Bachelor Informatica

University of Amsterdam

Informatica — Universiteit van Amsterdam

Robustness Analysis of Distributed Databases via Fault Injection

Gerard Schröder

June 8, 2016

Supervisor(s): Raphael Poss

Signed: Signees

Abstract

As the importance of reliable data storage increases, various techniques are applied to obtain the reliabilities. The usage of the assumed fault tolerant cloud providers increases, even though not a sufficient amount of research is performed on their fault tolerance. On cloud servers, a distributed Database Management System (DBMS) can be used to achieve safe data storage. Various DBMSs are making claims about its fault tolerance, a measure of *robustness*, they provide. However, the claims are rarely validated. This research targets to validate the fault tolerance of the DBMS. The research scope is limited to test a DBMS on its data integrity, if only original inserted data is returned to the client.

The research focusses on verifying the claimed fault tolerance of the Apache Cassandra DBMS, a popular, fault tolerant column store DBMS. A fault injection framework is proposed and implemented to create a suitable and flexible test environment. Fault injection experiments on the DBMS are performed with and without replication. Overall, Cassandra can be considered fault tolerant. When targeting data files only, most faults are detected immediately and no invalid data is returned to the client. Modified system files did not return an immediate result. When modifying the index files, faults were undetected in the short term analyses of the experiments, resulting in missing data both with and without replication. However, replication did reduce the number of faults returned by Cassandra.

Contents

CHAPTER 1

Introduction

As the importance of data storage increases, costs of data unavailability will increase. Data unavailability was estimated to cost $1 M/hour as of 2001 [15]. Data loss incidents inside U.S. co-operations are estimated to cost $5.4 million dollar on average [21]. For small businesses, data loss has larger consequences as a non-sufficient amount of backup ups are made. The main reason for the data loss is hardware failures along software failures. Different data storage techniques are applied, where cloud storage systems are becoming a more popular solution. However, assumptions are made about cloud environments being a fault tolerant storage environment. Moreover, a limited amount of research is performed about the fault tolerant robustness of the cloud environments.

Cloud-based systems hold a reputation of being reliable and scalable media. One such system is a cloud-ready Database Management System (*DBMS*). Multiple DBMS include mechanisms to ensure the availability of the data and data integrity. However, a limited amount of research is performed on the reliability claims of a DBMS. Often the robustness of cloud environments is only guaranteed in the measures of availability of data and performance. Malfunctioning or broken servers are replaced. When robustness test scenarios are considered, they consist most of the time only of scenarios with a full DBMS node failure. Most of the time the DBMS neglect to include tests when data corruption is introduced due to hardware or software faults, which can introduce faulty data when performing data analysis.

1.1 Problem definition

Hardware failures have the highest chance to introduce data loss [21]. Research is performed about the hardware failure prevention techniques in cloud environments. This thesis researches and validates the robustness of distributed DBMSs as they are a suitable medium to manage large amounts of data using cloud environments. The research verifies and tests if the state-of-the-art DBMSs are providing sufficient mechanisms to detect possible hardware failures. The system validation methods consist of using fault injection techniques to simulate hardware failures. A new proposed framework implements the verification methods using the fault injection techniques.

The Apache Cassandra DBMS is the chosen subject of research [17, 1]. It is a widely used column store cloud DBMS engine, where large clusters are using thousands of Cassandra nodes. The popularity and decentralized structure makes it interesting to research how well its fault tolerance works in a cluster. Moreover, claims are made about its robustness and still have to be validated.

Based on the above observations, the following research questions are formulated:

- How to implement and design a framework providing a suitable DBMS testing environment?

- To which extend will the Apache Cassandra DBMS react to data corruption caused by simulated hardware faults?

1.2 Contributions

Different DBMS properties have to be validated and tested. These validation tests will consist of data integrity, availability and consistency. The implementation and experimentation scope will only be limited to perform test on data integrity. As different definitions of data integrity exists, the following definition is retained here: data integrity is the measure difference between the original data and the current version of the data availability. An example where data integrity would cause problems is when data in a fixed data set changes, thus gets corrupted, causing different results than with the original data set.

This thesis introduces a framework used to create a suitable test environment for distributed DBMS and test its behaviour under hardware failures. Data integrity verification methods and requirements based on existing frameworks will be adapted to create this new testing environment. The fault injection framework is designed to be flexible and to be able to test multiple DBMSs. The framework provides measurements indicating the faults manifested.

The framework is used to get an indication of a robustness measurement of the Apache Cassandra DBMS. The reaction of Cassandra under the fault injections will be tested to validate its fault tolerance claims.

1.3 Outline

Chapter 2 represents the research performed about the fault classifications, origins and prevention techniques. Moreover, fault injection methods are researched to be applied to construct the framework. Chapter 3 describes the DBMS under test, thus the Apache Cassandra database. General information, its fault tolerance mechanisms and its robustness claims are discussed. Chapter 4 describes a design of the framework, including an overview of its components. Furthermore, other design considerations as modularity and the scenarios are discussed. Chapter 5 consists of the implementation and some of the design decisions of the framework. Then, in chapter 6, performed experiments are discussed including the results. It will be concluded with chapter 7 containing the conclusion and future research.

Appendix A provides steps to reproduce the experiment. Appendix B provides information about how a different DBMS can be integrated into the framework.

CHAPTER 2

Related work

This section is divided in several research topics, starting with the classification of the faults followed by their origin, prevention techniques and fault injection techniques.

2.1 Fault classifications

There are numbers of hardware fault classifications, where the most common ones are featured below. As the definition of a fault differs per source, the following interpretation is used as the definition of a fault: a difference between the expected and observed behaviour [30]. A list of different faults occurring when writing or reading data from memory or hard disk is shown below.

1. State faults: When a bit transition (*bit flip*) occurs when the cell is accessed.
2. Transition faults: When a bit stays stuck at its current value when written with a new value.
3. Stuck-at faults: A single bit in, example given, memory or hard disk are continually stuck on the value zero or one.
4. Read disturb faults: After a bit value is read, the bit value unwillingly changes.
5. Write disturb faults: When a bit value is changed into a wrong bit value when written.
6. Incorrect read/write faults: A wrong value is read or written.

2.2 Fault origins

The faults can originate from different hardware levels next to faulty software. Short descriptions about a number of components are covered in this paragraph.

- **Random access memory** (RAM) faults are mainly influenced by cosmic rays, which create soft errors such as bit flips [26]. The error rate of those errors is one error per GB per week.
- **Hard disk drives** (HDDs) store data on the magnetic media accessed by the read/write head [7]. The possible fault factors of a HDD include: i) media imperfections, ii) loose particles creating scratches on the disk surface, iii) high/low fly writes which cause incorrect bit patterns, iv) disk surface vibrations, v) read/write head hitting a bump or media track, vi) off-track reads and writes and vii) different quality classes of hard disks.

  The last mentioned fault factor is observed in a large scale test with enterprise and nearline HDDs. The enterprise disks have had an error rate of 1.9% and the nearline disks about 8.5% over a testing period of three years. The faults increase with age, linear using enterprise disks and a steeper increase with nearline disks.

Errors in the HDD controller, consisting of low-level firmware code can introduce silent data corruption [6]. In this case, no errors are reported to the operating system and the faults can not be repaired or detected by the HDD itself. RAID protection schemes may be unable to detect this.

- **Solid State Drives** (SSD) use non-volatile memory (NAND flash) [10]. When writing to a cell, the cell has to be erased before a new value can be written. When a NAND block is heavily used, the chance of read disturbance faults increases.

  Moreover, writing to neighbour cells can cause interference [11]. The cell-to-cell interference is a well known source of noise when writing to cells and the significance depends on the internal bit line structure of the SSD.

2.3 Fault prevention techniques

To reduce the chance of the faults, methods such as using Redundant Arrays of Inexpensive Disks (RAID) storage and implementing error correcting code.

2.3.1 Redundant Arrays of Inexpensive Disks

RAID is a disk configuration to speed up I/O and providing a fault tolerance mechanism when a disk fails [22]. Different levels of RAID exist, a simple form is described here. When a write operation is performed, the data is written to multiple disks [16]. As the data is written, the data is returned by a read on the disk unless a latent-sector error occurs. When an error occurs, a reconstruction of the data is executed using a parity calculation. Afterwards, a validation scheme is applied to check if the data is valid.

When disk writes are performed, parity is recalculated, which requires disk reads unless the whole disk sector is written. Optimized parity calculation can be performed using subtractive or additive methods. An XOR operation is used to calculate the new parity, which is applied in a different manner with the subtractive or additive methods. When the calculations are performed the written data should be verified again, to ensure all data is still correct.

2.3.2 Error correcting code

Error correcting code (ECC) is additional data stored to recover data when corruptions occur. An overview of a couple of ECC techniques is featured below [24]. The error correcting code can be classified as a repetition of information, checksum or encryption.

- **Repetition** is a method where each bit is repeated a number of times. When the number of repetitions increase, the probability that the data can be decoded correctly increases. However, since more data is written there is a higher probability of errors. This scheme is also considered highly inefficient.

- **Parity check** bits can be used to determine whether a set of bits is even or odd. Most of the time, the parity check bits attempt to make the number of ones even. When the parity bits are used to decode the data, there is no information about which bits are flipped. Multiple layers of parity checks are often used to gain those abilities.

- **Hamming codes** are using parity check bits in such a way a single mistake can be corrected. Hamming codes have the ability to correct one error per block. The Hamming code also introduces a definition of distance, between the bits flipped to go from one byte sequence to another. This is called the Hamming distance. A more extended version of the Hamming code can restore up to two bits. Improved versions are still used today.

- **Convolution codes** can be used for cryptographic purposes. The bit sequence is encoded based on a state determined by summing a fixed set of previous bits. Each input bit is manipulated to produce the output bits, where the output is generated from combined information from different input bits (convolution). A key to decode the data is created using the starting state.

Accuracy varies per convolution technique. Applying the technique in combination with a checksum before encoding the data, makes this technique more reliable.

- **Bose-Chaudhure-Hocquenghem code** is a cyclic scheme to encode data. The method can guarantee that n number of errors can be corrected based on the Bose-Chaudhuri bound. A step by step procedure has to be used to determine whether the error has increased or decreased when correcting and decoding the data [20].

  Moreover, the BCH encoding algorithm is often used in SSD ECC mechanisms [11].

2.3.3 Data integrity security measurements

Data dependencies can be introduces in DBMS environments to provide security measurement [23]. When a malicious transaction occurs damage assessment has to be performed to repair all transactions. The transactions have to be repaired in backward order. A dependency graph can be constructed to repair the malicious transactions, but can create a lot of false positives.

Another security method involves using an authentication server along with a storage server to keep data integrity [12]. The authentication is used to proof corruption or authenticity. The only trusted components are stored at the client and is safe as long the client is not compromised.

2.4 Fault injection techniques

The fault can be injected through hardware and software, each technique brings its advantage. Multiple fault injection frameworks and techniques already exist, where popular ones are featured below. First the fault injection techniques are featured, which often are applied by the frameworks.

2.4.1 Techniques

The main techniques can be divided into two categories of software and hardware fault injection techniques [13]. Hardware fault injections mainly consist of two categories using injections with - and without contact. Fault injections with contact are changing the voltage or current of a chip and are often called pin-level injections. Active probes can be used to simulate stuck at faults. Socket insertions make use of sockets to insert a number of more complex logic faults between target hardware and the hardware circuit. The non-contact faults typically consist of using heavy ion radiation. Most of the hardware fault injections can be destructive as properties of the hardware itself are manually changed with, example given, injected currents.

The software fault techniques can be divided in compile time and run time fault injections. The compile time fault injections are implemented by modifying the source of the system under test. The at run time injections can be configured using a timer, system exceptions/traps and code insertion. Disadvantages of the software technique are inaccessible targets that can not be modified. Moreover, the software can behave very different as it is modified, thus has to be used with caution.

2.4.2 Frameworks

FERRARI, Fault ERRor Automatic Real-time Injector, is a fault injection framework that emulates the hardware faults by using software fault injections [14]. The framework creates two concurrent processes by making use of process forks. One of the forks is used to run the executing program, the other fork is used to run the fault injections. By using forked processes, the program under test can be paused, a fault can be injected and then being resumed. FERRARI uses traps and system calls to emulate non-permanent and permanent software faults.

FIAT, Fault Injection based Automatic Test environment, is mainly designed for system validation [25]. FIAT is mainly written to analyse the error detection and recovery mechanisms of a system and provide a guide to its design. Its aims at fault test at development and run time. The framework is implemented using two special hardware components. The first consists of the Fault Injection Receptacles, which injects the faults. Secondly, the Fault Injection Manager collects and analyses data from the fault injection. The analyses provides a number representing the number of faults detected. Programs compiled using the framework are modified at link time so faults can be injected from external program requests.

NFTAPE, aims to solve the problem of capturing the correct behaviour of a corrupt system, by creating a portable and light weight fault injection framework [27]. It uses the concept of triggers to indicate when a fault has to be triggered in a system when, example given, a specific application state occurs. NFTAPE relies partly on the user for measuring the faults and identifying the application errors.

LFI, a Library-Level Fault Injector, provides methods to inject faults when specific instructions or library functions in the program are performed [19]. It is mainly designed to test added libraries to a program, since the use of libraries grows significantly. The library can create interceptor stubs, which consist of the library under test wrapped around by some code. The newly LFI library is then used to test different workloads. Test scenario in the LFI framework is also using triggers.

Gigan, uses a Virtual Machine (VM) to create an isolated test environment [18]. It is mainly designed to test and analyse a system under a faulty system and can perform a restart when the system crashes. The fault injector uses a trigger based system and is implemented in the VM kernel along with a light weight logging mechanism. If the VM is crashing it can be paused, in contrast of using an operating system as test environment. Moreover, valuable data can be recovered from a crashed VM state before it is restarted. Another advantage of using a VM is the ability to create snapshots of the system state, creating the possibility to restore a system without restarting all processes of the VM. This is useful for fast test restorations [29].

2.5 Fault injection on databases

The influence of hardware failures on DBMSs have been tested for the MySQL DBMS. It is important for a DBMS to ensure the reliability and availability of data pointers and format information [28]. MySQL uses a B-tree to organize its data, thus using pointers to target its information. Type-aware pointer corruption was applied to test the DBMS, to change the pointer to another DBMS pointer type. An online and offline checker were used to detect if faults have manifested. Both sometimes failed. Sometimes, corrupted pointers were simply trusted. Using pointer manipulation, a loop can be created which can result in a DBMS crash. Different kind of selection queries were used to test if all correct data was returned, range queries could sometimes return wrong records.

Exposing ACID violations of a DBMS under test have to be performed without creating unreasonable situations [31]. A record-and-replay configuration can be used to analyse and inject faults in each situation. This creates a possibility to inject faults at any moment. Sometimes, tests can be targeted at the naive assumptions about the behaviour of the operating system. This provides methods to find potential targets to inject faults and test the DBMS better under those faults. When testing on the ACID principle, consistency checking is performed using range queries, by checking if the same results are always provided. Unreadable rows are reported as durability errors. Testing DBMS instances on the ACID principle using an undirected way is not enough to detect all possible failures of a DBMS.

CHAPTER 3

Apache Cassandra

The main reason to choose this DBMS is of the column structure and its state-of-the-art reputation. The chapter is discussed as follows. First, general information about Apache Cassandra is provided, this includes information about the data storage system and the system used by Cassandra. Moreover, a short overview is given of several robustness claims of Apache Cassandra itself. At last, the fault tolerance mechanisms are covered, to get an indication how the DBMS provides its measures of robustness.

3.1 General information

Cassandra is a distributed DBMS, which mainly is designed to work on cheap hardware, but still offering both high read and write efficiencies [17]. The system is designed to handle node failures gracefully and considers node failures as a normal occurrence. The DBMS is build to be deployed using multiple servers and data centres. The Cassandra system consists of a column structure and ensures every single row key operation is atomic, even when multiple columns are used. Durability is achieved by use of replication, where the row data can be replicated over multiple data centres.

Its linear scalability and demonstrated fault tolerance makes the DBMS an ideal choice to store critical data [8]. Its built-in caching system provides high performance in both read and write operations.

When data has to be read, the system routes the requests to the closest known replication or to all replicas and waits till sufficient responses are received [17]. Its linear scalability is achieved by partitioning of the data over a set of nodes in a cluster using a modified consistent hashing function. The modified hashing function preserves order of the hash output values. The hashing is used to determine the position of key and its corresponding data value. The position yields a location of Cassandra nodes in a ”ring” formation. The obtained key is also used to coordinate the traffic of the nodes for this key. The mechanism is also used to perform load balancing among the Cassandra nodes.

3.1.1 Column data store systems

Column oriented database are mostly storing their data on separate locations on disk [5]. The values of the columns are typically using small structures and applying compression whenever possible. Most of the time the compression applies very well as the data in the column has a low entropy. When implementing or using a column store DBMS, write operations and construction of row tuple data are considered to be problematic. Writes have to be performed on separate locations and tuple creation is problematic to read the data from those multiple locations. Most of the time, the user is also interested in only row values, not column values. However, write problems can be reduced using means of replication and memory buffering and the tuple creation can use extra data to make indexing faster. Most of the time, hybrid column store data systems are more appealing to implement than using the pure column oriented structure.

Apache Cassandra provides a data model where tables are distributed over a map indexed by a key [17]. The row key in a table is not restricted in size and every operation applied on the row key is guaranteed to be atomic. The atomicity is guaranteed to hold per replica and over multiple columns. Cassandra provides two column types, super and simple column families. A column family can contain a column or a super column. The super column family itself can only contain columns. This can be used to create higher abstraction levels when storing the data.

Files are stored along with an additional file containing a list of available keys per file. The structure is kept in memory, which optimizes the lookup of keys. The data lookup is performed first in memory before a disk lookup is performed. Moreover, the write operations of Cassandra are morphed to sequential writes to maximize the write throughput.

Cassandra makes use of the advantages and solves the problems when designing the column store database by means of replication and partitioning. Using the column families, more structure can be created. Furthermore, the column store database uses memory structures to speed up the read and write processes.

3.1.2 Robustness claims

When looking at the official home page of Apache Cassandra its robustness is achieved by its decentralized structure and configurable replication factor [1]. Each node in the cluster contains identical information to gain its decentralized structure. Durability is gained by appending writes to a commit log first, which happens periodically to minimize the usage of the `fsync` command, thus disk I/O.

When analysing these claims, the implemented fault tolerance seems to be mainly build against node failures. This provides the guarantee that data stays available through the network.

3.2 Fault tolerance mechanisms

Multiple structural design choices and techniques are applied to create the robustness and fault tolerant architecture of Cassandra [17]. Cassandra uses replication to achieve its decentralized structure, thus offers high availability. Each node is a coordinator of data values in its range of keys and items. Additionally, this key range is replicated along $N - 1$ other nodes. When a data centre is involved, a leader of the node cluster is created called the Zookeeper. New nodes are contacting the Zookeeper, which divides the key range along the nodes and tries to divide them equally. These data ranges are cached locally at each node and stored fault tolerant at the Zookeeper. This provides an advantage when a node failure occurs and the node comes back to the cluster. The responsible key ranges are still known by the node. Configuring the key system this way, all nodes are aware of the key ranges they are responsible to, which provides durability and fault tolerance when a node fails. The network partitions relax the number of responses needed when nodes are failing.

Communication with failed nodes is also avoided. A modified version of the Accrual failure detector is used to detect the node failures. The detector provides a suspicion level to indicate the network load condition and can also be used to configure the server load. A gossip based setting is used to spread the node failure knowledge through the network. Additionally, the gossip mechanisms provide the communication and distributes system states through the network.

A node failure does not necessarily signify a structural change to the network and should not result in rebalancing the partitions of the cluster. When a node has to be removed, this has to be performed by an administrator. Another administrative task is repairing nodes when disks are failing. The repair task can be performed by the system itself.

Local data persistence is achieved by using a write to the commit log, before an in-memory data structure is modified. The data structure is written to disk when a certain data size exceeds. The writes are sequential and generate efficient lookup indices. The files are automatically merged by a compaction process. A read operation first queries the in-memory data, following a lookup on disk. A bloom filter is used to summarize the keys of the file and is stored in memory for each data file, to speed up the lookup process.

Failure detection and cluster membership is built on top of the network layer and uses non-blocking I/O to handle UDP control and TCP for application related messages. A state machine is used to process reads and writes. Globally, the steps are as following. First, nodes are identified to which data they contain for a key. Secondly, a routing request is performed to the responsible nodes of this data and wait a certain time limit. Then, the latest time stamp is obtained of the data and if one of the responses are not up to date, a repair has to be scheduled.

Cassandra uses a rolling commit log containing a bit vector header of a significant size, such that enough columns can be represented. When the in-memory data structure of a data file is committed to disk, a bit is set in commit log that the data is written to disk. When a commit log is rolled, the bit vectors of the prior commit logs are checked, before deleting from memory. A configuration change can be made such that only a in-memory buffer commit log is used, this against the risk of data loss when a crash occurs.

As all writes to disk are performed sequential, almost no locks are needed to perform the read and write operations. This has to advantage that no concurrency issues can occur which can exist in a B-Tree structure.

Cassandra provides several fault tolerant mechanisms following its design. Replication mechanisms and the means of coordinator nodes provide the decentralized structure. The Zookeeper and local key ranges provide fault tolerance against node failures as failed nodes can easily come back up. Moreover, when the failures occur the number of required responses are relaxed. The failures are spread through the network using the gossip system providing node conditions and data. Performing writes to the commit logs before writing to disk create its local persistence.

CHAPTER 4

Framework design

The framework aims at creating a design which is able to create a testing environment on general DBMSs. Referencing from the previous frameworks, the following features should be included to make the framework more reusable and applicable to other test situations.

- Setting up a test environment which can inject faults in a single DBMS.
- Make the testing environment modular, such that other DBMSs can be used and tested on other systems.
- Several number of specific fault injection scenarios should be possible, such as different fault injection types.
- Create a fixed reusable format which make it easier to analyse and contribute to the testing results. Several researches focusses on a singular type of DBMS, it might be interesting to compare the results of similar DBMSs.

The structure of this chapter is as follows. The first section is about an overview of the proposed framework. The modules and components are explained without going in the implementation details. The next section is about modular DBMS integration and how this could be achieved. This includes the configuration of a DBMS cluster. The third section is devoted to the definition of the fault injection test. At last, the definition of the framework tests is explained, including how test results are formatted.

4.1 Overview

The framework is implemented using a client-server structure, as it makes central communication between the different DBMS servers and the client possible. The client side has to provide functionalities to make communication with a server DBMS cluster possible, parsing a given test scenario and determine when faults have to be injected into the DBMS under test. The test results of the server will be communicated back to the client side and stored in a local database in a generic format. Meanwhile, the server side can perform validation and query work. The structure of the framework is shown in figure 4.1.

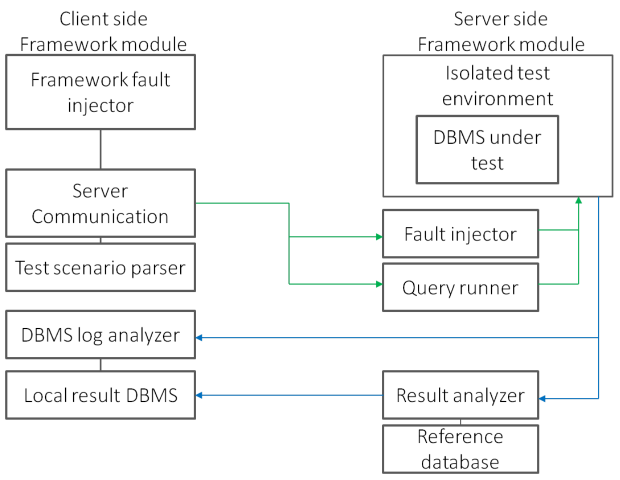

Figure 4.1: The structural plan of the framework, including all server and client side sub-modules. The relationship lines indicate the communication paths of the modules.

The DBMS under test is running in an isolated environment at the server side. This ensures multiple test scenario can be performed, the environment can be restored and another test can be started again. Moreover, this will ensure the previous tests will not influence the current running test. The isolated environment can consist of a virtual machine or container virtualization. Furthermore, queries defined in the client tests are performed as the the fault injection will be performed.

Also, to properly test the data integrity a validation is performed at the server side. All queries and its results are saved at the server side in a separate verification database. This database contains, example given, for files the hashes of the file, the query result number identifier and the query identifier. The result analyser will finally give an indication which errors occurred while querying the database.

4.2 Modular DBMS integration

The modular DBMS integration can be accomplished by writing a wrapper which executes several operations. The database should be able to be set up in a manner where different parameters could be used to set, example given, the replication factor of the data. Different ways to read and insert the test data, such as a CSV format or binary data. In addition, the database should be able to be queried, while catching unexpected exceptions due the fault injections. Eventually setting up a distributed DBMS on different servers and query the servers locally. Finally, a database log analyser has to be implemented to detect fatal or other database errors.

The JSON structure in figure 4.2 could be used to configure a database cluster providing initialization parameters for the DBMS cluster itself and the server configuration.

```
SSH server connection configurations
{ ...
    "n_nodes" : 3,
    "user" : [
      "user1", "user2", .. ],
    "password" : [
      "password1", "", ..],
    "host" : [
      "127.0.0.1", "", ..],
    "port" : 3022,
    ...
}
```

```
Database configurations
{ ...
  "db_type" : "cassandra",
  "db_version" : "3.5",
  "db_meta" : {
    "init" : {
      "class" : "SimpleStrategy",
      "replication_factor" : 3
    },
    "reuse_keyspace" : true,
    "keyspace" : "keyspace_name"
  },
  ...
}
```

Figure 4.2: Example configuration code how to connect with a server (*left*) and set up a test database (*right*).

Furthermore, there are different approaches to obtain information and implement the fault injections in the DBMS. The files used by the DBMS can be tracked and can give an indication about the structure of the DBMS. Using this information, a fault scenario can be further specified to ultimately modify important DBMS structure components, such as the B$^+$-tree pointers of a MySQL DBMS.

4.3 Fault injections

When designing the fault injections, the test environment and the faults itself should be defined. The fault injection tests should be performed in an isolated environment where the DBMS is running and can be queried. The isolated test environment can be obtained via means of using virtualization. Virtualization gives capabilities to start and restore the testing environments multiple times, without being influenced by previous fault injection tests.

The fault injections can be implemented using software simulated hardware faults. The faults can consist of, example given, multiple bit flips on several target files. The target files can be obtained by tracing the used database files while inserting and querying data. A list of those targets can be stored on the server side of the DBMS as its contents are independent of the other servers.

An alternative way of implementing the fault injections would be modifying the DBMS source code to allow injections in more specific targets, such as pointer manipulations. As this implementation would be very DBMS structure specific, it would not be feasible to create such injections in a general manner. The database file tracing method is the main focus to retrieve files to implement the fault injections.

4.3.1 Test scenario definition

The framework should be able to run test scenarios flexible enough to be used for different kinds of data and verification methods. The previously proposed JSON can be expanded to satisfy these needs. Moreover, a test scenario should contain which files can be targeted and which fault injection should be executed. This includes the number of faults to be injected per target and an interval between the injection of the faults. The corresponding JSON in figure 4.3 illustrates this.

```
Definition of a test scenario
{ ...
   "test_scenarios" : {
    "repetitions": number of scenario repetitions,
    "data_type": "files",
    "scenarios": [ {
        "FI_type": "bitflip",
        "num_files": number of target files,
        "excluded_extensions":
        [ ".log", ".so" ],
        "ignore_system_files": True,
        "flips_per_file": number of bitflip insertions per file,
        # Fault injection delays, a start and interval delay in milliseconds.
        "start_delay_ms": 1000,
        "delay_ms": 500
     }, {second scenario}, ...  ]
  },
  "data_to_insert" :  {
    "files" : "datasets/complete_ms_data/"
  },
  ...
}
```

Figure 4.3: A display of the different parameters of a fault scenario, including parameters to determine a test dataset.

4.4 Execution of tests

The framework should perform the following steps to run a test scenario properly. A JSON file containing the test scenarios, server - and DBMS configurations will be parsed and processed by the framework client. When the server connections are made, several installations and configurations have to be performed. First, on each test environment the framework dependencies should be installed. Afterwards, the distributed DBMS have to be made ready for data insertion and querying. As distributed DBMS are used, a server location has to be chosen to transfer and load the dataset.

After the database is configured, a verification database is initialized with the results of the queries used in a test scenario. As no fault injections took place, all results and verification calculations will be performed over valid data. When the data insertions take place, a database file tracer should be started, which provides the target files to be used when running the fault injection experiments.

As the framework configuration is finished, the test scenarios can be performed. However, some preparations should be made first before the test scenarios are executed. First, a proper backup of the DBMS and its inserted data set should be created. This is used to restore everything after a single test scenario is finished. Secondly, at the client side a list should be retrieved with possible fault injection targets. This minimizes the server communication between each test scenario run. The target file lists, cooperates all files used by the DBMS under normal working conditions.

The test scenarios should consist of the following steps:

1. Make sure all DBMSs are up and running on all servers. This, in order to be certain the DBMSs do not falsely return `query time out` or `host unavailable` errors.

2. Start a fault injection thread on the client side, which should perform a fault into the server side code. In addition, a process should be started which queries the DBMS with the same queries as the DBMS is initialized.

   - When querying the DBMS, each query result should be verified on, example given, duplicates and checksums. Each error or exception should be obtained.

3. The next step consists of performing analyses of the DBMS logs on warnings and errors if they occurred while querying. A small interval in seconds between the query fault and the log occurrences should take place, so no irrelevant logs are extracted.

5. Insert the results into a local database, as formatted in figure 4.4, and restore all the database results.

4. After restoring and starting each DBMS instance, step one should be performed till all repetitions of the test scenario are performed.

The result format should be adaptable to different database results, but could be formatted as shown in figure 4.4. The format is flexible enough to obtain different results from different kind of tests scenarios and other DBMS. Some of the stored data, like the test scenario and DBMS data, could be de-duplicated by storing a reference to another database collection. A NoSQL database is the most suitable format as the test results could be different in each run.

```
General result format definition.
{ "server_params": { # Relevant server information to the test scenario.
    "n_nodes": n_nodes
  },
  "db_type": db_type,
  "db_version": db_version,
  "db_meta": db_meta (database initialization parameters),
  "tested_dataset": which test dataset,
  "test_scenario": test scenario data,
  "injection_times": time of when the fault injection were performed,
  "target_files": the targeted database files,
  "db_error_logs": The database error or warning logs,
  "effects": { # Per query performed
    query num: {error1:0, error2:1, timestamp: query time},
     ...
  },
  "run_id": A fault injection run id,
  "res_id": Result id (Nth repetition),
  "time": #time the result is stored in the local client DBMS.
}
```

Figure 4.4: A possible result format containing the most important fields to be obtained from each fault injection run. With this information additional statistics can be calculated such as the most common database errors and how well the database behaves when performing the fault injection scenarios. The field descriptions contains additional information about what kind of information is stored.

CHAPTER 5

Framework implementation

This chapter covers the implementation details and design decisions while implementing the framework. The implementation is created and based to work in a Linux environment and is written using Python 2.7.

The chapter covers the following subjects. First the framework configuration is discussed along with the configuration of the isolated test environments. Next, the implementation and functioning of the fault injections are covered. The section includes how the DBMS targets are determined and how the implemented fault injections exactly functions. Afterwards, the test scenarios and the data validation are discussed. The chapter covers the population of the Cassandra DBMS cluster, its validation and the extraction of relevant log results from a test scenario run.

5.1 Framework configuration

First, the framework parses a given JSON file to establish connections with multiple servers. The established connections are implemented using the `paramiko` Python module. The paramiko module utilizes an SSH connection to establish the server connections, which enables the execution of command line commands on the server. Moreover, the module supports file transfers to the server. Before transferring the files, `bzip2` compression is applied to minimize the network load. As all connections are made, the server dependencies can be installed using the SSH connections. Afterwards the necessary framework files are transferred.

5.1.1 Testing environment

Container virtualization is used to create the isolated server test environments. `Docker` containers are used to configure the distributed DBMS test environment [3]. A Docker image consists of container layers which itself consists of, example given, a version of Linux, the DBMS and other dependencies. When running multiple docker images on the same machine, Docker is guaranteeing the isolation of all processes from each other. Another advantage is the ability to create snapshots, *commits*, of Docker containers and provides fast restorations of the snapshots.

The Docker Apache Cassandra image is used to configure the DBMS cluster, can be found on [4]. Some of the Docker DBMS images do not store the DBMS database data itself. This is solved by mounting a volume to the Docker container, consisting of a local directory or a Docker volume format, providing the DBMS data storage. The directory that the Apache Cassandra DBMS utilizes would be `/var/lib/cassandra`. This directory is mounted to a non-virtual directory on the DBMS server. The used Docker command to set up the image is shown in figure 5.1. The additional flags used are explained as well.

```
docker run -d --name cassandra-node -e CASSANDRA_BROADCAST_ADDRESS=$OwnPrivateIP \
          -e CASSANDRA_SEEDS=$ConnectedPrivateIPNode -P -p 7000:7000 --net=host \
          -v $UserDir/fi-framework/db_data:/var/lib/cassandra cassandra:3.5
```

Figure 5.1: The Docker command to run the Apache Cassandra DBMS on a cluster using two nodes. The Docker command creates a new Docker container running the Cassandra Instance. The `-d` flag runs this process in the background, the `-e` flags are setting environment variables in the Cassandra configuration files. The broadcast address has to be set to the private IP address of the current server, the Cassandra seeds variable represents the private IP address of a target connection node, which can be omitted. The `-P` and `-p` flags are registering the communication ports of the Docker image. The `--net=host` flag is used to bind the container to the Ethernet ports of the current host, which allows communication through from other node locations. The `-v` flags is used to mount a directory to the container. Finally, the last argument is the image id or image which the Docker container instance has to start.

When creating a backup of Apache Cassandra DBMS Docker instance, all the file data is flushed to disk by executing Cassandra's `nodetool flush` command. This ensures all data is properly stored on disk instead, instead of only in memory. Afterwards the Docker Cassandra instance is stopped and its non-virtual data directory is archived by using the Linux tar command. As the backup has to be restored, a Python script is used to compare the current data file MD5 hashes against the MD5 checksum calculations of the backup directory. A list of MD5 hashes is created only once of the backup directory and loaded from disk when the restoration has to be made. This optimizes the backup restoration as not all files have to be extracted, which can be slow while extracting multiple small files.

5.2 Fault injection

The implementation of the fault injection can be divided in two stages. The first stage is retrieving potential target files on which fault injections can be performed, the second stage is performing the fault injection itself on one of the retrieved files.

5.2.1 Retrieving target files

The target files of the DBMS are retrieved using a combination of Linux system commands. The `strace` process is used to track all read, write and open commands from the DBMS. Moreover, when those commands are executed the corresponding file descriptor is retrieved. The strace process indicates which file descriptors are used when, for example, querying the DBMS. Using the `lsof` command on the DBMS process retrieves a list of the file descriptors and all opened files by the DBMS process. Combining this information creates a list of potential target files. Figure 5.2 shows the command line commands used to retrieve this information.

```
strace -p $DBMS_PID -f -e trace=read,write,open
lsof -p $DBMS_PID
```

Figure 5.2: The specific command line commands used to get a target file list of a running DBMS process. The `strace` and `lsof`'s `-p` flag indicate which process to track, and the `-f` flag of the strace command indicates to track forked processes of the DBMS. When tracking only the Apache Cassandra main DBMS process, no read, write or open system calls are observed as forked processes are used to perform those operations.

A Python script is used to start the processes using the `subprocess` module and analysing its output. The DBMS process identifier (PID) is retrieved using the Linux `grep` command performed on the process list command `ps -aux`.

The target file list retrieved is stored on the server side, as the DBMS can create and use different files on each server. One downside of the approach of using the system commands, is the necessity of `root` privileges to run. However, this is already needed as all Docker commands needs these privileges.

Before a test scenario is started, a target list is transferred from the server side to the client side. At the server side all listed files are checked if they actually exists in the Docker image. Moreover, the test target file list is filtered on the file extensions to be excluded from the test scenario. The client receives this filtered target list once per test scenario to be used by the fault injection thread, which is discussed further in section 5.2.2.

5.2.2 Fault injection implementation

The chosen fault injection to implement is the bit flip. A Python script is used as Python is supported by the basic Docker Cassandra image. First, the implemented bit flip in Python retrieves a byte character from a random location of a file. On this byte character a single bit is changed by applying an XOR operation with the integers of the form 2^{n-1}, with $n \in [1, 8]$. This will result in a single bit flip in the byte character. Then, the resulting modified byte character is written to its original file location. Using this script, arbitrary files can be targeted, such as the process memory files in Linux. Multiple bit flips can be applied to a single file if desired.

Using the modification of files a possible HDD fault can be simulated in a DBMS. The fault injection Python script is stored in the Docker container image itself and can be executed using the Docker `exec` command. As the fault injection script only targets DBMS files in the Docker container, no additional measures have to be taken to protect the operating system of the server. Meanwhile the DBMS is queried in a test scenario, the fault injections takes place. The fault injection is performed in a separate thread and chooses a configurable number of times a target file where a fault injection takes place. A list can be provided containing a specific number of target files if desired, else targets are chosen at random from the target list. Two different types of delay can be introduced within the fault injection thread. The first type, delays the start of the fault injection in its entirely, the second is the interval delay between the fault injections itself.

Per fault injection a request is send form the client to the server's Docker image to perform the requested injection. This introduces some overhead, but keeps the simplicity of the fault injection script. Exceptions happening in the fault injection script are handled at the server side to guarantee all faults will be injected.

5.3 Test scenarios

When implementing the test scenario parsing and configuration, a test data set has to be inserted into the DBMS. The data set is first transferred using the SSH connections. In the meantime, a process is started to trace the behaviour of the DBMS and retrieve its potential targets. After the transfer completes, the test data set can be inserted into the DBMS. The implemented Apache Cassandra DBMS file type insertions is limited to files, which are inserted using the scheme shown in figure 5.3.

```
CREATE TABLE table_name (
  id INT PRIMARY KEY,
  file_name TEXT,
  file_data BLOB
);
```

Figure 5.3: Creation and format of a Cassandra table used to initialize the database with a file data set. The primary key is a numeric file identifier. Next the file name and the binary file data itself is stored.

When designing the database scheme, it is assumed the DBMS will keep an internal checksum along different data formats. If data corruption should occur, action should be taken to add, for example, the checksum of the blob data along side the blob data itself. This would only deem to be necessary if the corrupted data is also returned to the client as valid data. In Apache Cassandra would a possible solution consists of adding a column with the *decimal* data type, which can be used to store the MD5 hash of the blob data in base 10.

After the data is inserted, the data is queried with the test queries and used to populate a local SQLite database with the query number, the file name and the MD5 hashes of the file data. This is used to perform the data integrity analysis, which is described in 5.3.1.

When running a test scenario at the client, first the backup has to be performed of the current Docker DBMS image along with its mounted Docker volume. This step can be omitted when another test scenario has to run with the same test environment configuration. Next, a UUID is generated to define the test scenario result identifier. Thereafter, the target files are retrieved specified by the scenario. The scenario can then be started. First all Docker DBMS containers are verified to be running. A script performs creations of database session until all the connections succeed.

Afterwards, the fault injection along with the database querying and validation steps on the server are started. The results are then processed by the client, extracting the relevant DBMS log results on a new thread, further described in 5.3.2. Next, the data integrity results and the relevant logs are then stored in a local result database. An arbitrary NoSQL database can be used to store the scheme as shown previously in figure 4.4. The chosen NoSQL DBMS was MongoDB, mainly chosen for the simplicity of the PyMongo Python library.

When the logs are analysed, the backup configuration is restored. The Docker containers are stopped, removed and newly created with the backup image. Then, the next test repetition or test scenario can be performed.

5.3.1 Data validation

When the DBMS is queried while running a test scenario, its data is returned in the format as shown in figure 5.3. However, the `file_data` column is immediately hashed by an MD5 hasher. The first validation step consist of checking the created file hash with the reference SQLite database. The SQLite database is chosen as the contents are only needed locally and the Python `sqlite3` module is already mostly on systems including Python.

Additional to comparing the SQLite database for checksum mismatches, the ordering of the query results can be verified. It can be checked if all results are available and no results are missing. Afterwards, the results are checked on duplicates, which can occur when data pointers are not pointing at the right data. An example would be two keys pointing to the same value.

The DBMS itself also has its validation mechanisms and can throw exceptions when these occur. All assembled results are returned to the client DBMS into the format shown in figure 5.4. This format is flexible, as new error statistics can easily be added and multiple errors can be registered at once.

```
{"query_id":
  { "timestamp": time when query was requested,
    "verification_errors": SQLite database checksum mismatches includes a check on the result ordering,
    "results_missing": the number of results missing,
    "results_duplicated": number of duplicates,
    "time_out": if the DBMS response timed out,
    "read_failure": a replica has send a failure message when reading the data,
    "write_failure": a replica has send a failure message when writing data,
    "invalid_request": Query is malformed; this can happen when a
                       table does not exists or can not be reached,
    "no_host_available": DBMS is not available for querying.
  },
"query_id": {...}, ...
}
```

Figure 5.4: The information about specific Apache Cassandra errors are retrieved from the Python driver documentation [2].

5.3.2 Logged results extraction

Having the data validation errors obtained, there are faults that are not giving a direct error, but are only logged into the DBMS logs. Using the timestamps as defined in figure 5.4, timestamps from the database logs can be extracted, which cover the errors or warnings - if any. A small interval of +/-3 seconds is used between a query result and a logged result to obtain the relevant logs.

The Cassandra DBMS logs are obtained using the Docker `logs` command, returning all database logs. The Cassandra log format and the Python regular expression used to extract information from the log are shown in figure 5.5. The time and the type of the log are extracted to get an indication of the importance and the Unix time stamp time to calculate its relevance. In general, the info, debug and warning messages are ignored. The warning message is only retrieved when it is considered relevant. This is necessary, or else irrelevant warnings of the test scenario are extracted from, example given, when the database was performing its start up.

```
=== Cassandra log format ===
INFO|DEBUG|WARN|ERROR|FATAL    hours:minutes:seconds     message
=== Python Regular expression to extract relevant data ===
(WARN|ERROR|FATAL)\s*([0-9]{2}:[0-9]{2}:[0-9]{2})
```

Figure 5.5: The format of a Cassandra DBMS log. The python regular expression will only match when the log begins with WARN, ERROR or FATAL. Additionally, the time stamp is extracted. The parentheses group the results to ease the use of the expression results in Python.

CHAPTER 6

Experiments

The experiments are using the Multi-spectral Image Dataset provided by the Columbia University [9]. The image dataset is chosen as it provides large data structures to be inserted into the DBMS which may be less fault tolerant than other inserted data. The provided data scheme for Cassandra, as shown in the implementation chapter, is used to query and load this data. Each directory in the data set containing files is used as a table containing the directory contents.

A Google cloud environment is used to employ a Cassandra DBMS cluster. A three node cluster was configured with an Ubuntu server operating system of version 16.04. The Apache Cassandra Docker containers were using Cassandra instances of version 3.5. The used Docker version was 1.11.1 - API version 1.23.

6.1 Testing local data integrity

The experiment focusses on the influence of bit flips on random location in non-index files only and index and non-index files. When only non-index files are targeted, the `"Index.db"` files are ignored. The bit flips are not targeted at `.jar`, `.log` and other non-DBMS file system files. Those files are considered irrelevant into real deployment of the DBMS. The DBMS files will be of such a significant size such that the chance is higher that faults occur in files covering a large part of available space.

The experiment can be divided into three phases. The first phase covers the influence of the bit flips on the non-index files and the non-index and index files. This will give an indication if different types of files are less fault tolerant then other files. The second phase covers the influence as the number of bit flips performed at the target files increases. More introduced bit flips will influence the DBMS more and can have a higher chance of detection. The third phase is looking at the influence of replication on the fault injection experiments. This to research if replication increases the fault tolerance of the DBMS.

The influences of the fault injections are measured at the server side. The metrics used are how many faults are detected by a client querying the server and detected by the server itself. When the server side detects faults, it should return an error to the client side. Prevention of returning corrupted data is preferred then allowing it. The client side does not always detect cases of corruption, thus undetected cases can lead to, example given, wrong data in data analyses experiments.

Reasoning from this perspective, the following measurements have to be made to get a picture of the significance of the results. At first, the number of experiments where the server detects faults have to be measured. Secondly, the number of faults detected by the client have to be measured. Thirdly, the number of faults detected by the server and not the client have to be measured.

At last, the number of faults detected by the client and not at the server have to be measured. The metrics provide an overview of how well the DBMS protects the client against incorrect data.

The queries performed range from 'select all' queries to some more advanced queries. The queries are written in the Cassandra Query Language (CQL). The more advanced queries consist of range queries and including sometimes a result limit. The appendix provides the used queries. A decent amount of queries are written such that all tables are first queried with 'select all' queries on alphabetic order (not necessary the order of insertions), followed by range queries queried on the tables on a random order.

Moreover, the experiment focusses on the short term effect of such faults. When faults are not immediately registered by the DBMS after the querying and the verifications are performed, it is considered to be unregistered.

6.1.1 Expectations

When targeting the non-index files only, it would be expected that almost all errors are detected by the DBMS. Undetected cases would happen when files are targeted which are not used in the experiment run. The detected faults would consist of errors returned by the DBMS along with a log entry indicating the fault occurrence. The expected faults would mostly be read faults and if the injection is undetected in the data, some checksum faults.

When targeting both non-index and index files, it would be expected that more faults are being detected. Index files are important and probably are more protected than the data files. When faults occur it would yield possible duplication errors, missing data and checksum mismatches.

When more faults are injected, it is expected that more faults are detected and possible data corruption will be observed more often. The chance that the faults are undetected will slim as possible important data files are being modified. Replication will lessen the influence of the faults as when errors occur, the replication instances can still respond with valid data.

In general the query results would be often the same as expected. The more advanced queries would yield faults more often.

6.2 Results

The experiment is taken over 1600 runs, where 800 are performed with replication and 800 without. The replication results are using a Docker cluster and the non-replication experiments are only using a single Docker instance on a node. The Docker cluster with replication was configured using a single main node and two nodes, which where communicating with this main node, but not with another nodes.

The log results obtained from the experiment are mostly the same and are providing information about corrupted SSTables, occurring when a row could not be build or when checksum corruption was detected while decompressing serialized data.

6.2.1 Results without replication

	Using a single target			**Using two targets**	
Using a single node - no replication	Non Index files	All files		Non index files	All files
Server detects fault	65	54		97	82
Client detects fault	0	1		0	7
Administrator detects, client does not	65	54		97	82
Client detects, administrator does not	0	1		0	7

Table 6.1: Results of the first four runs without replication. Each run, with and without non-index files, is repeated a hundred times. The first two runs results are obtained using a single bit flip on a single file, the last two results are obtained using two target files.

	Using a five target files			**Using ten targets**	
Using a single node - no replication	Non Index files	All files		Non index files	All files
Server detects fault	100	94		100	98
Client detects fault	0	3		0	0
Administrator detects, client does not	100	94		100	98
Client detects, administrator does not	0	3		0	0

Table 6.2: Results of the last four runs without replication. Each run, with and without non-index files, is repeated a hundred times. In the first two runs five files are targeted with a single bit flips. In the last two ten files are targeted.

6.2.2 Results without replication - percentages

	Using a single target file			**Using two targets**	
Using a single node - no replication	Non Index files	All files		Non index files	All files
% Faults observed by client and admin	65%	55%		97%	89%
% Faults observed by db admin only	65%	54%		97%	82%
% Faults observed by client only	0%	1%		0%	7%

Table 6.3: Results displayed with fault percentages derived from the result tables. The larger the value, the larger the chance the fault is detected.

	Using a five target files			**Using ten targets**	
Using a single node - no replication	Non Index files	All files		Non index files	All files
% Faults observed by client and admin	100%	97%		100%	98%
% Faults observed by db admin only	100%	94%		100%	98%
% Faults observed by client only	0%	3%		0%	0%

Table 6.4: Results displayed with fault percentages derived from the result tables. The larger the value, the larger the chance the fault is detected.

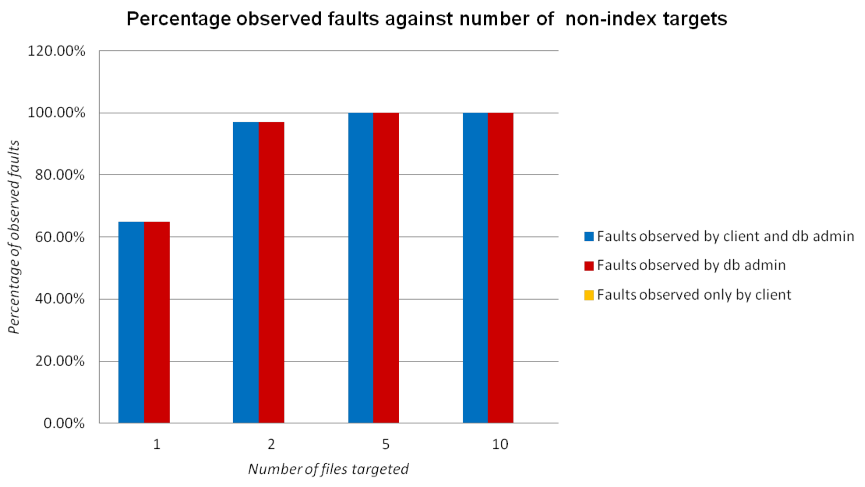

Figure 6.1: Graph representing the percentages of observed faults when only non-index files are targeted.

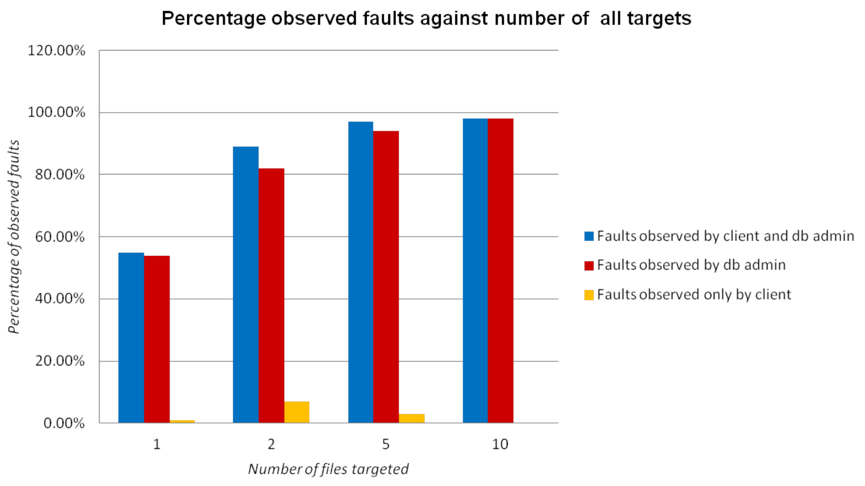

Figure 6.2: Graph representing the percentages of observed faults when non-index and index files are targeted.

6.2.3 Results with replication

	Using a single target			**Using two targets**	
Using three nodes - replication factor 3	Non-index files	All files		Non-index files	All files
Server detects fault	57	40		78	64
Client detects fault	0	15		0	5
Administrator detects, client does not	57	40		78	64
Client detects, administrator does not	0	15		0	5

Table 6.5: Results of the first four runs without replication. Each run, with and without non-index files, is repeated a hundred times. The first two run results are obtained using a single bit flip on a single file, the last two results are obtained using two target files.

	Using a five target files			**Using ten targets**	
Using three nodes - replication factor 3	Non-index files	All files		Non-index files	All files
Server detects fault	98	92		100	95
Client detects fault	0	1		0	0
Administrator detects, client does not	98	92		100	95
Client detects, administrator does not	0	1		0	0

Table 6.6: Results of the last four runs without replication. Each run, with and without non-index files, is repeated a hundred times. In the first two runs five files are targeted with a single bit flips. In the last two ten files are targeted.

6.2.4 Results with replication - percentages

	Using a single target file			**Using two targets**	
Using three nodes - replication factor 3	Non-index files	All files		Non-index files	All files
% Faults observed by client and admin	57%	55%		78%	69%
% Faults observed by db admin only	57%	40%		78%	95%
% Faults observed by client only	0%	15%		0%	0%

Table 6.7: Results displayed with fault percentages derived from the result tables. The larger the value, the larger the chance the fault is detected.

	Using a five target files			**Using ten targets**	
Using three nodes - replication factor 3	Non-index files	All files		Non-index files	All files
% Faults observed by client and admin	98%	93%		100%	95%
% Faults observed by db admin only	98%	92%		100%	64%
% Faults observed by client only	0%	1%		0%	5%

Table 6.8: Results displayed with fault percentages derived from the result tables. The larger the value, the larger the chance the fault is detected.

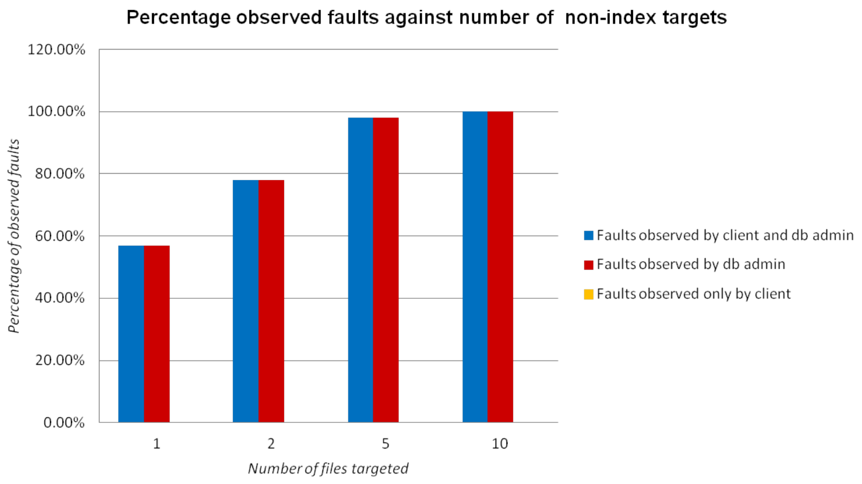

Figure 6.3: Graph representing the percentages of observed faults when only non-index files are targeted.

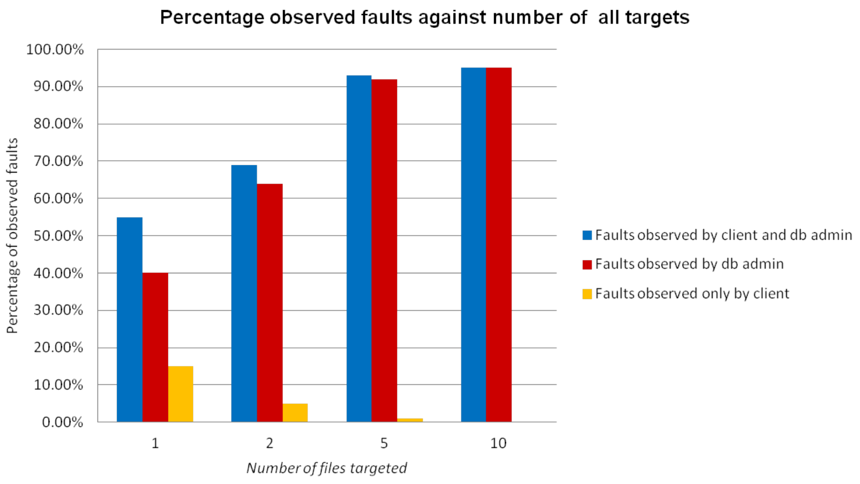

Figure 6.4: Graph representing the percentages of observed faults when non-index and index files are targeted.

6.3 Discussion

Before starting the discussion, the definition of undetected faults at the client and server have to be elaborated in this context. Undetected faults at the clients side consist of errors in the received results of the client performing queries at the DBMS, while no explicit errors are returned to the client by the DBMS itself. However, the faults could still be logged by the DBMS. Undetected server side faults consist of unexpected result data returned by the DBMS to the client, while the DBMS does not detect any faults.

From the results can be derived that most of the fault injections results are detected by a fault generated at the server side. When the data files of the key space are targeted, exceptions are logged and a read failure is returned.

Undetected faults at the server side occur when DBMS and system_schema data or index files are targeted. Since the queries performed at the server side are not using the system key spaces directly, the faults remain undetected. However, it is expected that in long term the faults would be detected, which has not been tested due time constraints.

The undetected faults at the client side are mainly occurring when index files are targeted. The faults consist of missing a single row in the result data, thus out of order results. As Cassandra uses the index file to look for data of keys, it could be that a key is modified in this file. One of the keys is different, such that it is not considered to be relevant for certain queries and is not returned. This can also be the reason why this fault goes undetected as the behaviour is not considered wrong at the server side.

When a DBMS with a larger replication factor is used instead of just a replication factor of one, the DBMS returns the right results more often. Even when 10 faults are injected some results are still having a chance to survive and to be properly returned to the client. Remarkable is the larger number of undetected faults at the server side. A possible explanation is not yet found. More experiments should be performed to analyse the behaviour better.

As expected, the number of detected faults increase, when the number of target injection files are increasing. This satisfies for the tests with replication and without.

At last, it can be remarked that the server faults are always being detected by a DBMS administrator. Exceptions could always be handled by the system itself without needing to handle other exceptions.

Overall, Cassandra can be considered fault tolerant. No data is returned and a error message is returned when faults are detected. In the DBMS log the corruptions are detected correctly. Only when index files are modified, data is sometimes omitted probably because of its modified key.

CHAPTER 7

Conclusions

A framework was implemented to be able to inject faults and analyse the behaviour of a target DBMS. An isolated environment was created via means of container virtualization using Docker. The tests are flexible by parsing JSON files where all configurations can be stored. Data corruption can be injected through the implemented software fault injection, which can indicate the robustness of the DBMS under test. The verifications of the DBMS at the server are providing sufficient information to analyse the behaviour of the DBMS under faults along with its log results. The assembled results stored in the NoSQL DBMS are formatted that multiple NoSQL DBMS collections are not needed to store all test results.

Apache Cassandra provides a high fault tolerance rate and can satisfy its claims, under the fault scenarios exercised in this thesis. Apache Cassandra provides mechanisms against data integrity and no checksum mismatches were observed. However, faults in index files seem to go by undetected sometimes, which results in results missing and thus out of order results. Applying replication lessens the number of errors, which can indicate that Cassandra makes use of the other nodes to provide the results.

All in all, the behaviour of Cassandra is as expected, providing log and server errors when faults are detected. Looking at the logs, the errors are correctly identified as corruption warnings. Cassandra provides means to keep data integrity and does not return invalid result data.

7.1 Future research

Further research could be performed about how the framework can simplify the implementation of other fault DBMS. Also, the framework can be extended to provide more scenario possibilities. Moreover, research can be performed at the long term effects as the system and system_schema key spaces are modified. Different data types can be inserted and verified on fault tolerance. To be more thorough with the fault tolerance research, the offline fault checker should be enabled to check if the system and system_schema faults are also detected.

Bibliography

[1] Apache Cassandra. `http://cassandra.apache.org/`. Accessed: 2016-05-30.

[2] Apache Cassandra Datastax Python-driver documentation. `http://datastax.github.io/python-driver/api/cassandra.html`. Accessed: 2016-05-28.

[3] Docker Documentation. `https://docs.docker.com/`. Accessed: 2016-05-20.

[4] Docker Hub - Offical Cassandra Docker image. `https://hub.docker.com/_/cassandra/`. Accessed: 2016-05-20.

[5] Abadi, D. J., Boncz, P. A., and Harizopoulos, S. Column-oriented Database Systems. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1664–1665.

[6] Bairavasundaram, L. N., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Goodson, G. R., and Schroeder, B. An Analysis of Data Corruption in the Storage Stack. *ACM Transactions on Storage (TOS) 4*, 3 (2008), 8.

[7] Bairavasundaram, L. N., Goodson, G. R., Pasupathy, S., and Schindler, J. An Analysis of Latent Sector Errors in Disk Drives. In *ACM SIGMETRICS Performance Evaluation Review* (2007), vol. 35, ACM, pp. 289–300.

[8] Chandra, D. G., Prakash, R., and Lamdharia, S. A Study on Cloud Database. In *Computational Intelligence and Communication Networks (CICN), 2012 Fourth International Conference on* (2012), IEEE, pp. 513–519.

[9] Computer Vision Labratory Columbia University. Multispectral Image Database.

[10] Cornwell, M. Anatomy of a Solid-State Drive. *Commun. ACM 55*, 12 (2012), 59–63.

[11] Dong, G., Xie, N., and Zhang, T. On the Use of Soft-Decision Error-Correction Codes in NAND Flash Memory. *Circuits and Systems I: Regular Papers, IEEE Transactions on 58*, 2 (2011), 429–439.

[12] Heitzmann, A., Palazzi, B., Papamanthou, C., and Tamassia, R. Efficient Integrity Checking of Untrusted Network Storage. In *Proceedings of the 4th ACM international workshop on Storage security and survivability* (2008), ACM, pp. 43–54.

[13] Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. Fault Injection Techniques and Tools. *Computer 30*, 4 (1997), 75–82.

[14] Kanawati, G. A., Kanawati, N. A., and Abraham, J. A. FERRARI: A Flexible Software-Based Fault and Error Injection System. *Computers, IEEE Transactions on 44*, 2 (1995), 248–260.

[15] Keeton, K., Santos, C. A., Beyer, D., Chase, J. S., and Wilkes, J. Designing for Disasters. In *FAST* (2004), vol. 4, pp. 59–62.

[16] Krioukov, A., Bairavasundaram, L. N., Goodson, G. R., Srinivasan, K., Thelen, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Parity Lost and Parity Regained. In *FAST* (2008), vol. 8, pp. 127–141.

[17] Lakshman, A., and Malik, P. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[18] Le, M., and Tamir, Y. Fault Injection in Virtualized SystemsChallenges and Applications. *Dependable and Secure Computing, IEEE Transactions on 12*, 3 (2015), 284–297.

[19] Marinescu, P. D., and Candea, G. LFI: A Practical and General Library-Level Fault Injector. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on* (2009), IEEE, pp. 379–388.

[20] Massey, J. L. Step-by-step Decoding of the Bose-Chaudhuri-Hocquenghem codes. *Information Theory, IEEE Transactions on 11*, 4 (1965), 580–585.

[21] Mozy Enterprise. Data Loss: Understanding the Causes and Costs.

[22] Patterson, D. A., Gibson, G., and Katz, R. H. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, vol. 17. ACM, 1988.

[23] Ray, I., McConnell, R. M., Lunacek, M., and Kumar, V. Reducing Damage Assessment Latency in Survivable Databases. In *Key technologies for data management*. Springer, 2004, pp. 106–111.

[24] Schriebman, R. Error Correcting Code, 2006.

[25] Segall, Z., Vrsalovic, D., Siewiorek, D., Ysskin, D., Kownacki, J., Barton, J., Dancey, R., Robinson, A., and Lin, T. Fiat - Fault Injection Based Automated Testing Environment. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on* (1995), IEEE, p. 394.

[26] Semiconductor, T. Soft Errors in Electronic Memory-A White Paper, 2004.

[27] Stott, D. T., Floering, B., Burke, D., Kalbarczpk, Z., and Iyer, R. K. NF-TAPE: A Framework for Assessing Dependability In Distributed Systems with Lightweight Fault Injectors. In *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International* (2000), IEEE, pp. 91–100.

[28] Subramanian, S., Zhang, Y., Vaidyanathan, R., Gunawi, H. S., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Naughton, J. F. Impact of Disk Corruption on Open-Source DBMS. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on* (2010), IEEE, pp. 509–520.

[29] Süsskraut, M., Creutz, S., and Fetzer, C. Fast Fault Injections with Virtual Machines.

[30] Van de Goor, A. J., and Al-Ars, Z. Functional Memory Faults: A Formal Notation and a Taxonomy. In *vts* (2000), IEEE, p. 281.

[31] Zheng, M., Tucek, J., Huang, D., Qin, F., Lillibridge, M., Yang, E. S., Zhao, B. W., and Singh, S. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 449–464.

APPENDIX A

Reproducibility

The next steps have to be performed to reproduce the experiment. First, the code repository has to be downloaded from `https://github.com/zen-char/DBMS-Fault-Injection-Framework`, followed by downloading the image dataset from [9]. The following JSON file shown in figure A.1 was used to run the experiment. Multiple scenarios are defined in the actual test file, but are omitted to preserve space.

```
=== cluster_test_file_replication_3.JSON ===
{"server_meta" : {
    "n_nodes" : 3,
    "user" : "user",
    "password": "password",
    "host" : [ "HOST_IP1", "HOST_IP2", "HOST_IP3" ]
  },
  "db_type" : "cassandra",
  "db_version": "3.5",
  "db_meta" : {
    "connection_ip": [ "PRIVATE_IP1", "PRIVATE_IP2", "PRIVATE_IP3"],
    "init" : {
      "class": "SimpleStrategy",
      "replication_factor": 3
    },
    "reuse_keyspace" : true,
    "keyspace" : "files"
  },
  "test_scenarios" : {
    "repetitions": 100,
    "data_type": "files",
    "scenarios": [{
        "FI_type": "bitflip",
        "num_files": 1,
        "excluded_extensions": [
          ".log", ".jar", ".java",
          ".log.0.current", ".so", ".yaml",
          "Index.db" # When targeting non-index, else removed.
        ],
        "exclude_containing": [
          "/jvm/" # Exclude java vm files, they are in directories containing this sub-string.
        ],
        "flips_per_file": 1,
        "start_delay_ms": 0, # Delay FI thread when querying
        "delay_ms": 0        # Delay interval of FI thread between injections
      }, ..., ...]
  },
  "data_to_insert" : { "files" : "datasets/complete_ms_data/" },
  "query_file" : "queries.JSON"
}
```

Figure A.1: The test scenarios as used in the test, providing the results as shown in the paper. Multiple scenarios were used, where the `num_files` value was changed from one to two, five and ten. Four runs were performed with the "Index.db" file added to the excluded extensions list and four without. This to only target non-index and targeting both index and non-index files. Moreover, JSON comments are not supported by the framework yet, thus have to be removed from this example.

The query JSON file is structured in the following way: {"queries": ["query1", ...]}. The queries used to test the system are shown in figure A.3. Note that multi-line strings are not allowed in JSON, thus have to be formatted as a single string. To run the test scenario the steps shown in the figure A.2 have to be followed. The code itself contains comments to provide clarifications of the framework files.

```python
import fi_client as fi
# Create SSH connections.
client = fi.FIClient('cluster_test_file_replication_3.JSON')

# Install dependencies and transfer all needed framework files. This can take a while.
client.setup_framework(install_server_deps=True)
# When setting the execute_startup flag to false the docker container ids and images of
# the non-backup docker instances are obtained. This is needed when the backup
# images have to be created.
client.start_docker_instances(execute_startup=True)

# Insert the data set and fill the verification database. When insert_data is set
# to false, the verification database will only be re-initialized.
client.insert_data_and_verify(insert_data=True)

# Start the fault scenarios! Set the commit_image parameter to false
# when a backup is already made.
client.run_test_scenarios(commit_image=True)
```

Figure A.2: Steps to run the test framework experiments. Most of the code is self explainable and performs the steps as seen in the implementation and design.

As automatically switching from a single to a three node configuration is not supported yet by the framework, this has to be done manually. The following steps have to be performed from a single node to a multi-node configuration.

1. Ensure all Docker DBMS backup instances are stopped and remove them using the i) `docker ps` command to get the running container identifier, ii) `docker stop container_id` to stop the running instance and iii) `docker rm container_id` to clean up the instance. Note that all Docker commands need sudo privileges.
2. Remove the Docker DBMS backup image by using the `docker images` command to get the image id and `docker rmi image_id` to remove the image.
3. Now the mounted directory, the tar backup of the directory and the backup json list have to be removed. Assuming the user is in the user directory:
   `sudo rm -rf fi-framework/db_data && sudo rm -rf fi-framework/backup.tar.gz && sudo rm -rf fi-framework/backup_file_list.json`.
4. Now the framework files can be used to load the right configuration. Create another `FIClient` object using a different configuration, and perform the `start_docker_instances` command on this object (do not forget to set the `execute_startup` parameter to true). Now insert the dataset again and the new test scenario can be executed.

A.1 Test queries

The following Apache Cassandra test queries where used when performing the test scenarios. First, a set of queries is performing `SELECT ALL` queries, while the other subset consists of more advanced queries. The files in each dataset directory are consisting of its directory name suffixed with '`_[number.png|RGB.bpm]`', which allows to create structured range queries. Each table name is named after the directory name where the data files are placed.

```sql
SELECT * FROM balloons_ms;
SELECT * FROM beads_ms;
SELECT * FROM cd_ms;
SELECT * FROM //... all other tables of the data set from a-z;
SELECT * FROM watercolors_ms;

SELECT * FROM clay_ms WHERE file_name > 'clay_ms_10.png' LIMIT 20 ALLOW FILTERING;
SELECT * FROM clay_ms WHERE file_name <= 'clay_ms_10.png' LIMIT 10 ALLOW FILTERING;
SELECT * FROM face_ms WHERE file_name <= ’face_ms_20.png’ LIMIT 15 ALLOW FILTERING;
SELECT * FROM face_ms WHERE file_name > ’face_ms_20.png’ LIMIT 10 ALLOW FILTERING;

SELECT * FROM sponges_ms WHERE file_name = ’sponges_ms_10.png’ LIMIT 1 ALLOW FILTERING;
SELECT * FROM sponges_ms WHERE file_name > ’sponges_ms_10.png’ LIMIT 30 ALLOW FILTERING;

SELECT * FROM beads_ms WHERE file_name >= ’beads_ms_01.png’ LIMIT 33 ALLOW FILTERING;
SELECT * FROM feathers_ms WHERE file_name >= ’feathers_ms_10.png’ AND
                                file_name <= ’feathers_ms_20.png’ LIMIT 10 ALLOW FILTERING;

SELECT * FROM watercolors_ms WHERE file_name >= ’watercolors_ms_05.png’ AND
                                   file_name <= ’watercolors_ms_15.png’ LIMIT 7 ALLOW FILTERING;
SELECT * FROM cloth_ms WHERE file_name >= ’cloth_ms_03.png’ AND
                             file_name <= ’watercolors_ms_23.png’ LIMIT 17 ALLOW FILTERING;
SELECT * FROM cd_ms WHERE file_name > ’cd_ms_15.png’ AND file_name <= ’cd_ms_RGB.bpm’ LIMIT 10 ALLOW FILTERING;
SELECT * FROM paints_ms WHERE file_name > ’paints_ms_11.png’ AND file_name < ’paints_ms_RGB.bpm’ ALLOW FILTERING;
SELECT * FROM pompoms_ms WHERE file_name >= ’pompoms_ms_07.png’ AND file_name <= ’pompoms_ms_19.png’ ALLOW FILTERING;


SELECT * FROM oil_painting_ms WHERE file_name >= ’oil_painting_ms_08.png’ AND
                                    file_name <= ’oil_painting_ms_22.png’ ALLOW FILTERING;
SELECT * FROM balloons_ms WHERE file_name >= ’balloons_ms_01.png’ AND file_name <= ’balloons_ms_15.png’ ALLOW FILTERING;
SELECT * FROM photo_and_face_ms WHERE file_name >= ’photo_and_face_ms_10.png’ AND
                                      file_name <= ’photo_and_face_ms_30.png’ ALLOW FILTERING;
SELECT * FROM thread_spools_ms WHERE file_name <= ’thread_spools_ms_15.png’ ALLOW FILTERING;
SELECT * FROM fake_and_real_beers_ms WHERE file_name > ’fake_and_real_beers_ms_09.png’ LIMIT 20 ALLOW FILTERING;

SELECT * FROM glass_tiles_ms WHERE file_name >= ’glass_tiles_ms_09.png’ AND file_name <= ’glass_tiles_ms_29.png’ ALLOW FILTERING;
SELECT * FROM fake_and_real_peppers_ms WHERE file_name > ’fake_and_real_peppers_ms_21.png’ AND
                                             file_name < ’fake_and_real_peppers_ms_28.png’ LIMIT 5 ALLOW FILTERING;
SELECT * FROM jelly_beans_ms WHERE file_name > ’jelly_beans_ms_03.png’ AND
                                   file_name < ’jelly_beans_ms_19.png’ LIMIT 13 ALLOW FILTERING;
SELECT * FROM superballs_ms WHERE file_name > ’superballs_ms_26.png’ AND file_name < ’superballs_ms_RGB.bpm’ ALLOW FILTERING;
SELECT * FROM chart_and_stuffed_toy_ms WHERE file_name > ’chart_and_stuffed_toy_ms_01.png’ LIMIT 20 ALLOW FILTERING;
```

Figure A.3: Queries used by the test experiment.

APPENDIX B

Adding a different DBMS environment

Adding another DBMS under the testing environment requires to implement Python code in several files. The code which is executed, is determined by if else blocks due time constraints and can be probably be rewritten via means of inheritance.

The framework is implemented under the assumptions that i) a docker image exists of the system under test, ii) the docker image is flexible to allow the creation of clusters if needed and can set several environment variables and iii) a docker logs command exists returning the log files of the DBMS under test.

The following files have to be modified and have to implement the following methods. As this is the first design and implementation, this can maybe simplified and configured differently.

- `fi_client.py` - Multiple functions and the implemented_db_types variable at the top of the FIClient object.
    - `get_docker_run_command` - Provide the command to run the Docker instance of the DBMS with the given parameters.
    - `FIClient -> __init__` - Set DBMS listen port.
    - `FIClient -> _setup_db_docker_cluster` - Get the Docker run command given db meta in the configuration file. The command is used to initial start up the Docker cluster.
    - `FiClient -> _get_log_results` - provide a method to extract a list of relevant logs to be inserted in the local result database.
    - `FiClient -> _create_backup_image` - Do optional preprocessing before a backup image is created. Such as with Cassandra, flush all memory to disk.
    - `FiClient -> ensure_all_running` - Create file in the `./src/databases/DBMS_Name` directory which can indicate all Docker instances are up and running.
- `install_server_deps.py` - Add required dependencies to the list to pull the DBMS image from Docker and Python dependencies.
- `./src/db_server_querying.py` - Used to perform the server Python commands.
    - `create_dbsession_from_type` - Return and define a DBSession object wrapper, to be used to insert and query data. This is covered after this list of needed function changes.
    - `_test_cmd` - Function used to run the database queries while verifying the results with the verification database. Provide an error list typically for the DBMS under test.
- `src/attach_strace.py` - Provide a method to get the process identifier of the DBMS instance.

Additionally to these methods, it is expected that a database python wrapper is written. The wrapper has to be able to provide methods as defined in figure B.1. Additional methods can be implemented and used. A method is recommended to check the connection of the DBMS, if it is ready to use. This method can then be performed via the `__main__` method to be used by the `FIClient.ensure_all_running` method.

```
class DBSession:

  # Create the connection in the init function such that querying of the DBMS will work.
  def __init__(self, ...):
    pass

  # Function to query the database. It is expected to provide the return results in the following format,
  # Which is a bit hard coded for now.
  # {
  #   "timestamp": timestamp of query expected to be a unix timestamp.time() string,
  #   "query_exception1": 1, (optional)
  #   "result": [[ID, File contents (MD5 Hashed), File name], ...]
  # }
  def query_db(self, query, params=None, time_out=.., hash_files=False):
     return_res = {}
     try:
        q_res = self.session.execute_query(...)
        return_res['timestamp'] = q_res.time()
        reutrn_res['result'] = q_res.results()
     except query_exception1:
        return_res['query_exception1'] = 1
     except query_exception2:
        return_res['query_exception2'] = 1
     ...
     return return_res

  # Insert files into the DBMS using a certain schema.
  # The schema used and expected by the server only uses
  # an integer id, file contents, file name
  def insert_files(self, directory):
     file_id = 0
     for root_dir, _, files in os.walk(directory):
       table_name = get_table_name_from_dir()
       paths = [os.path.join(root_dir, name) for name in files]
       for path in paths:
           execute_file_insertion(file_id, path)
           file_id += 1
```

Figure B.1: The definition of the Python wrapper used by the framework.