NoSQL Undo: Recovering NoSQL Databases by Undoing Operations

David Matos Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal

***Abstract*—NoSQL databases offer high throughput, support for huge data structures, and capacity to scale horizontally at the expense of not supporting relational data, ACID consistency and a standard SQL syntax. Due to their simplicity and flexibility, NoSQL databases are becoming very popular among web application developers. However, most NoSQL databases only provide basic backup and restore mechanisms, which allow recovering databases from a crash, but not to remove undesired operations caused by accidental or malicious actions. To solve this problem we propose NOSQL UNDO, a recovery approach and tool that allows database administrators to remove the effect of undesirable actions by undoing operations, leading the system to a consistent state. NOSQL UNDO leverages the logging and snapshot mechanisms built-in NoSQL databases, and is able to undo operations as long as they are present in the logs. This is, as far as we know, the first recovery service that offers these capabilities for NoSQL databases. The experimental results with MongoDB show that it is possible to undo a single operation in a log with 1,000,000 entries in around one second and to undo 10,000 incorrect operations in less than 200 seconds.**

I. INTRODUCTION

Most NoSQL databases aim to provide high performance for large-scale applications [1], [2]. Their ability to split records and scale-out horizontally allows them to maintain performance when dealing with high traffic loads and peaks. In comparison with traditional relational databases, NoSQL databases offer better performance and availability, over strong consistency, relational data and ACID properties [1]. These characteristics make NoSQL databases a good choice for applications with high availability and scalability requirements, but no need of strong consistency and complex transactions.

Currently there are many NoSQL databases.[1] Some of the best known are: Cassandra [3], MongoDB [4], Hadoop HBase [5], Couchbase [6], DynamoDB [7], and Google BigTable [8]. These databases vary mainly in the format of stored data, which can be key-value [7], columnar [3], [8], [9], or document oriented [4]. In terms of scalability, all these systems can be deployed in large clusters and have the ability to easily extend to new machines and to cope with failures.

Most NoSQL databases offer simple recovery mechanisms based in *local logs* and *snapshots* that support data recovery when a server crashes. They also use *global logs* that keep data consistency across replicas. These mechanisms are useful but not sufficient to remove the effect of *faulty operations* from the system state, e.g., of an exploit to a website vulnerability or an incorrect update command by a database administrator that changes or deletes the wrong document. If an incorrect operation is executed and corrupts the database, an administrator may restore an old snapshot that does not include the faulty operation. However, although this solution removes the faulty operation from the database it also discards correct state changes. Even worse, if the faulty operation is detected late, the amount of data lost may be huge. A better solution is to manually execute a command, such as an update or a delete, that removes the effects of the incorrect operation, but this is difficult and time consuming for the administrator. The time it takes to recover a database is critical; if a recovery takes too long it could be impossible to successfully recover the data without collateral damage.

This paper presents NOSQL UNDO, a recovery approach and tool that allows database administrators to automatically remove the effect ("undo") of faulty operations. NOSQL UNDO is a client-side tool in the sense that it does not need to be installed in the database server, but runs similarly to other clients. Unlike recovery tools in the literature [10]–[12], NOSQL UNDO does not require an extra server to act as proxy since it uses the built-in log and snapshots of the database to perform recovery. It also does not require extra meta-data or modifications to the database distribution or to the application using the database. The tool offers two different methods to recover a database: *Full Recovery* that performs better when removing a large amount of incorrect operations; and *Focused Recovery* that requires less database writes when there are just a few incorrect operations to undo. NOSQL UNDO supports the *replicated* (primary-secondary) and *sharded* architecture of NoSQL databases. The tool provides a graphical user interface so that a database administrator is able to easily and quickly find faulty operations and perform a recovery.

To evaluate NOSQL UNDO, we integrated it with MongoDB and conducted several experiments using YCSB [13]. The latter is a benchmark framework for data-servicing services that provides realistic workloads that represent real-world applications. It allows configuring the amount of operations, records, threads and clients using the database. With the experimental evaluation we wanted to compare both approaches to undo incorrect operations (Focused Recovery and Full Recovery), and how both methods perform with different sets of operations. The experimental results show that it is possible to undo a single operation in a log with 1,000,000 entries in around one second and to undo 10,000 incorrect operations in less than 200 seconds.

[1]We use "databases" for short to mean database management systems.

978-1-5090-3216-7/16/$31.00 ©2016 IEEE

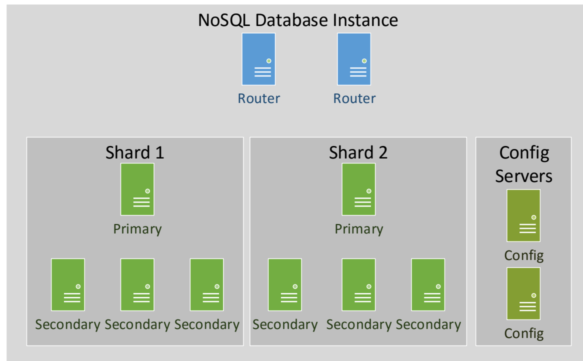

Fig. 1: Example of NoSQL instance with two shards.

This work provides the following contributions. First, a novel recovery approach and tool that supports the distributed – replicated and sharded – architecture of NoSQL databases, with two recovery methods, that does not require a proxy to log operations, and does not require modifications to the service. This is the first system combining these characteristics. It is also the first that supports such a replicated and sharded architecture, that does not require a proxy, and that does not require modifications to the service to support recovery. Second, a detailed experimental evaluation of the tool and comparison of the two recovery methods using YCSB.

II. NoSQL Databases

Most NoSQL databases provide replication, horizontal scaling, unstructured data storage and simple backup and restore capabilities. There are different NoSQL databases, but there are many common elements in their architectures. In some databases several elements can be incorporated in the same server [6], whereas others require that each component of the system is placed in a dedicated server [3], [4]. Some databases [3], [9] may require additional components, such as Zookeeper [14] to manage group membership. Despite their differences, NoSQL databases that provide replication and horizontal scaling tend to have a similar architecture.

A. Architecture

Figure 1 represents the architecture of a common NoSQL database instance with two *shards*. In this configuration each piece of data is divided into slices that are stored in the shards. Each shard is a collection of servers that store the same data, i.e., that are *replicas*. This redundancy provides fault tolerance (no data is lost in case a replica fails) and more performance (any of the servers can respond to read operations). In each replica a server acts as *primary* and coordinates the replication actions of the remaining, *secondary*, servers. The primary server is responsible for keeping data consistent inside a shard.

In order to correctly split data in shards, a special server (or a collection of servers, depending on the complexity of the instance) is responsible for redirecting the requests to the correct shards and divide the records. These servers are usually called *routers*. The routers are the components of the system that interact with the application. If there are no routers alive then the database is inaccessible. To prevent this usually there are several routers. Besides increasing availability of the database, having several routers also increases the performance since if reduces the bottleneck of having a single server responding to every request of the application.

Some NoSQL databases have extra servers that are responsible for recording configuration information of the instance (*configuration servers* in the figure). This allows separation of concerns: while some server are only responsible for storing data, other are responsible for storing metadata and configuration parameters of the instance.

This architecture is very similar to a distributed MongoDB instance. The main difference is that the configuration servers in a Mongo database are grouped in a replica set (primary with a set of secondary servers) which means that all the configuration servers store the exact same information. Cassandra also has a similar architecture except that the configuration servers are not present. The metadata and configuration parameters are stored in the data servers. HBase has a slightly different architecture. Master servers are in a group and manage how data is partitioned to the slave servers (also called region servers). This approach does not separate the servers by data partitions, but instead separates the servers in two groups: master servers and region servers. In Couchbase it is possible to aggregate every component in every server. This way all the servers perform the same tasks. It is also possible to deploy a Couchbase database having each server responsible for a specific task. This way the architecture of a Couchbase database would be like the one in Figure 1.

B. Logging Mechanisms

Most databases have a logging mechanism that records database requests and take periodic snapshots, allowing the recovery of the system in case of failure. NoSQL databases also have logging mechanisms and take snapshots to allow the recovery of an individual server. However, these logs are specific to an individual server, so if the entire database (all servers) fails it is difficult to recover it using the local logs of each server. Besides the local log of each server, most NoSQL databases also have a global log that is used to maintain consistency across all the servers. This global log is not intended to recover the database, but instead to guarantee that all the servers receive the same set of operation in the correct order.

1) Local logs: Any cluster should be prepared for single servers failing unexpectedly. After a failure, a server has to perform fail-over, i.e., to take the required actions to come back to work without interfering with the remaining servers. In order to do that the server must have a diary in which it records every operation it writes to disk, so in case of failure the server only needs to repeat every operation and it should reach the state right before the failure. In some cases this diary is not sufficient since it only contains recent operations, so it is necessary to use a snapshot (a full copy of the server in

a previous moment in time) of the server and complete the missing information with the entries in the diary. The diary in which the server stores the operations is a local log and the information it contains is usually non human readable and specific to the server itself. A simple query can be decomposed in several local log entries corresponding to all the disk writes necessary to execute that query.

2) Global logs: In order to keep data consistent across servers the requests have to be delivered in total order, i.e., all requests delivered in the same order to all servers. On the contrary, simply sending the requests to all the servers might not work since some messages could be lost in the network or delivered out of order. To prevent this most databases use a global log in which they store every request they receive. This log is then used to propagate the operations to all the servers. If a server fails to receive an operation it can later consult the global log and execute it. Some databases have a fixed storage limit for this global log, i.e., it is implemented as a circular array (older operations are overwritten by new ones to prevent the log from growing indefinitely). The way the operations are stored in the global log is usually similar to the way data is stored in the database, meaning that is is possible and fairly easy to perform normal queries over the global log. Random values are converted into deterministic values to guarantee that every operation in the log is idempotent. Besides the operation itself, each log entry contains also a numeric value that allows ordering the executed operations. This numeric value guarantees that every server in the instance is able to reach the same state, since they all execute the operations in the same order.

In most NoSQL databases the global log has a format that is close to the format of the requests, contains the executed queries, is in a human readable form, and stores the operations in the order they were executed. Therefore, in NoSQL Undo we use this log to perform recovery.

III. NoSQL Undo

NoSQL Undo is a client side tool that only accesses a NoSQL database instance when the database administrator wants to remove the effect of some operations from the database, e.g., because they are malicious. The client does not need to be connected to the server in run time since it uses the database built-in logs to do recovery.

A. Undo vs Rollback

Every database provides rollback capabilities, meaning that if an incorrect operation is detected and needs to be removed then it is possible to revert the entire database to a previous point in time prior to the execution of that incorrect operation. The problem with this approach is that every single correct operation executed after the point in time to recover is lost. Figure 2 represents the state of a database with three different documents (D1, D2 and D3). All three documents were updated 6 times, i.e., there are 6 versions of each of these documents in the log of the database. Two of these documents (D1 and D2) were corrupted by malicious operations (red dots). D1 is later updated with valid operations meaning that part of the document may be corrupted while the other part is valid. To use a traditional rollback, the administrator needs to select a point in time prior to any malicious operation, which in this case is when all documents were in version 2. After the rollback the database is clean, however all the documents were reverted to older versions. Every correct operation executed after version 2 is then lost. By using any of the algorithms of NoSQL Undo, the administrator is able to clean both D1 and D2 and still keep every correct operation that was executed after, since both algorithms correct the corrupted documents by removing the effects of the malicious operations, instead of replacing them with older versions.

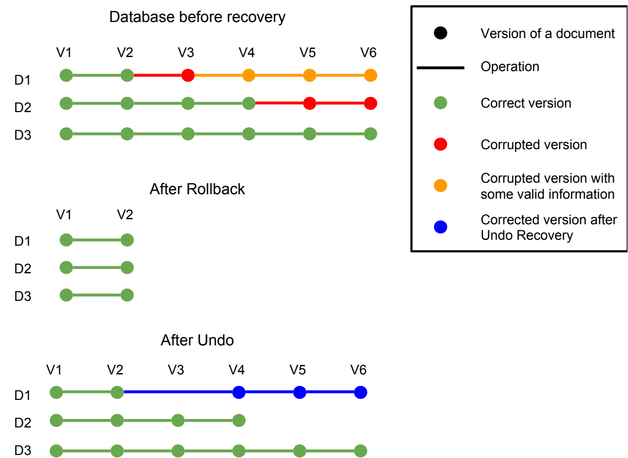

Fig. 2: Comparison of using rollback to recover a corrupted database with using undo to revert incorrect operations.

B. Architecture

Figure 3 presents an example of a NoSQL instance with NoSQL Undo. The architecture is logically divided in two layers: the database layer in which the database runs without any modification to the configuration or to the software; and the support layer, which includes optional modules that can also be deployed to improve the capabilities of NoSQL Undo.

In order to undo undesired operations there has to be a log with every executed operation and snapshots with previous versions of the database. Most recovery systems, such as Operator Undo [10] and Shuttle [15], implement a proxy that intercepts every request to the database and saves these requests in a specific log, independent from the DBMS. That approach may impact performance since every operation must pass through a single server [15]. The proxy may also be a single point of failure; if it fails, the clients become unable to contact the DBMS. It is possible to circumvent this limitation by replicating the proxy, however this introduces more complexity in the system. NoSQL Undo handles this issue using the built-in logs and snapshots to do recovery, so it does not require an additional server (proxy).

Since NoSQL Undo only accesses the built-in log to perform recovery it does not have control of when log entries

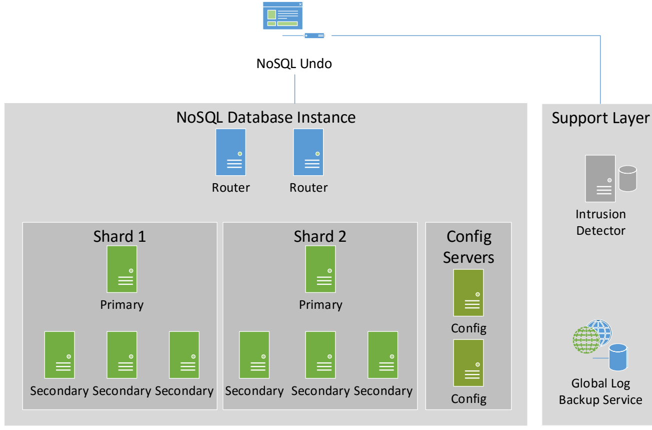

Fig. 3: A NoSQL database with NoSQL Undo.

are discarded. A database administrator may define a high storage threshold for the log, but an unpredictable peak of traffic may be enough to exhaust that limit. To guarantee that any operation can be undone, the log has to be saved regularly. NoSQL Undo does this using a service that runs along with the database instance and listens to changes in the log, the *Global Log Backup Service* (Figure 3). This service is a daemon that is constantly listening to the database for changes and keeping a copy of every log entry in an external database. Every time an operation is executed in the database instance, a new log entry is created and the global log backup service is notified, then it copies the global log record to another database that should be stored in a different server for availability purposes. This backup operation executes concurrently with the clients' operations and does not require the database to lock until the record was successfully stored, so it does not interfere with the original database functioning.

The last component of the architecture is the Intrusion Detector, which provides assistance with the process of identifying operations that need to be undone. We postpone the explanation of this component to Section III-D.

C. Recovery Mechanisms

NoSQL Undo uses two methods to undo the effects of incorrect operations leaving the database in a correct state: full recovery and focused recovery. Both methods take as input a list with operations to undo.

1) Full recovery: The full recovery algorithm is the simplest recovery method among the two. It works by loading the most recent snapshot of the database, then it updates the state by executing the remaining operations, which were previously recorded in a log. The algorithm takes as input a list of incorrect operations that it is suppose to ignore when it is executing the log operations.

Algorithm 1 shows the full recovery procedure. The algorithm takes as input the most recent snapshot before the first incorrect operation and a list with the incorrect operations to undo. In line 1 a new database instance is created using

Algorithm 1 Full recovery algorithm.

1: $recoveredDatabase \leftarrow snapshot$
2: $logEntries \leftarrow getLogEntries(snapshot)$
3: **for** $logEntry \in logEntries$ **do**
4: **if** $logEntry \notin incorrectLogEntries$ **then**
5: $recoveredDatabase.execute(logEntry)$
6: **end if**
7: **end for**
8: **return** $recoveredDatabase$

that snapshot. The log entries are fetched from the global log (line 2) using the *getLogEntries* method. This method returns an ordered list with every log entry after $snapshot$. Correct operations are executed in line 5. When the algorithm finishes (line 8), $recoveredDatabase$ is a clean copy of the database without the effects of incorrect operations. This algorithm is simple and effective, but is not efficient when there are a small number of operations to undo, since it requires every correct operation in the log after the snapshot to be re-executed.

2) Focused recovery: The idea behind Focused Recovery is that instead of recovering the entire database just to erase the effects of a small set of incorrect operations, only compensation operations are executed. A compensation operation is an operation that corrects the effects of a faulty operation. The algorithm works basically the following way. For each faulty operation the affected record is reconstructed in memory by NoSQL Undo. When the record is updated, NoSQL Undo removes the incorrect record and inserts the correct one in the database. On the contrary of Algorithm 1, this algorithm only requires two write operations in the database for each faulty operation.

Algorithm 2 describes the process of erasing the effect of incorrect operations. The algorithm iterates through every incorrect operation (line 1). For each incorrect document it fetches every log entry that affected this record (line 3). For simplicity the pseudo-code assumes that there is no older snapshot and that every operation executed is in the log, therefore the recovered document is initialized empty (line 4). If there was a snapshot, then the recovered document would be initialized as a copy of the incorrect document in the snapshot. Then the reconstruction begins and every correct operation is executed in memory in the recovered record, not in the database (lines 4 to 6). Once every correct operation is executed, the record is ready to be inserted in the database. First the incorrect record is deleted (line 10) and finally the correct one is inserted (line 11).

3) Comparison of the two recovery schemes: Both methods are capable of removing the effects of undesirable operations, but there are differences. Focused Recovery does not require a new database to be created ($recoveredDatabase$) because it does compensation operations in the existing database. For each record affected by incorrect operations, it does two write operations in the database: one to remove the corrupted record, and another to insert the fixed record. On the contrary, with Full Recovery every correct operation executed after the last correct snapshot is executed in a new database instance. In terms of writes in the database, Focused Recovery is much

Algorithm 2 Focused recovery algorithm.

```
1: for incorrectOperation ∈ incorrectOperations do
2:     corruptedRecord ← incorrectOperation.getRecord()
3:     recordLogEntries ← getLogEntries(corruptedRecord)
4:     recoveredRecord ← {}
5:     for recordLogEntry ∈ recordLogEntries do
6:         if recordLogEntry ≠ incorrectOperation then
7:             recoveredRecord ←
               updateRecord(recoveredRecord, recordLogEntry)
8:         end if
9:     end for
10:    database.remove(corruptedRecord)
11:    database.insert(recoveredRecord)
12: end for
```

lighter if the number of incorrect operations is reasonably small. On the contrary, if the number of incorrect operations is greater than the number of correct operations in the log, then using the Full Recovery will be more efficient because there are less write operations to execute in the database (see Section V-B).

Although the algorithms leave the database in a consistent state, a user that has read a corrupted document and suddenly reads the corrected document may believe that the state is inconsistent. To solve this problem, the tool can be configured to leave a message to the users so that they understand why the state suddenly changed [10].

Both algorithms are able to remove the effects of faulty operations but they require the database to be paused, i.e., not executing operations while recovering. If the database keeps serving clients, then data consistency after recovery cannot be guaranteed.

D. Detecting Incorrect Operations

NoSQL Undo provides an interface for administrators to select which operations should be discarded during recovery. This interface provides searching capabilities making it easier to find incorrect operations. An interesting case to use this tool is to do recovery from intrusions.

One of the problems in recovering faulty databases is to detect when the database became corrupted in the first place. Detecting the incorrect operations in a log with millions of operations can be a difficult and error prone task. Searching for regular expressions of possible attacks may not be sufficient since a database administrator may not remember every search pattern to all possible malicious operations.

To cope with this, it is possible to deploy alongside with the NoSQL database an Intrusion Detection System (IDS). This IDS permanently listens to the requests to the database and if they match a certain signature, it logs this operation as suspicious. The most conspicuous case of requests that match signatures are attempts of doing NoSQL injection attacks, which are similar to SQL injection attacks but target NoSQL databases [16], [17]. Later, when NoSQL Undo is being used it first consults all the suspicious operations and suggests them to the database administrator who then decides if they should be removed from the database. This automates the identification process and reduces the time to recovery, which can be critical in a highly accessible database. An example of an IDS that can be deployed in this manner is Snort [18]. Different solutions to detect malicious operations in the log can be used, such as [19]–[21].

E. Recovery Without Configuration

An interesting feature of NoSQL Undo is that it does not have to be configured *a priori* to be able to recover a database. If an incorrect operation is detected soon enough, i.e., while it is still present in the log, then it is possible to remove the effects of this faulty operation without any previous configuration of NoSQL Undo. This is interesting as many organizations do not take preventive measures to allow later recovery.

This approach however has some limitations in comparison to the full-fledged recovery scheme presented in the previous sections. When a recovery service uses specific logging mechanism it is able to store additional information useful for later recovery (e.g., dependencies between operations, origin of the operations and versions of the affected documents). With this extra information it is possible to improve the recovery process as well as provide more information to the database administrator to help him find the faulty operations.

IV. Implementation with MongoDB

To evaluate the proposed algorithms, a version of NoSQL Undo was implemented in Java. This instance of the tool allows recovering MongoDB databases.

A. MongoDB

MongoDB supports replication to guarantee availability if servers fail, and horizontal scaling to maintain performance when the traffic load increases [22], [23]. A set of servers with replicated data is called a *replica set*. In each replica set there is a server that coordinates the replication process called *primary*, while the remaining servers are called *secondary*. Each secondary server synchronizes its state with the primary. It is possible to fragment data records into MongoDB instances, called *shards*, to balance load. A shard can be either a single server or a replica set. Data in MongoDB is structured in *documents*. Each document contains a set of key-value pairs. A set of related documents is a *collection*. The documents in a collection do not need to have the same set of key-value pairs nor the same type, as opposed to relational DBMSs that impose a strict structure to the records (rows) of a table.

MongoDB uses two logging mechanisms: *journaling*, which is the *local log* used to recover from data loss when a single server crashes; and *oplog*, which is the *global log* that ensures data consistency across replicas of a replica set. From time to time a database copy, a *snapshot*, is saved in external storage and the journal logs are discarded, otherwise they would grow indefinitely.

In relation to journaling, MongoDB uses a local log called journal to recover a single server that failed unexpectedly. Every time MongoDB is about to write to disk it first logs the write to a journal file. The journal is then used to recover a

single replica that failed without intervention by other servers. The journal file contains non-human readable, binary, data so it is hard to process.

Oplog is the global log used by MongoDB. The primary server uses it to log every operation that changes the database. The oplog collection has limited capacity, so MongoDB removes older log entries. The oplog is stored in a (MongoDB) database, not in a file as logs usually are.

B. NoSQL Undo with MongoDB

The integration of NoSQL Undo with MongoDB is pretty straightforward and follows the architecture in Figure 3. We installed the Global Log Backup Service in the same network of the database. NoSQL Undo accesses both the database and the backup of the log in order to perform recovery and undo operations.

We used two scenarios in the experiments. *Scenario 1* corresponds to Figure 3. It is a fully distributed instance of MongoDB that was installed in 10 EC2 machines of *Amazon AWS*. The database is divided in two shards, each one containing 3 servers (one primary and two secondary). The configuration servers are grouped in a replica set since that is how MongoDB uses the configuration servers. Finally there is a single router (unlike the 2 in the figure), which is a MongoDB server that is responsible for redirecting the requests to the correct servers. All the servers have the exact same configuration: t2.small instances with a 1 vCPU and 2GB of memory and running Ubuntu 14.04LTS.

We also used a second configuration in our experimental evaluation for diversity: *Scenario 2*. In that scenario NoSQL Undo was deployed in Google Compute Engine. The deployment was composed by a single replica set with 4 machines: 1 primary and 3 secondaries. Each machine had 1 vCPU and 4GB of memory. The OS used was Debian 8.

V. Experimental Evaluation

The objective of the experimental evaluation was to answer the following questions using the implementation of NoSQL Undo for MongoDB: (1) what is the performance trade-off between Focused Recovery and the Full Recovery mechanism? (2) How long does it take to undo different numbers of operations? (3) How does the number of versions of a file affects the time to recover?

To inject realistic workloads we used YCSB [13]. YCSB is a framework to evaluate the performance of different DBMSs using realistic workloads. Some examples of DBMSs supported by YCSB are MongoDB, Cassandra [3], Couchbase [6], DynamoDB [7], and Hadoop HBase [5]. We choose this framework because it is widely adopted for benchmarking NoSQL DBMSs, it provides realistic workloads, and has several configuration options (number of operations, amount of records to be inserted, distribution between reads and writes).

The YCSB workloads used in the experiments were: (A) update heavy, composed by 50% reads and 50% writes; (B) read mostly, with 95% reads and 5% writes; (C) only read, without write operations; (D) new records inserted and the most read, simulating social networks and forums where users consult the most recent records; (E) short ranges, where read operations fetch a short range of records at a time, like a conversation application, blogs and forums; and (F) records updated right after read, simulating social networks when users update their profile.

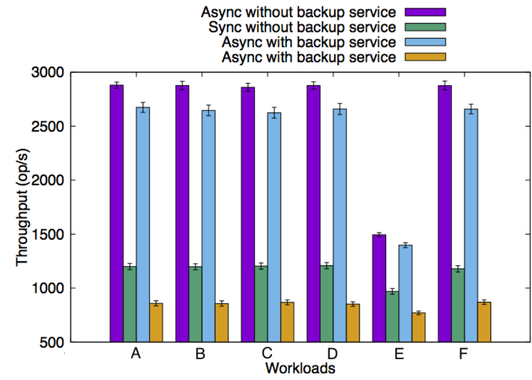

Fig. 4: Overhead of using the Global Log Backup Service with a confidence level of 95%.

MongoDB offers two different interfaces: *synchronous*, where only one operation can be submitted at a time, and *asynchronous*, where operations can be executed in parallel.

A. Global Log Backup Service Overhead

The Global Log Backup Service runs alongside with the database. It listens for changes in the log, so when an operation is executed in the database server, the Global Log Backup Service is notified and saves the operation in external storage. To evaluate the throughput overhead of using this backup service we executed several workloads of YCSB 10 times each using Scenario 2.

Figure 4 presents the average throughput (operations per second) of 10 executions of several workloads of YCSB using both the asynchronous and the synchronous driver with a confidence level of 95%. The cost of having this additional service varies from 6% to 8% when using the asynchronous driver, and from 20% to 30% when using the synchronous driver. The overall throughput seems acceptable given the advantages of storing every executed operation in the database.

In terms of storage, after executing every workload of YCSB the database occupied 100MBs in disk while the global log backup occupied 120MBs. The overhead in terms of storage is considerable (120%), but this is an unavoidable cost of supporting state recovery.

B. Focused Recovery versus Full Recovery

To evaluate how Focused Recovery performs in comparison with Full Recovery a set of operations were undone from the database using both algorithms in Scenario 1. The size of this set varied from 1 to 10,000. Each case was repeated 10 times. The goal of using different sets of incorrect operations to undo was to understand how both algorithms perform when they

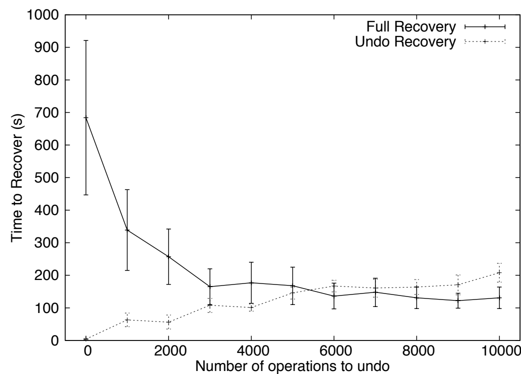

Fig. 5: Focused Recovery performance vs Full Recovery time with a confidence level of 95%.

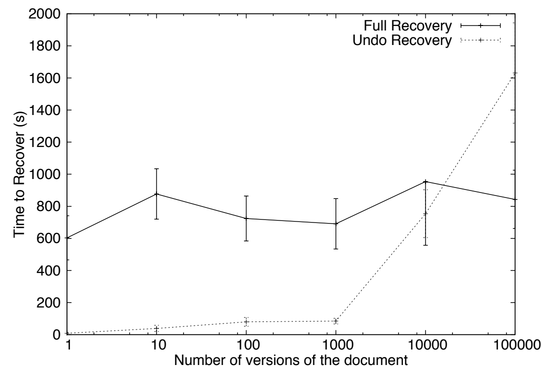

Fig. 6: Focused and Full recovery a document with different versions with 95% confidence level.

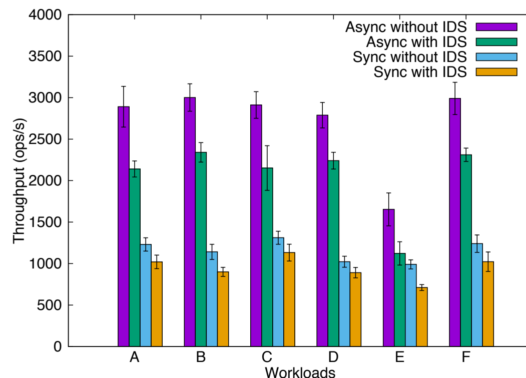

Fig. 7: Overhead of using Snort to detect incorrect operations in a MongoDB Cluster.

undo from just a few operations to almost every operation in the database.

Figure 5 shows the average time to recover using both methods, with a confidence level of 95%. The Full Recovery method performs better as the number of operations to remove increases, which makes sense as less operations have to be executed. On the other hand, the Focused Recovery method performs a lot better when there are just a few operations to be removed and it degrades the performance linearly as the number of operations increases. Focused Recovery takes almost a second to remove 1 operation, whereas Full Recovery takes around 700 seconds. Both methods achieve a similar performance around 5,000 operations to be removed. This result shows that for a small number of operations to be removed Focused Recovery is a better choice, but if more than 60% of the operations are incorrect than the Full Recovery method should be used.

C. *Recovery with Different Versions*

The focused recovery method reconstructs every document affected by incorrect operations, meaning that if a document has a thousand versions and one of them is incorrect, then the focused recovery method needs to re-execute the remaining 999 operations in order to reconstruct the document correctly. To evaluate how focused recovery is able to remove incorrect operations in documents with different number of versions, we executed both the focused and the full recovery methods in a document varying the number of versions from 1 to 100,000 in steps of 10. Each recovery execution was repeated 10 times. These experiments were conducted in Scenario 1. The average recovery time of each execution can be seen in Figure 6. The time to recover increases exponentially for the Focused Recovery, while it remains almost constant for the Full Recovery. This result was expected since the Full Recovery needs to undo one incorrect operations in every case. Focused Recovery has more work to do if the number of versions of a document increases. This is because it needs to reconstruct the affected record.

D. *Intrusion Detection Overhead*

Using an IDS to tag incorrect operations facilitates the database administrator job. When an alarm is triggered the administrator consults the IDS log and can immediately undo incorrect operations, instead of browsing the entire database log for incorrect operations. To evaluate the cost of using an IDS to detect incorrect operations, we set up an extra machine (with the same characteristics of the others) running Snort in Scenario 1. We then added 10 rules to Snort and executed every YCSB workload.

Figure 7 shows the overhead of using Snort. The throughput is degraded by 10 to 30%. It is a considerable cost given the benefits of allowing the database administrator to recover a database immediately without loosing time searching in the database log for incorrect operations.

VI. Related Work

The use of logs and snapshots to recover databases after a crash is far from new and is covered in textbooks in the area, e.g., [24]. This work follows a more recent line of work on recovering databases [11], [25], operating systems [12], web applications [15] and other services [10] by eliminating the effect of undesirable operations, but not of the rest of the operations. We discuss some of these works next.

Operator Undo seems to be the first work in this line [10]. It is an architecture that aims to provide administrators the ability to undo operations. The authors argue it is generic, but it has been applied specifically to email servers. To remove the effect of an operation –a verb– from the state, the system is rollback to the moment immediately before that verb (Rewind), the verb is removed (Repair), then every entry in the log after that point is re-executed until the present moment (Replay).

ITDB is a self-healing database built on top of a commercial DBMS [25]. It detects intrusions and is able to isolate the effect of attacks. It also provides recovery mechanisms that repair the effects of intrusions in useful time. ITDB provides a repair mechanism that removes the effect of intrusions similarly to NoSQL Undo.

Phoenix is another recovery system for databases [11]. The operations executed on a database may depend on each other, e.g., a write operation may modify record a using values read from record b. Phoenix tracks these dependencies and considers them during recovery. It consists in a PostgreSQL-based database that gathers record dependencies while logging requests.

Shuttle is a recent recovery system form applications deployed in Platform-as-a-Service clouds [15]. It combines the use of snapshots with selective re-execution of log operations to recover a web application and undo the effects of intrusions. It considers the existence of more than one server (application servers) and a back-end database, unlike the previous systems.

NoSQL Undo is inspired in these systems but has several crucial differences. Firstly, it is the first to support a distributed DBMS, that can be replicated for fault tolerance (replica set) and performance (sharding), where most of the other systems consider a single server. Secondly, it is the only system of the kind that does not require a proxy or modifications to the service to support recovery (MongoDB in this case). Instead, it takes advantage of the built-in log and uses an external component to guarantee that no log operations are lost. Thirdly, except for Shuttle it is the only system that supports two modes of recovery and NoSQL databases.

VII. Conclusion

This paper presents a tool that allows the database administrator to remove incorrect operations from a MongoDB database. It runs as a client of the DBMS and uses its built-in log and snapshots to do recovery. The tool provides two different approaches to recover a database: Focused and Full Recovery. Both methods are capable of recovering databases, but there is a trade-off between performance and number of operations to undo.

Acknowledgements This work was supported by the European Commission through project H2020-653884 (SafeCloud) and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

References

[1] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[2] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," in *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2013, pp. 15–19.

[3] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[4] "MongoDB." [Online]. Available: http://www.mongodb.org

[5] T. White, *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[6] M. Brown, *Getting Started with Couchbase Server*. O'Reilly, 2012.

[7] S. Sivasubramanian, "Amazon DynamoDB: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 729–730.

[8] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, p. 4, 2008.

[9] M. Vora, "Hadoop-HBase for large-scale data," in *Proceedings of the 2011 IEEE International Conference on Computer Science and Network Technology*, 2011, pp. 601–605.

[10] A. Brown and D. Patterson, "Undo for operators: Building an undoable e-mail store," in *Proceedings of the USENIX Annual Technical Conference*, 2003, pp. 1–14.

[11] T. Chiueh and D. Pilania, "Design, implementation, and evaluation of a repairable database management system," in *Proceedings of the 21st IEEE International Conference on Data Engineering*, 2005, pp. 1024–1035.

[12] T. Kim, X. Wang, N. Zeldovich, and M. Kaashoek, "Intrusion recovery using selective re-execution," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010, pp. 89–104.

[13] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 143–154.

[14] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[15] D. Nascimento and M. Correia, "Shuttle: Intrusion recovery for PaaS," in *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems*, 2015, pp. 653–663.

[16] OWASP, "Testing for NoSQL injection," https://www.owasp.org/index.php/Testing_for_NoSQL_injection.

[17] J. Scambray, V. Lui, and C. Sima, *Hacking Exposed Web Applications: Web Application Security Secrets and Solutions*. Mc Graw Hill, 2011.

[18] M. Roesch, "Snort: Lightweight intrusion detection for networks." in *Proceedings of LISA'99: 13th Systems Administration Conference*, 1999, pp. 229–238.

[19] S. Lee, W. Low, and P. Wong, "Learning fingerprints for a database intrusion detection system," in *Computer Security – ESORICS 2002*. Springer, 2002, pp. 264–279.

[20] Y. Hu and B. Panda, "A data mining approach for database intrusion detection," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, 2004, pp. 711–716.

[21] ——, "Identification of malicious transactions in database systems," in *Proceedings of the 7th International Database Engineering and Applications Symposium*, 2003, pp. 329–335.

[22] K. Chodorow, *MongoDB: The Definitive Guide*. O'Reilly, 2013.

[23] "MongoDB Manual." [Online]. Available: https://docs.mongodb.org/manual/

[24] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Pearson, 2008.

[25] P. Liu, J. Jing, P. Luenam, Y. Wang, L. Li, and S. Ingsriswang, "The design and implementation of a self-healing database system," *Journal of Intelligent Information Systems*, vol. 23, no. 3, pp. 247–269, 2004.