A new model for testing CRUD operations in a NoSQL database

María Teresa González-Aparicio

Computing Department
University of Oviedo
Gijón, Spain
maytega@uniovi.es

Muhammad Younas

Oxford Brookes University

Oxford, United Kingdom
m.younas@brookes.ac.uk

Javier Tuya

Computing Department
University of Oviedo
Gijón, Spain
tuya@uniovi.es

Rubén Casado

R&D Department
Treelogic
Asturias, Spain
ruben.casado@treelogic.com

Abstract— **NoSQL databases provide high availability and efficiency in data processing but at the expense of weaker consistency. In this paper, we propose a new approach in order to test NoSQL key/value databases in general and their CRUD operations in particular. We design a new context-aware model that takes into account the contextual requirements of clients (users) and the NoSQL database system. Accordingly, we develop a transaction model and testing criteria in order to test NoSQL databases by taking into account transactional and non-transactional CRUD operations. Results from testing criteria are used to analyse the trade-off between availability and consistency of NoSQL databases. In addition, these are used to help NoSQL database users and developers to choose between transactional and non-transactional CRUD operations.**

Keywords—Context-aware; CRUD; NoSQL key/value database; Riak; transaction

I. INTRODUCTION

Data has become one of the key sources for economic growth in the 21 century. Public and private sectors organizations and enterprises can achieve profitable results through an efficient and intelligent analysis of data which come from various sources such as web, cloud, IoT, and online social media among others [1]. This phenomenon has led to the concept of big data which has a higher order of magnitude and a variety of origins generating such data in different structures and different formats. Big data is characterized by 3Vs (Volume, Variety, Velocity) or 4Vs (Volume, Variety, Velocity, and Value) models [2].

Traditional databases, based on ACID (atomicity, consistency, isolation durability) properties, do not fit well to the characteristics of big data. One of the main reasons is the stronger consistency enforced by traditional databases. However, in big data environment, enforcing stronger consistency can affect high availability and efficiency, which are equally important given the high volume, variety and velocity of big data. Therefore, a new generation of databases, named NoSQL ("Not only SQL"), has been developed as the core technology for storing and processing big data. NoSQL databases are generally schema-free and support replication and eventual consistency using correctness criteria such as BASE (Basic, Availability, Soft-state, Eventual consistency) [3].

These have been categorized as document store, column families, key/value, graphs and multimodel databases.

In order to process big data, NoSQL databases adopt CRUD operations (Create, Read, Update and Delete) from traditional databases. Nevertheless, in the Internet-based big data applications, CRUD operations should also manage additional features such as data replication over different nodes, and concurrent requests from several clients requesting a common (shared) data. Data replication is required to provide high availability and efficiency as same data can have multiple copies. For instance, efficiency can be achieved as multiple requests can access different copies of the same data. Similarly, data replication can provide data availability in situation when any part of data centres (hosting big data) may face temporary unavailability due to failures of network communication or software systems. Though data replication is a viable solution it may result in data inconsistency or stale data, in the case of updating replicated data. Therefore, there should be a tradeoff between data consistency, availability and efficiency.

In this paper, our research focuses on testing the CRUD operations of NoSQL key/value databases. In such databases, access to data is based on a primary key. There exist a number of NoSQL key/value databases such as Aerospyke, Berkeley DB, Redis, Voldemort, Dynamo and Riak among others. In this paper, we use Riak for implementation of the proposed approach. Riak is widely used and is based on Dynamo. It is remarkable that Amazon, one of the biggest e-commerce sites, is using a proprietary key/value database, Dynamo [4], for its online services such as shopping carts, session management, product catalog and customers preferences.

This paper proposes a new model for testing the CRUD operations in NoSQL key/value databases. The proposed model is context-aware as it takes into account client and system contextual information. For instance, if during the execution of a CRUD operation, client context is not met then the operation should be automatically rolled back. However, existing NoSQL database systems do not provide an automatic internal mechanism to rollback CRUD operations. It is left to the developer (or programmer) to implement a corresponding rollback procedure for CRUD operations.

The key contributions of this paper include:

- A new context-aware model for testing NoSQL key/value databases by taking into account transactional and non-transactional CRUD operations.

- Analysing the trade-off between availability and consistency of NoSQL key/value databases and thus helping clients and developers to choose between transactional and non-transactional CRUD operations.

- Evaluation of the proposed model using a widely used NoSQL key/value database, Riak (by Basho). Riak, based on Dynamo [4], is an open source which provides high scalability and availability.

The reminder of this paper is organized as follows. Section II presents an overview of NoSQL key/value databases and key features of Riak. Section III reviews and analyses related work. Section IV presents a general transactional model for CRUD operations. Section V illustrates an example of how the general transactional model could be used for generating tests in order to test CRUD operations. Conclusions and future work are described in Section VI.

II. OVERVIEW OF NoSQL KEY/VALUE DATABASES

NoSQL databases process large volume of data and generate results in real time such as analysis of millions of tweets or processing of live road traffic data. Such applications demand high response time, scalability and availability but at the expense of sacrificing stronger consistency. NoSQL databases are used for different applications and have different types including document stores, column families, key/value, graphs and multimodel databases. In this paper, our research is focused on key/value databases, where data access is based on a primary key. Indeed, many Internet services have been designed and implemented using key/value databases like best seller lists, shopping carts, product catalog, and so on [4]. Nevertheless, developers (programmers) have to choose a key/value database (Redis, Belkeley DB, Voldemort, Dynamo and Riak, among others) that best suits their needs. This is due to the fact that each of them provides a distinct operating policy for data management. For instance, handling of transactions and consistency management varies from one key/value database to another as shown in Table I.

TABLE I.   FEATURES OF SOME KEY/VALUE DATABASES

	Redis	Voldemort	Dynamo	Riak
Transactions	Execution of group of commands	No	No	No
Roll back operations	No	No	No	No
Consistency adjustment	Yes	Yes	Yes	Yes

In this paper our work focuses on Riak for two main reasons: Firstly, it is an open source key/value database. Secondly, it is based on Dynamo, which is used for large

systems such as Amazon. Riak uses cluster which is made of multiple physical nodes, although each node is logically divided into virtual nodes. The set of key/value pair is assigned over the different virtual nodes by a hash function. Data scalability and availability are through a partition and replica technique. Indeed, each pair is replicated at N virtual nodes, which are located in distinct physical nodes. Moreover, key/value pairs are grouped in a namespace named "bucket" in order to allow storing different pairs with the same key but in different buckets. In addition, buckets are grouped in another namespace named "bucket type". This mechanism allows setting of system behaviour properties in all key/value pairs which have been stored inside the bucket that belongs to a specific bucket type. For instance, properties like the number of replicas (N) and the level of consistency/availability could be set at the bucket type in order to establish when a read ('r') or a write ('w') operation will be considered successful or not. For instance, if r and w have a value lower than N, then the system will never reject the operation as long as there are at least r and w nodes available. Several instances of Dynamo set (3, 2, 2) as a common configuration for (N, r, w) in order to achieve satisfactory levels of performance, consistency and availability [4]. If $r + w >$ N, the system will be quorum-like, on the contrary if $r + w \leq$ N the system will provide better latency.

III. RELATED WORK

NoSQL databases and big data platforms provide availability, low latency, partition tolerance and high scalability. In addition, they deal with distributed databases, where the maintenance of data consistency is one of the key factors for achieving a quality system response. They are based on several types of models depending on the level of consistency, such as: strong consistency or linearizability (global real-time ordering) [5], sequential consistency (global ordering) [6], causal+consistency (combination of causal consistency and convergent conflict handling) [7], causal consistency (partial orderings between dependent operations) [8], FIFO consistency (partial ordering of an execution thread) [9], per-key sequential consistency (global order operations for each key) [10], and eventual consistency (convergence to an agreement, which does not order concurrent operations) [11].

Generally, NoSQL key/value databases leave the responsibility of managing transactional data access to the client side application developer. However, this may lead to inconsistent results if the client side application developer incorrectly implements transactional data access operations. In order to address this issue, various transactional NoSQL key/value technologies have emerged. Such technologies handle transactions at three different levels: data store, middleware and client side. Systems such as Spanner [12], COPS [7], Granola [13] and HyperDex Warp [14] have been developed to handle transactions at the data store level. At the middleware level, transactional application is hosted in the cloud with a controlled set of data. For this group, systems like Google Megastore [15], G-Store [16], Deuteromony [17] and CloudTPS [18] have been developed. Finally, for the client side, Percolator [19] and ReTSO [20] have been developed.

In this paper, we advocate that the execution of transactions must be context-aware due to the fact that not all transaction

operations have the same client contextual needs. For instance, in an e-commerce application it is logical that a purchase operation must be executed with a stronger consistency than a simple browse operation. A hierarchical transactional model could be a solution, where transaction operations are classified into four different data consistency levels [21]. If we consider the hierarchy from the strongest to the weakest data consistency level, the transaction management protocols applied to each level are as follows: Snapshot Isolation based on serializable transactions where ACID properties are ensured (SR level), Causal Snapshot Isolation (CSI) referred to as the fork-join model [22], Causal Snapshot Isolation with concurrent commutative updates (CSI-CM) and asynchronous updates (ASYNC).

IV. THE PROPOSED MODEL

In NoSQL applications, data are generally replicated over multiple hosts in order to provide high availability and efficiency. However, data replication complicates the process of managing CRUD operations in situations where data are concurrently read and updated by multiple clients.

In existing NoSQL databases, it is the responsibility of the client side developers to manage several replicas (versions) of data and the related concurrency issues. However, this may lead to a high risk of causing implementation errors and erroneous results when concurrency and updates are managed by different developers. The work presented in this paper attempts to minimise this problem by introducing a new model which supervises/monitors the execution of CRUD operations in order to ensure that the system response (outcome of CRUD operation) meets the expectation of clients (users) of the NoSQL databases. The architecture of the proposed model is depicted in Fig. 1. The main components of the architecture are described as follows.

Coordinator: Coordinator is the main component (or module) that manages the overall execution of CRUD operations. It ensures that every CRUD operation is executed in such a way that the outcome of the whole operation is correct and that the desired context is also fulfilled. For instance, execute a CRUD operation so that it maintains the required level of consistency as well as system response time. If CRUD operation cannot be processed in the desired time, then it needs to be rolled back.

As shown in Fig. 1, Coordinator interacts with other components in order to execute CRUD operation. It interacts with the module, Data and Semantic Rules, which contains information about data design and related semantic rules. This module provides information about the different types of data the application should handle and the relationships between them. Semantic rules are used to establish and identify relationships between data entities. For instance, if one (data) entity belongs to a user and another belongs to a computer and a relationship exists between both entities, then the semantic rules would be to establish the level of permission or access rights that a specific user (administrator, anonymous, etc) has to a particular computer. In addition, data and semantic rules can also provide information if there is any failure of a node in the system. Failure of nodes may lead to an erroneous interpretation of the actual data value, i.e., data values before and after node failure. Therefore, if the Coordinator has

appropriate knowledge (using semantic information) of the data and failures then it could correctly process the operations, which are accessing the data effected by failures of nodes.
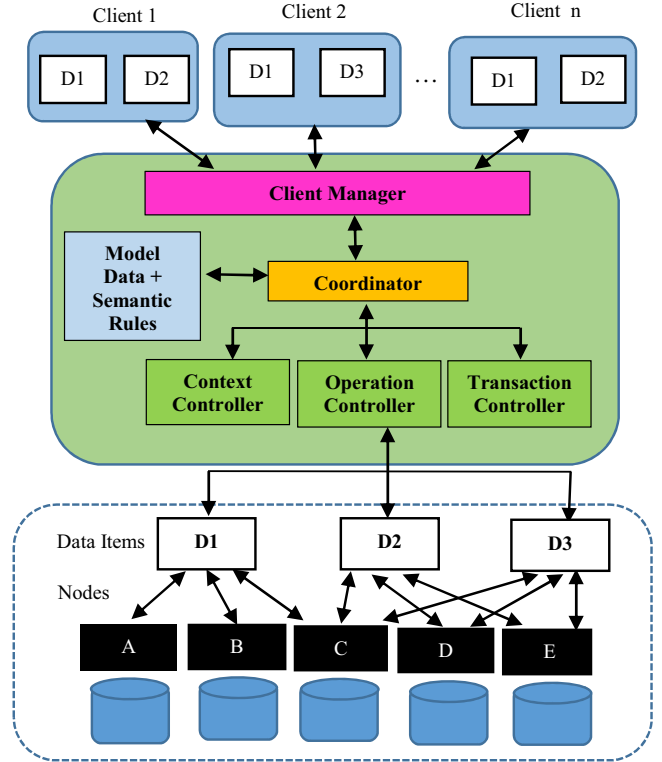

Figure 1. The proposed model for CRUD operations.

Context Controller: This deals with managing contextual information related to the execution of CRUD operations and the data. Sources of context information are systems (NoSQL databases) and their clients (users and developers). For instance, context information can come from system configuration – i.e., how a particular system (or NoSQL database) is configured to execute CRUD operations and to manage the data internally, such as creating and processing different versions (replicas) of the same data, the distribution of replicas among different nodes, and the client access to the data. Client context information can be related to the required response time of their applications or the level of consistency of the data they require. Client context information is related to, and should be in line with, system context. For example, the level of consistency required by the client application should be based on the level of consistency supported by the system. In the proposed model, system context and client context are referred to as internal and external context respectively. A description of each context is described as follows:

- **Internal or implicit context:** it is a set of parameters that allows the configuration of the behaviour of the system with respect to two properties, consistency and availability. For instance, if a NoSQL key/value database manages the administrators who are allowed to access to a corporation network, the system behaviour could be

different depending on how the parameters N, *r*, *w* are set. If (N, *r*, *w*) are set to (3, 1, 1), then the write operation would be successful when only one node is written and the other two are updated asynchronously later on. Therefore, if a registered administrator is unsubscribed, the second and the third node would maintain the administrator in the database unlike the first node which would drop him out. This is why the system becomes temporarily inconsistent. If during this uncertain situation, the first node fails, the administrator who is unsubscribed would be allowed to log in to the network, owing to the fact that the read operation is made from the second node (r=1). This fact reflects a lack of access security. On the contrary, if the parameters (N, r, w) are set to (3, 2, 2), the administrator would be dropped out from the first and the second node (w=2), but would temporarily exist on the third node. If the first node fails, this situation is completely different from the previous one (N=3, r=1, w=1), because the read operation (r=2) would be made from the second (latest version) and the third node (stale version), and the newest version would have priority over the oldest version. Therefore, the unsubscribed administrator would not be allowed to access the corporation network. As a consequence, the configuration of the system parameters establishes a specific level of consistency and availability.

- **External or client context:** it is the environment which establishes where, when and how a transactional/non-transactional CRUD operation is executed. The goal is to guarantee a set of functional and non-functional application requirements. On the one hand, the execution time of a CRUD operation could be limited (non-functional requirement). If the time expires, one solution could be to execute the CRUD operation a second time or abort it. On the other hand, when a CRUD operation finishes its execution, the actual and the expected outcome must agree (functional requirement). If the CRUD operation is successful the execution is finished (non-transactional operation), otherwise a rollback operation must be launched in order to guarantee the stability of the system (transactional operation).

In addition, context information can be classified into vital and non-vital [24]. Vital context is that it must be fulfilled for a CRUD operation to be considered as successful. Non-vital can be flexible and it may or may not be fulfilled depending on the situation. For instance, a session management system must not allow any user to log in without a complete registration into the system (N=3, r=2, w=3). In this situation, a strong consistency is required (consistency=vital, availability=non-vital). Nevertheless, some online applications like those orientated towards e-commerce would like to work with a weak consistency policy. For example, if two clients buy the same product at the same time and only one item is left, and the configuration of the system parameters is set with a weak consistency (N=3, r=1, w=1), then there is a possibility (if first node fails) that both clients buy the same product. In this situation, some e-commerce companies would refund to

potential clients (consistency=non-vital, availability=vital). In short, the level of consistency/availability will be the same throughout the execution. This fact will be referred to as "stable behaviour".

Nevertheless, during the execution of the online application the configuration of the system parameters could change. The period of time when client's requests occur (peak hours, weekends, holidays, etc) and the type of object requested, are examples of two possible factors to take into account. For instance, in an e-commerce application which deals with a product catalog, some products could be published with a lower price as a part of promotion or advertisement. Thus, it could be advantageous to guarantee a weak consistency and a strong availability during the period of promotion. However, both properties could work the other way round when the promotion ends. Therefore, the properties of consistency and availability can fluctuate between vital and non-vital. This is because their values vary during execution, and thus both properties could be named as hybrid. This fact will be referred to as "dynamic behaviour".

Operation Controller: it is in charge of the communication with the NoSQL key/value database when transactional/non-transactional CRUD operations are executed by the Coordinator. It proceeds with the execution and sends the new database state to the Coordinator.

Transaction Controller: in collaboration with the above modules, Transaction Controller has to deal with the transactional features of the CRUD operations – i.e. a CRUD operation can be successfully executed if it meets the required context and transactional policies. If not, then the operation has to be rolled back in order to maintain the required level of consistency.

Fig. 2 models one of the execution scenarios of a transactional CRUD operation.

Figure 2. General transactional model for CRUD operations.
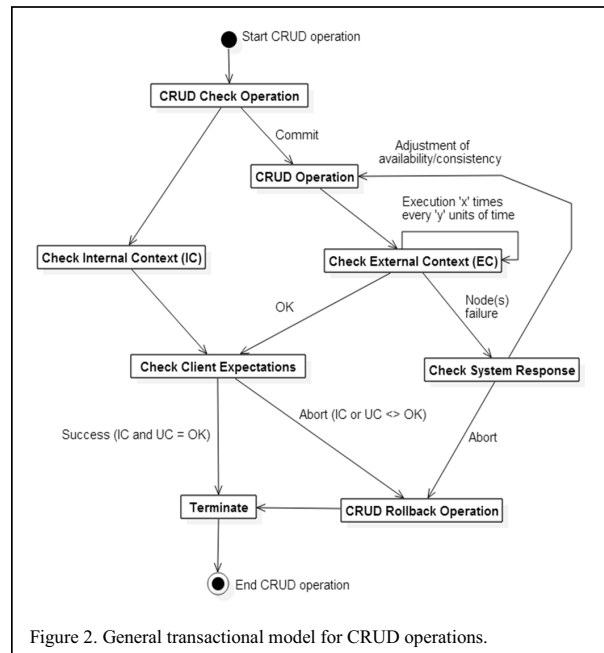
In general, the following functions are involved in execution of this scenario.

1. CRUD Check Operation (Coordinator): it checks the CRUD operation when it is submitted to the system by a client. In other words, before starting the CRUD operation, the Coordinator checks the operation and its related contextual requirements.

2. CRUD Operation (Operation Controller): it proceeds with the execution of transactional/non-transactional CRUD operations on the NoSQL key/value database.

3. Check External Context (Operation Controller + Context Controller): it is responsible for checking whether the execution of a CRUD operation obeys non-functional application requirements (part of the external context). Indeed, due to the nature of distributed systems (propagation of information, recovery of nodes, etc), a CRUD operation could not be executed successfully. Then, if the Context Controller detects that the execution of the operation exceeds its time, it must inform the Coordinator, which must order a retry of the execution after a period of time (µsg, msg, sg, hours, etc). If the problem persists, the execution could be repeated a number of times. The length between requests will depend on the characteristics of the application. Finally, if the operation cannot be executed, then the problem might have been that one or more nodes were not able to recover.

4. Check System Response (Coordinator + Operation Controller): If a CRUD operation is executed, but and it is not able to be finished, then it is highly likely that one or more nodes have failed. In this case, the operation could be aborted or on the contrary the Coordinator could order the Context Controller to make an adjustment in the level of consistency/availability (internal context). For instance, suppose a database which manages the set of administrators of a company with $(N, r, w) = (3, 3, 3)$ (strong consistency, weak availability). If an administrator needs to be registered but the first node fails, then the write operation will fail. If the registration needs to be done as soon as possible, one possibility could be to decrease the level of consistency for the (uncertain) write operation from 3 to 2 (w=2, weaker consistency and stronger availability).

5. Check Client Expectation (Coordinator + Context Controller + Transaction Controller + Operation Controller): the Operation Controller informs the Coordinator about the current state of the database after the CRUD operation has finished. The Coordinator then has to check if the new state is consonance with the application design (Model Data+Semantic Rules) and both the internal and the external contexts (Context Controller). If the operation matches all features, then the operation will be considered as successful. Otherwise, the operation must abort and a transaction must rollback the CRUD operation (Transaction Controller).

V. APPLICATION SCENARIO OF THE GENERAL TRANSACTIONAL MODEL

The general transactional model will provide useful information for detecting defects in the implementation of CRUD operations on NoSQL key/value database. In this section, we will use the model to generate a number of tests that cover the different paths, which are described in the model.

A. Data Modeling

A real-world example, named as "authorization and access control" [25], was implemented in order to look into how the correctness of the CRUD operations could be checked using tests derived from the general transactional model. The experiment was conducted with the following hardware/software: a CPU core with 2.4 GHz Intel(R) Core(TM) i7-5500; the operating system Ubuntu 14.04 LTS of 64 bits; Eclipse Luna 4.4.2 as IDE (Integrated Development Environment); Oracle Java 7 as the programming language; and the NoSQL key/value Riak (by Basho) 2.1.1 as database management.

The authorization and access control system is composed of three types of entities, which are identified as administrators (A), groups (G) and enterprises (E). Each administrator must belong to at least one group, and each group will stablish what type of access permission will exist between an administrator and a specific enterprise. In addition, enterprises are related with a parent-child relationship. Indeed, three distinct access restrictions could be set between a group and an enterprise: a group could have access permission to an enterprise and all its children (ALLOWED_INHERIT: AI), only to the father (ALLOW_DO_NOT_INHERIT: ADNI) or to none of them (DENIED: D). A fragment of this example is shown in Fig. 3.
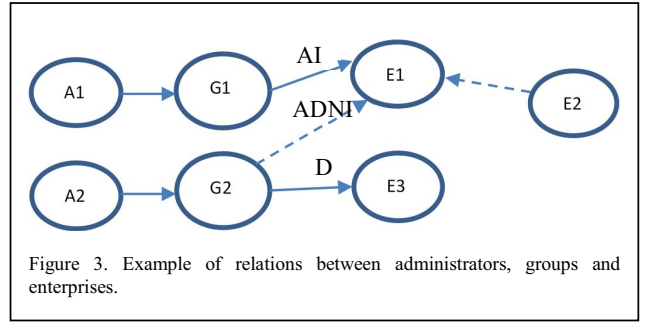

Figure 3. Example of relations between administrators, groups and enterprises.

During system operation, the administrator of the system may introduce changes in the relationships between the different entities. For instance, the administrator A1 could change the relationship between G1 and E1 from 'AI' to 'D'. In this example, the test derived from the model reveals a defect in the Riak implementation of this CRUD operation.

B. Generation of tests

The first step is to select the paths of the model that have to be covered by the tests. Looking at Fig. 2 (Section IV, part B), we can see that each path which starts at the initial point "Start CRUD operation" and ends at the point "End CRUD operation" will be considered as a use case to be tested. The idea is that each path of the graph should be covered by at least one test. In our example, we focused on two paths: (1) the path which

represents an unsuccessful CRUD operation due to faulty nodes ("*CRUD Check Operation*" - "*CRUD Operation*" - "*Check External Context (EC)*" - "*Check Client Expectations*" - "*CRUD Rollback Operation*" - "*Terminate*"), and (2) the path without problems ("*CRUD Check Operation*" - "*CRUD Operation*" - "*Check External Context (EC)*" - "*Check Client Expectations*" - "*Terminate*").

After selecting the paths, the internal context of the system must be configured in order to tune a number of parameters in relation to data management. In Riak, there are two parameters which determine how CRUD operations will work : (1) the number of nodes where data must be replicated (N) to guarantee a specific level of consistency/availability, and (2) the level of success in read and write operations with the parameters r ($r \leq$ N) and w ($w \leq$ N), respectively. In our experiment, the datacenter was made of 5 nodes, where N was equal to 3. The influence of the aforementioned parameters was checked against the paths selected from the model.

The data model design for the entities, both enterprises ("**Enterprise**") and groups ("**Group**"), is shown in Fig. 4. An integrity restriction must exist between the "*key_enterprise*" in "**Enterprise**" and the "key_group_enterprise" in "**Group**". The level of consistency and availability was set in "**Enterprise**" ("*level_r_enterprise*", "*level_w_enterprise*") and in "**Group**" ("*level_r_group*",   "*level_w_group*").   Moreover,   the relationship between groups and enterprises was established as a property ("*relationship_enterprise*") of the "**Group**".

Enterprise = bucket_type + bucket + key_enterprise
+ level_r_enterprise + level_w_enterprise
+ name_enterprise;

Group = bucket_type + bucket + key_group
+ level_r_group + level_w_group + name_group
+ key_group_enterprise
+ relationship_enterprise

Figure 4. Data model for enterprises and groups.

Finally, we tested the different CRUD operations over the selected paths (fail/no fail) and internal context configurations; values for the parameters r and w were ((1,1), (1,2), (2,1), (2,2)). This example shows the tests for the CRUD operation 'update' for a link. There are three kinds of link updates: ADNI to AI, AI to D and ADNI to D. The tests related to the update of the relationship between group G1, and enterprise E1, from AI to D are shown in Table II, which represents the results in eight test cases.

TABLE II.   TESTS RESULTS FOR THE UPDATE AI TO D OPERATION FOR A LINK

Tests	System conditions			System response	
		r	w	Expected	Actual
Case 1	No fail	1	1	( )	( )
Case 2	Fails	1	1	( )	(E1, E2)
Case 3	No fail	1	2	( )	( )
Case 4	Fails	1	2	( )	( )
Case 5	No fail	2	1	( )	( )
Case 6	Fails	2	1	( )	(E1, E2)
Case 7	No fail	2	2	( )	( )
Case 8	Fails	2	2	( )	( )

The first column represents the path, and the second and third column represent the internal context configurations. System response column "expected" displays the expected output (the set of enterprises with access from group G1): group G1 was able to access the enterprises E1 and E2 before the update operation, but after the operation G1 must not be allowed to access any of them. This is represented in the table as ( ).

The "**Actual**" column in Table II shows system behaviour when the test cases are executed against the current implementation. Cases 2 and 6 reveal a failure, because administrators belonging to group G1 are allowed to access E1 and E2. In fact they should not be allowed to access E1 and E2. This situation occurs when a node fails and $w=1$. This security problem has been caused because the new permission was not updated in time, in the second and the third node, when the first node failed ($w=1$). Due to this reason, a stale version was read from the second node (Case 2), or from the second and the third node (Case 6).

C. Fault Localization

The update operation, concerning the relationship between a group and an enterprise ("***updateRelationWithEnterprise***", ***updateMapWithContext***"), was implemented in Java as well as using the NoSQL key/value database API for Riak. The source code, shown in Fig. 5, indicates the failure, which was revealed by the tests in the previous subsection.

```
public void updateRelationWithEnterprise
                (String keyEnterprise, String relationship) {

   this.location = new Location(new Namespace(bucket_type,
                                    bucket), key);
   RegisterUpdate reg2 =
        new RegisterUpdate(BinaryValue.create(keyEnterprise));

   RegisterUpdate reg3 =
        new RegisterUpdate(BinaryValue.create(relationship));

   MapUpdate enterprise = new MapUpdate()
                        .update("keyEnterprise", reg2)
                        .update("relationship", reg3);

   MapUpdate mu = new MapUpdate()
                     .update("enterprise_info", enterprise);

   updateMapWithContext(mu);
}

private Context getMapContext() throws Exception {
    FetchMap fetch = new FetchMap.Builder(location).build();
    return client.execute(fetch).getContext();
}

private void updateMapWithContext(MapUpdate mu) {
   try{
       Context ctx = getMapContext();
       UpdateMap update =
            new UpdateMap.Builder(location, mu)
             .withOption(Option.W,new Quorum(level_w))
            .withContext(ctx)
            .build();
       client.execute(update);
   } catch (Exception e){
           System.err.println(e.getMessage());
}}
```

Figure 5. Faulty version of the Riak source code for updating a relationship.

The fault (defect) in the implementation of the CRUD update operation emerges due to the fact that the temporary failure of nodes has not been checked. When this event occurs, the CRUD operation should not proceed in its execution in order to assure that data have not changed. The program should either rollback the operation or notify (help) the client to make a decision between a transactional and a non-transactional CRUD operation. If the program does not take either of these actions, an access permission security could be broken when a node fails. A modified version of the function "**updateRelationWithEnterprise**" is shown in Fig. 6.

```
public void updateRelationWithEnterprise
                  (String keyEnterprise, String relationship) {
   this.location =
       new Location(new Namespace(bucket_type, bucket), key);

   RegisterUpdate reg2 =
       new RegisterUpdate(BinaryValue.create(keyEnterprise));
              …………

   updateMapWithContext(mu);
   //Check external context and check internal context

   if (!relationship.equals(getRelationshipEnterpriseGroup ())
                  && (level_w  != level_required)){
       //Client's expectations are unreachable
       rollbackCRUDoperation();
}}
```

Figure 6. Corrected Riak source code for preserving internal and external contexts.

The aim is to avoid these access permission failures when a node fails. This new function has been implemented according to the design of the general transactional model.

Therefore, the system response must be analysed. On the one hand the expected and the actual relationship must have the same external context:
"***relationship.equals(getRelationshipEnterpriseGroup()***".
On the other hand, the r and w parameters must be set to the desired level of consistency and availability – i.e. internal context: "*level_w == level_required*". According to the model, both internal and external context must satisfy client's expectations, otherwise a roll back operation must be committed.

VI. CONCLUSIONS AND FUTURE WORK

This paper has investigated NoSQL CRUD operations and the related issues of consistency and availability. We have developed a new model that takes into account the contextual information of clients (users) and the NoSQL database system. We have identified and classified various types of contexts related to the execution of CRUD operations, including internal and external contexts, and vital and non-vital contexts. The proposed model supports the execution of both transactional and non-transactional CRUD operations. We have also developed the testing criteria needed to test the CRUD operations and to identify their correctness. The proposed model is implemented using a widely-used Riak NoSQL database.

Based on the proposed model, various test cases have been developed. We take into account every path in the model (from the initial point "Start CRUD operation" to the end point "End CRUD operation") in order to provide the basis for developing test cases, and to check the CRUD operations correctness. A real-world example, of "authorization and access control" [25], was implemented in order to look into how CRUD operations correctness could be checked using tests derived from the general transactional model. Various experiments have been conducted. These show promising results and identify various faults in the execution of CRUD operations, which have otherwise not been detected when using standard Riak technology.

We believe that the proposed model would be of significant help to developers of NoSQL key/value databases (such as Riak) in order to test CRUD operations in a transactional or non-transactional mode. The future work will investigate different versions of data and concurrent requests, as part of the corresponding key/value NoSQL database.

REFERENCES

[1] M. Chen*, S. Mao, and Y.* Liu, "Big data: A survey," *Mobile Networks and Applications,* vol. 19, pp. 171-209, 2014.

[2] J. S. Ward and A. Barker, "Undefined by data: a survey of big data definitions," *arXiv preprint arXiv:1309.5821,* 2013.

[3] M. A. Mohamed*, O. G. Altrafi, and M. O Ismail*, "Relational vs. NoSQL Databases: A Survey," *International Journal of Computer and Information Technology (ISSN: 2279–0764) Volume,* 2014.

[4] G. DeCandia*, et al.*, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.,* vol. 41, pp. 205-220, 2007.

[5] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.,* vol. 12, pp. 463-492, 1990.

[6] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Comput.,* vol. 28, pp. 690-691, 1979.

[7] W. Lloyd*, M. J. Freedman, M. Kaminsky, and D. G. Andersen*, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," presented at the Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais, Portugal, 2011.

[8] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing,* vol. 9, pp. 37-49, 1995/03/01 1995.

[9] R. J. Lipton and J. S. Sandberg, *PRAM: A Scalable Shared Memory*: Princeton University, Department of Computer Science, 1988.

[10] B. F. Cooper*, et al.*, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.,* vol. 1, pp. 1277-1288, 2008.

[11] W. Vogels, "Eventually consistent," *Communications of the ACM,* vol. 52, p. 5, 2009.

[12] J. C. Corbett*, et al.*, "Spanner: Google’s Globally Distributed Database," *ACM Trans. Comput. Syst.,* vol. 31, pp. 1-22, 2013.

[13] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *USENIX ATC'12*, Boston, MA, 2012

[14] R. Escriba*, B. Wong, and E. Gün Sirer*, "Warp: Multi-Key Transactions for Key-Value Stores," 2013.

[15] J. Baker*, et al.*, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," in *Innovative Data system Research (CIDR)*, Asilomar, California, 2011, pp. 223-234.

[16] S. Das*, D. Agrawal, and A. El Abbadi, "G-Store: a scalable data store for transactional multi key access in the cloud," presented at the Proceedings of the 1st ACM symposium on Cloud computing, Indianapolis, Indiana, USA, 2010.

[17] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao, "Deuteronomy: Transaction Support for Cloud Data," in *CIDR*, 2011, pp. 123-133.

[18] Z. Wei, P. Guillaume, and C. Chi-Hung, "CloudTPS: Scalable transactions for Web applications in the cloud," *Services Computing, IEEE Transactions on,* vol. 5, pp. 525-539, 2012.

[19] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications," in *OSDI*, 2010, pp. 1-15.

[20] F. Junqueira*, B. Reed, and M. Yabandeh, "Lock-free transactional support for large-scale storage systems," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, 2011, pp. 176-181.

[21] A. Tripathi and B. Thirunavukarasu, "Design and Evaluation of a Transaction Model with Multiple Consistency Levels for Replicated Data," 2015.

[22] Y. Sovran*, R. Power, M. K. Aguilera, and L. Jinyang, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 385-400.

[23] J. Scott and G. Marshall, Eds., *A dictionary of Sociology.* Oxford: Oxford University Press, 2009.

[24] M. Younas and F. S. K. Most, "A new model for context-aware transactions in mobile services," *Personal and Ubiquitous Computing,* vol. 15, pp. 821-831, December 2011.

[25] I. Robinson*, J. Webber, and E. Eifrem, *Graph databases*, 2013.