed: 2016-08-31.

[7] Couchbase, . URL http://www.couchbase.com/. Accessed: 2016-08-31.

[8] Couchdb, . URL http://couchdb.apache.org/. Accessed: 2016-08-31.

[9] DB-Engines Ranking - popularity ranking of database management systems. URL http://db-engines.com/en/ranking.

[10] Dynamodb. URL https://aws.amazon.com/dynamodb. Accessed: 2016-08-31.

[11] Hbase. URL https://hbase.apache.org/. Accessed: 2016-08-31.

[12] Memcached. URL http://www.memcached.org/. Accessed: 2016-08-31.

[13] An elastic metadata store for ebay's media platform. URL https://www.mongodb.com/presentations/elastic-metadata-store-ebay%E2%80%99s-media-platform?c=574f6104b6. Accessed: 2016-08-31.

[14] Neo4j, . URL https://neo4j.com/. Accessed: 2016-08-31.

[15] 7 use cases of neo4j, . URL https://neo4j.com/use-cases/. Accessed: 2016-08-31.

[16] Orientdb. URL http://orientdb.com/orientdb/. Accessed: 2016-08-31.

[17] 8 surprising facts about real dockeradoption. URL https://www.datadoghq.com/docker-adoption/. Accessed: 2016-08-31.

[18] Riak kv. URL http://basho.com/products/riak-kv/. Accessed: 2016-08-31.

[19] Titan. URL http://titan.thinkaurelius.com/. Accessed: 2016-08-31.

[20] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 14–22, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1976-8. doi: 10.1145/2494444.2494447. URL http://doi.acm.org/10.1145/2494444.2494447.

[21] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. *Fundamental concepts of dependability.* University of Newcastle upon Tyne, Computing Science, 2001.

[22] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[23] E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2): 23–29, February 2012. ISSN 0018-9162.

[24] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2): 4:1–4:26, June 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL `http://doi.acm.org/10.1145/1807128.1807152`.

[27] DataStax. Big data challenges. URL `http://www.datastax.com/big-data-challenges`.

[28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294281.

[29] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. ISSN 0360-0300. doi: 10.1145/289.291.

[30] Coda Hale. You can't sacrifice partition tolerance. URL `http://codahale.com/you-cant-sacrifice-partition-tolerance/`.

[31] R. Hecht and S. Jablonski. NoSQL evaluation: A use case oriented survey. In *2011 International Conference on Cloud and Service Computing (CSC)*, pages 336–341, December 2011. doi: 10.1109/CSC.2011.6138544.

[32] David Holmes. Inside the hotspot vm: Clocks, timers and scheduling events - part i - windows. URL `https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks`.

[33] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997. ISSN 0018-9162. doi: 10.1109/2.585157.

[34] Karama Kanoun and Lisa Spainhower. *Dependability Benchmarking for Computer Systems*. John Wiley & Sons, October 2008. ISBN 978-0-470-37083-4.

[35] Martin Kleppmann. Please stop calling databases cp or ap. URL `https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html`.

[36] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2): 12–14, February 2010. ISSN 0018-9162.

[37] João R. Lourenço, Bruno Cabral, Paulo Carreiro, Marco Vieira, and Jorge Bernardino. Choosing the right NoSQL database for the job: a quality attribute evaluation. *Journal Of Big Data*, 2(1):18, August 2015. ISSN 2196-1115.

[38] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it? In *4th USENIX Symp. on Internet Technologies and Systems (USITS'03)*, Seattle WA, USA, 2003.

[39] Stephen Pimentel. The rise of the multimodel database. URL `http://www.infoworld.com/article/2861579/database/the-rise-of-the-multimodel-database.html`.

[40] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008. ISSN 1542-7730. doi: 10.1145/1394127.1394128.

[41] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367512.

[42] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley Professional, 1st edition, 2012. ISBN 0321826620, 9780321826626.

[43] Guthemberg Silvestre, Carla Sauvanaud, Mohamed Kaâniche, and Karama Kanoun. An anomaly detection approach for scale-out storage systems. In *26th International Symposium on Computer Architecture and High Performance Computing*, Paris, France, 2014.

[44] B.G. Tudorica and C. Bucur. A comparison between several NoSQL databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5, June 2011. doi: 10.1109/RoEduNet.2011.5993686.

[45] L. Ventura and N. Antunes. Experimental Assessment of NoSQL engines Dependability. In *12th European Dependable Computing Conference (EDCC 2016)*, Gothenburg, Sweden, 2016.

[46] M. Vieira, N. Laranjeiro, and H. Madeira. Assessing robustness of web-services infrastructures. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 131–136, June 2007.

[47] Marco Vieira and Henrique Madeira. A Dependability Benchmark for OLTP Application Environments. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 742–753, Berlin, Germany, 2003. VLDB Endowment. ISBN 0-12-722442-4.