TÜBİTAK

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2018) 26: 677 – 692
© TÜBİTAK
doi:10.3906/elk-1701-135

# Automated testing for distributed databases with fuzzy fragment reallocation

**Akhan AKBULUT**[1,*], **Gözde KARATAŞ**[2]
[1]Department of Computer Engineering, Faculty of Engineering, İstanbul Kültür University, İstanbul, Turkey
[2]Department of Mathematics and Computer Science, Faculty of Science and Letters, İstanbul Kültür University, İstanbul, Turkey

**Received:** 14.01.2017 • **Accepted/Published Online:** 04.01.2018 • **Final Version:** 30.03.2018

**Abstract:** As the Internet and big data become more widespread, relational database management systems (RDBMSs) become increasingly inadequate for web applications. To provide what RDBMSs cannot, during the past 15 years distributed database systems (DDBSs) have thus emerged. However, given the complicated structure of these systems, the methods used to validate the efficiency of databases, all towards ensuring quality and security, have become diversified. In response, this paper demonstrates a system for performing automated testing with DDBSs, given that testing is significant in software verification and that accredited systems are more productive in business environments. The proposed system applies several tests to MongoDB-based NoSQL databases to evaluate their instantaneous conditions, such as average query response times and fragment utilizations, and, if necessary, suggest improvements for indexes and fragment deployment. Within this context, autogenerated data, replica, meta, system, fragment, and index tests are applied. Clearly, the system's most important feature is its fuzzy logic-enabled fragment reallocation module, which allows the creation and application of reallocation strategies that account for data changes in query frequency.

**Key words:** Distributed database testing, NoSQL testing, empirical software validation, fragment reallocation, fuzzy logic, MongoDB

## 1. Introduction

With the development of cloud computing and the Internet, to make storing and managing big data possible and to optimize processing performance, new database systems—namely, distributed database systems (DDBSs)—have become preferable to relational database management systems (RDBMSs). As computer and Internet use continues to expand worldwide, nonstructural and scattered data continually emerge on a large scale, and recent reports have indicated that traditional relational database system (RDBS) techniques are consequently liable to be unable to store and process data. Research [1] points out that RDBSs are indeed designed to store and process structural and relational data in a centralized way. As an alternative, DDBSs have been created—most notably, NoSQL databases—and are growing in popularity [2] for certain technical reasons. NoSQL databases have developed solutions for:

- Reading and writing big data simultaneously;
- Storing ever-increasing data more accurately and efficiently;
- Allowing high scalability and access at a lower cost;

*Correspondence: a.akbulut@iku.edu.tr

- Overcoming the slow reading and writing speeds of RDBMSs;
- Expanding the limited capacity of RDBMSs [1,3–6].

As interest in NoSQL systems began to proliferate, so did the software developed with its systems. However, because these systems are still developing, proper testing and examinations have yet to be established.

The contribution of our paper is fourfold:

- We developed a web-based automated testing framework for NoSQL databases of MongoDB;
- We showed that database testing is an efficient way to improve QoS in NoSQL;
- We enhanced Abdalla's method [7] to reduce the number of fragments in a NoSQL deployment using fuzzy logic and we applied this extension in our approach to minimize hosting costs;
- We empirically analyzed the performance of our framework using the NoSQL version of the Northwind database and proved the benefit of this approach.

The rest of this paper is organized as follows. After Section 2 introduces a selection of NoSQL databases and MongoDB, Section 3 reviews related work. Next, Section 4 discusses our experimental setup and what we evaluate, after which Section 5 presents and discusses our experimental results. Lastly, Section 6 concludes our study with a summary.

## 2. NoSQL databases and MongoDB

To accommodate massive amounts of data without an SQL interface, designer and developer Carlo Strozzi introduced NoSQL in 1988 as an open source, nonrelational distributed database [8]. Unlike RDBMSs, NoSQL systems offer large data storage, data query, and easier ways to read and write different data, all of which when consistently combined make storing and processing data easier. In contrast to the vertical scaling performed by RDBMSs, NoSQL uses alternative data representation so that NoSQL can realize not only horizontal scaling but also the systematic dissemination of data known as sharding. Unlike RDBMSs, NoSQL databases do not have any tables, let alone connections between tables. Instead, NoSQL databases keep data in JSON, XML, or BSON format. Consequently, instead of wasting time by adding a new column for a new feature, as in RDBMSs, a new parameter can be formed in our JSON-formatted document [9–12].

In 2000, Eric Brewer introduced the CAP theorem, which holds that distributed databases cannot provide consistency, availability, and partition tolerance at the same time. In short, units of distributed systems are incapable of accessing the same data, of responding to all requests at the same time, and of retaining complete data in the case of unit loss. By extension, when NoSQL was reintroduced, the aim had become to improve performance by scaling out in light of these concepts [13–15].

In this study, tests were performed on MongoDB, a NoSQL database written in C++. MongoDB documents are similar to JSONObjects, yet stored as BSON. As such, MongoDB objects are not required to have the same structure or field name. Furthermore, common fields do not need to be of the same type. In that sense, MongoDB supports sharding, in which it partitions data collections and stores partitions across servers [1,16,17].

Today NoSQL is available in roughly 150 distributed database systems, such as MongoDB, Cassandra, HBase, CoucheBase, Berkeley DB, and Bigtable. With the development of the network system in recent years, however, issues have emerged such as big data management and big data storage. In response, new database systems have been developed, namely NoSQL databases.

## 3. Testing methods for NoSQL databases

The first testing frameworks and utilities in database testing were designed for use with RDBMSs. Working with good testing applications can allow users to reduce bugs in new and existing features, render tests suitable for documentation, improve designs, and activate refactoring. Among such tests, CUnit [18] is a system test for writing and running unit tests developed with the programming language C. Another testing framework, JUnit [19], focuses on unit testing developed with Java. Meanwhile, Cactus [20] is a test framework for unit testing server-side Java code, the cost of which it aims to reduce. By contrast, CppUnit [21] is a unit-testing framework developed with C++. Lastly, Google Test [22] is a unit test for C++ and based on xUnit.

Aside from general purpose testing frameworks, some research has focused on overcoming specific database problems. Wu et al. [23] examined how to create a synthetic database similar to the production database. Their approach is to form a general location model using various characteristics and to create synthetic data using the model. Marcozzi et al. [24] proposed a relational symbolic execution algorithm for testing Java methods, reading, and writing with SQL in RDBMS.

However, too few NoSQL testing frameworks are as readily available as RDBMSs. NoSQLUnit [25] is an open source testing framework for writing tests for Java that uses NoSQL. The aim of NoSQLUnit is to manage the lifecycle of NoSQL systems, for which it maintains two sets of rules and annotations. Buffalo Software [26] is another testing framework for NoSQL that uses Java. Younas et al. [27] and Truica et al. [28] proposed a model for testing create, read, update, and delete (CRUD) operations and related issues of consistency and availability in a NoSQL database.

## 4. Proposed framework

Our proposed framework was developed to facilitate software testing on distributed databases. Accordingly, MongoDB was chosen as the default platform for evaluating tests, and the C# programming language was used to develop the testing modules and interfaces. Our proposed framework consists of seven modules, each with a different purpose yet responsible for examining specific tests designed to evaluate the current conditions of databases. In total, 12 classes were developed to perform these tests, as shown in Figure 1. To demonstrate the results of every test that was performed in the system, two classes were developed, namely *TestError* and *TestFailure*, which are responsible for managing and logging operational outcomes. Only three testing classes (*TestFragment*, *TestReplica*, and *TestIndex*) can suggest critical improvements using their delegated events (respectively *ReallocateFragment*, *RepairMissingData*, *CreateIndex*, and *RebuildIndex*), which are triggered automatically under essential circumstances. Operational details are explained in the related testing parts. In order to begin the tests, the system's access information—that is, the connection details of the database—is required.

In effect, our proposed framework enables two options for users. First, the quick test applies all tests simultaneously and displays the results on the screen. In this test, everything about the system is determined by the developers of the tests, whereas the database owner sees only the outcome of the transactions. All functionalities of the proposed framework are depicted in Figure 2. The rest of this section explains the main functionalities of these tests and the suggestions for improvements that are offered by the system.

### 4.1. Structural analysis

This functionality is used to display catalogues, collections, and other database details for the tester. From the welcome screen of the system, the user can start any type of testing mentioned in the previous section. The

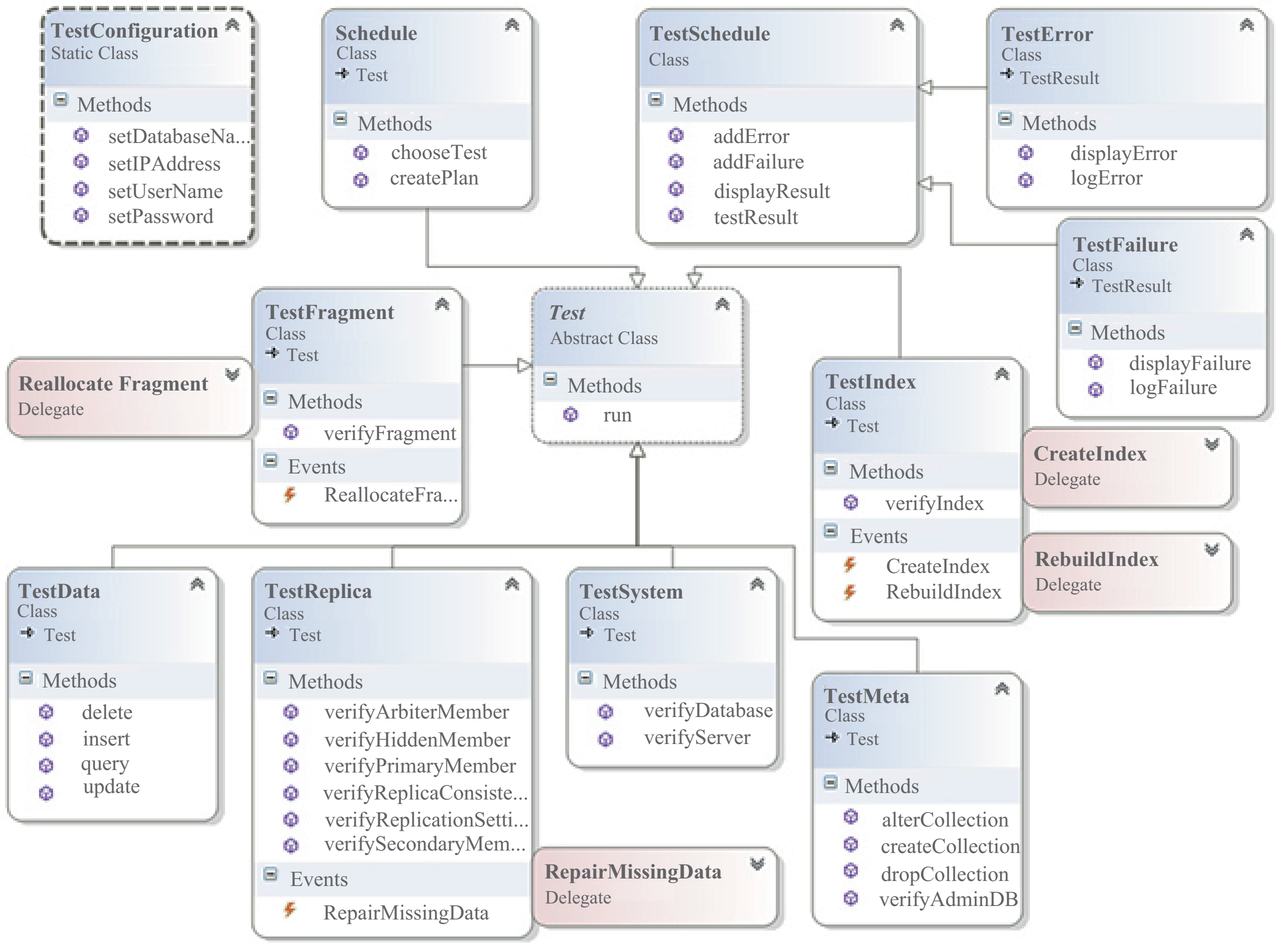


**Figure 1**. Proposed frameworks class diagram.

aim of showing this structural overview is to verify that the database structure exhibits an appropriate form and is to reveal details concerning collections in terms of the test database. With this functionality, users can verify collection properties, key-value pairs, and data types. It also shows how long the test took.

## 4.2. Data testing

The most important features of a database are CRUD operations, including data query and the adding, updating, and deleting of any amount of data. All CRUD operations of selected databases are tested during this module. Every operation is realized with synthetic data produced by the framework according to the collection's data types, which allows us to run four different tests: select, insert, delete, and update data operations. The main method of the tests is the same for all operations. The client determines which collection and how much data should be used. We used the NBuilder [29] test object generator to produce synthetic data to be used in CRUD operations as input parameters. All response times of these operations are measured and then displayed to the tester. This module act as a stress test. These tests demonstrate how successful and rapid basic operations in the database are. The main difference between relational database queries from MongoDBs is that it uses JavaScript-looking queries instead of traditional SQL queries.

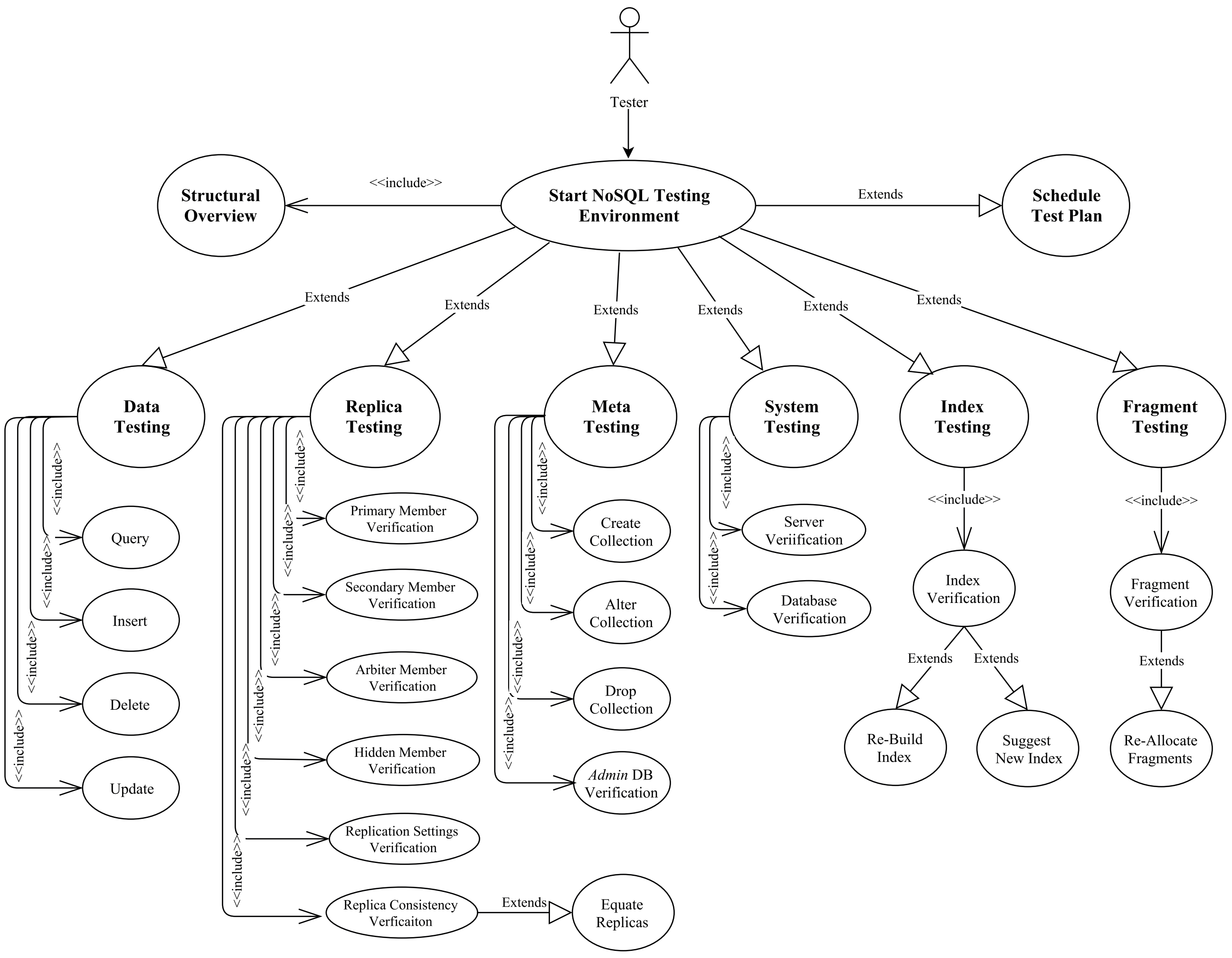


**Figure 2**. Testing scenarios' use case diagram.

## 4.3. Replica testing

After applying the integrity tests for replication-made system information, our proposed framework starts a test to ensure that strict replication consistency is provided in the forest. In order to maintain up-to-date copies of the shared dataset, all replica servers should house the exact same data. The *VerifyReplicaConsistency* method is used to compare the replica datasets within the test subject. If it discovers a sync problem, the *RepairMissingData* method is fired automatically to fix corrupted or missing data. At the end of synchronization therapy, a result output is produced including the number of altered records.

## 4.4. Metatesting

Metatesting involves several activities that evaluate the create, alter, and drop operations' response times of a whole new collection. In these tests, randomly selected data types will form a collection to be included in the test database. If these tests fail, presumably it means something is wrong with the metadata of the test subject. To verify the system preferences, another test called *verifyAdminDB* is fired to evaluate the *Admin* database (which stores metadata info such as logins, privileges, etc.) to be in covetable form. Problems related to the *Admin* database quite often can be solved by reinstalling the current instance.

### 4.5. System testing

This module includes tests for inspecting database connectivity and server communication, in two parts. The *verifyServer* method tests the connection status of the database server and shares results with the user, and then the *verifyDatabase* method tests the connection status of the database and shares results with the user. To troubleshoot connection problems, horizontal scaling is recommended in NoSQL platforms for dealing with heavy loads.

### 4.6. Index testing

Indexing is an important feature for MongoDB, by which every operation can be executed efficiently. This test finds the index count, as well as identifying useless indexes, in a chosen collection as a means of finding key information used often but not indexed in a chosen collection. In doing so, it suggests adding key information to the indexes. To perform this test, the *System.Profile* collection must be activated in the database before operations. This collection records all queries that are operated in the active instance. By inspecting that database's query frequencies, the most-queried fields can be designated as suggested places to build new indexes in.

Additionally, existing indexes are also inspected by using MongoDB's \*$indexStats* feature to get usage statistics. Long-term usage of NoSQL collections requires maintenance of indexes. Because operating performance of the index decreases over time with the multiple insert and delete operations on records, our system's *RebuildIndex* method provides an opportunity to test databases under conditions similar to their first deployment in terms of query performance.

### 4.7. Fragment testing

In the reallocation phase, the particular database must be separated into three parts, which are vertical, horizontal, or mixed for the fragmentation. Because DDBSs must have both high performance and availability, the reallocation system must decrease the cost of communications after allocation. Namely, during execution for queries, the total load of data should be at a minimum. The fragment reallocation module of our system recommends that the most effective application is the data fragment reallocation by considering the cost of updating and communication to every fragment. Fragment testing inspects the current form of fragments and offers a new deployment if necessary. This is the most important feature of our proposed framework, where optimizing fragment deployment is directly related to hosting and bandwidth costs.

For the technique used in fragment reallocation modules, several approaches have been investigated: the load-aware fragment allocation strategy [30], the region-based fragment reallocation for nonreplicated distributed databases [31], the fragment reallocation strategy based on hypergraphs, the fragmented–partitioned database system [32], the complete replication database system [33], and the partial replication database system. For this module, we selected Abdalla's method [4,7], considering its computational performance, to apply to fragment reallocation. That proposed method is an effective solution to vertical fragmentation and allocation of data storage in distributed environments. The number of database queries and their frequencies is used to determine in which servers to place DDMS fragments and consequently communication costs are reduced. The computation method is much faster than the alternatives because the method used is basically based on matrix operations. However, using this technique allows a server to host only one fragment, so there needs to be an extension to allow multiple fragments on a single server. We used fuzzy logic [24] to determine the ideal number of fragments to be deployed in the scenario. Hereafter, the original form of Abdalla's method will be detailed and our extension is explained in Subsection 4.7.1.

The new reallocation model for a distributed NoSQL database supposes the existence of a fully connected network formed from multiple sites S = {S1, S2, ..., Si} called Si. CCij is the communication cost, a value of the data transfer costs between two sites. Meanwhile, queries (Qi) represent the most operated queries Q = {Q1, Q2, ..., Qj} for each site.

All queries can be accessed and operated from each site by using a specific frequency, for which the query frequency matrix (QFij) represents the frequency for k queries on m sites. Each site has fragments (Fi) or partitions in the fragmentation phase. During allocation, both the appropriate allocation technique for queries and the replication number are determined for each fragment.

To render allocation effective, minimizing cost and maximizing performance should be considered [4]. The minimal cost is the sum of the storage cost, the updating cost, the communication cost between data, and the query cost for each fragment. In addition, the minimum reaction time and maximum productivity of the system are required for high system performance.

In our reallocation strategy, the first aim is to obtain the minimal cost so that optimization can be achieved. The database system that will be used, as well as fragments, queries, sites, and connection networks, must be analyzed ahead of time in order to identify the communication cost formula [4]. Assume that the length of fragments is L, the importance of fragments is Card(Fi), the number of attributes is h, the Ath attribute is a, and the record number of fragments is n, as in the following:

$$\text{Length}\,(Fi) = \sum La,\ \ 1 \leq a \leq h. \tag{1}$$

$$Z\,(Fi) = Card\,(Fi) \times Length\,(fi)\,,\ \ i = 1, \ldots, n \tag{2}$$

$$\text{Fragment size}:Z\,(Fi) = n \times \sum Lj \tag{3}$$

The Xij matrix needs to be identified to appoint fragments of each site by using the initial allocation matrix and retrieval matrix (RM):

$$\text{Xij} = \begin{cases} 1, & \text{if Fi is assigned to S} \\ 0, & \text{otherwise} \end{cases}$$

The RM and update matrix (UM) are identified for queries. The RM keeps retrieval data, whereas the UM keeps updated data; Rkj is the frequency of accessing fragments (Fj) for query k in the RM, while Ukj yields the Fj for query k in the UM:

$$\text{RM(Rkj)} = \begin{cases} 1, & \text{if Qk retrieve Fj} \\ 0, & \text{otherwise} \end{cases}$$

$$\text{UM(Ukj)} = \begin{cases} 1, & \text{if Qk update Fj} \\ 0, & \text{otherwise} \end{cases}$$

The number of queries was observed in integer format by using the *System.Profile* collection of the MongoDB that counts specific queries at the times of queries mentioned in the system database. *System.Profile (<database>.system.profile)* is the default collection of MongoDB that calculates query numbers. To activate the database, the *db.SetProfilingLevel* function is used. Identified for queries, the UM calculates updated data on MongoDB and considers the *System.Profile* collection to be the RM. This collection must be activated before queries such as remove, insert, update, and command.

To activate the database, the *db.SetProfilingLevel* function is used. If the parameter of the function is 0, then activation is closed, meaning that there is no profiling. If the parameter of the function is 1, then activation is open.

Identified for queries, the UM calculates updated data for the MongoDB and considers the *System.Profile* collection to be the RM. Unlike the RM, the UM keeps the number of queries for the operations of removing, inserting, updating, commanding, and querying. Each query has a frequency value at the site where the query operates. In this context, the query frequency matrix (QFM) maintains the query frequency executed in the site.

The system poses some constraints, including with capacity (C), or the maximum size and limit for fragments (FL)—that is, the maximum fragment number for processing at a site. What follows are some formulae for the limits used to manage the initial allocation and reallocation phases:

$$\sum_{i=1}^{m} Xij >= 1, \quad 1 \leq i \leq n. \tag{4}$$

$$\sum_{i=1}^{n} Qij \times size\,(Fi) \leq Ci, \;\; 1 \leq j \leq m. \tag{5}$$

$$\sum_{i=1}^{m} Qij \leq FLi, \; 1 \leq i \leq n. \tag{6}$$

$$\sum_{i=1}^{m} Qij = 1, \; 1 \leq i \leq n. \tag{7}$$

In Eq. (4) every fragment (F) has to be dedicated to one or more sites; in Eq. (5) each site will not have more than the capacity (C) of the site; in Eq. (6) each site will not have a value greater than the (FL) fragment's number; and in Eq. (7), every fragment (F) has to be dedicated on site(s) determined by fuzzy logic during reallocation. All notations are listed in Table 1.

We compared all servers and identified the minimal distance between sites by applying the minimum algorithm because the communication cost value between servers did not represent the most efficient method. However, it is possible to connect a server to a target server cheaply and for quicker results, since the network band varies worldwide. In any case, the distance cost matrix (DM) can be determined by considering the communication cost matrix.

In that sense, this reallocation strategy protects the existence of information about the estimated frequency value of updating transactions in the distribution of fragments between sites. At the initial allocation, every fragment (F) has to be allocated to one or more sites in the distributed database system. Since distribution includes optimal fragment allocation between sites, the most appropriate allocation technique will include minimum total updating cost within the network without violating the restrictions of sites for fragments of various sizes and executed query numbers for network sites.

Queries published on multiple sites will be accepted, though they differ and have different values of frequency. However, to achieve the most appropriate dynamic reallocation, the updated query values for the frequency of each distributed fragment should be used, chiefly to take advantage of mean values. To execute the

**Table 1**. Notations of the algorithm.

| Notation | Meaning |
|---|---|
| S | Set of sites |
| QS | Queries of every site |
| FUPM | Fragment update pay matrix |
| CUFT | Cumulative update frequency matrix |
| $QF_{ij}$ | Access frequency of the ith query at site j |
| FUFM | Fragment update frequency matrix |
| DM | Distance cost matrix |
| $Uk_j$ | Frequency of accessing fragments for query k in the UM |
| $CC_{ij}$ | Communication cost between site i and j |
| UM | Update matrix |
| RM | Retrieval matrix |
| $X_{ij}$ | Assignment of fragments to sites |
| QF | The query frequency matrix |
| L | Length of fragments |
| $FL_i$ | Maximum number of fragments for site j |
| $Rk_j$ | Frequency of accessing fragments for query k in the RM |
| $C_j$ | Capacity of site j |
| C | Set of capacities |
| $Q_j$ | The jth query |
| Q | Set of queries |
| $Z(f_j)$ | Fragment size |
| F | Set of fragments |
| $S_j$ | The jth site |
| $F_j$ | The ith data fragment |
| QFM | Query frequency matrix |

initial fragment allocation phase, which can include duplicated fragments, the reallocation strategy supposes that fragments will be allocated between sites, all by considering the RM and QFM.

Our study presents a new data reallocation strategy for DDBSs to both distribute and reallocate initial fragments that are not redundant. Our strategy is applied by considering the updated matrix in terms of the frequency of queries since updating queries poses a higher cost than requests retrieved. In that way, the values of the UM and QFM are used to obtain the fragment update frequency matrix (FUFM). For reallocation, the FUFM is therefore the base and can be determined in terms of the updated query value, as long as the fragments are for a specific site.

We used FUFM values to obtain the cumulative update frequency matrix (CUFT), which is multiplied by the DM and yields the fragment update pay matrix (FUPM). As a result, the value of the maximum updating cost can be defined for each fragment, as well as the voluntary site with the maximum cost for storing the selected fragment. However, when deciding upon the voluntary site, the limitations of the site should be considered, for if a limit violation exists, then the fragment is stored on the next site, which inevitably has only the second greatest maximum updating cost.

For a particular fragment, if multiple sites have the same value for the updating cost, then the fragment priority (FP) method is used for a particular site. Thanks to the FP method, fragments can be allocated to a site by using the maximum FP value. After obtaining the maximum FP value, the fragment is definitely the only copy in the network.

The FP method can also be used to identify the access the number of a particular fragment, or how many times the query was called. For the same particular fragment (F), if multiple sites have the same value for updating the cost, then the FP method is used. Therefore, to prevent fragment repetition through sites, the FP method is used to determine Fi allocated on the site with the maximum fragment priority value. The steps of this new reallocation strategy are shown in Figure 3.

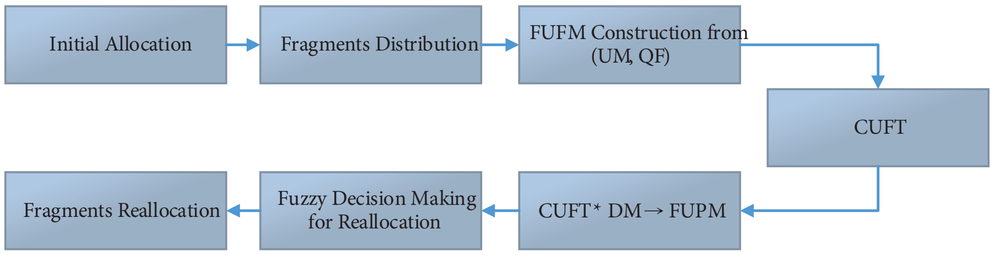


**Figure 3**. Phases of reallocation.

The FP method can be used to identify the access number of a particular fragment:

$$FP\,(Sj, Fi) = \sum_{i=1}^{m} (QFhi \times QShi) \times DMj, h, \;\; 1 \leq j \leq m. \tag{8}$$

$$QShi = \sum_{i=1}^{n} RMhi + \sum_{i;=1}^{n} UMhi, \;\; 1 \leq h \leq h. \tag{9}$$

In Eq. (8), calculating a fragment's priority value prevents fragment repetition across sites; in Eq. (9) queries are given cumulative access numbers.

Queries accrue cost value for access and updating processing at selected sites. Thus, by considering the FUPM and some cost methods, the FUPM matrix can be obtained:

$$CUFT = \sum_{i=1}^{n} \sum_{i=1}^{m} (QFi \times UF\,(Fi)). \tag{10}$$

$$DMij = Min\,(CCij)\,, \;\; 1 \leq i,\, j \leq m \tag{11}$$

$$FUPM = \sum_{i=1}^{m} CUFT\,(Fi) \times DMjj^{'}, \;\; 1 \leq j, \;\; j^{'} \leq m. \tag{12}$$

$$\sum_{i=1}^{m} Max\,(FUPMij)\,, \;\; 1 \leq i \leq n. \tag{13}$$

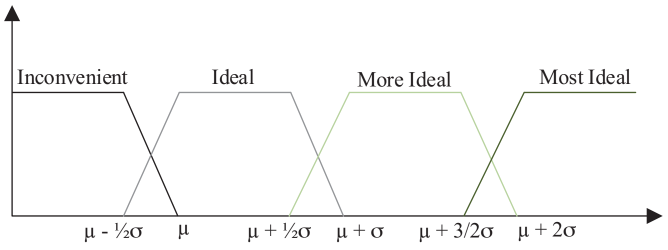
The CUFT matrix is obtained by using Eqs. (10) and (11). To find the CCM matrix, the minimum algorithm in Eq. (12) is used. The FUPM matrix is obtained by using Eqs. (12) and (13). The result is the fuzzy system used to determine reallocation.

Generally, when queries have the minimum total cost for updating the process in the allocation phase, the distribution is arguably appropriate. In that sense, the fuzzy system works to decide the allocation of fragments among sites.

### 4.7.1. Fuzzy reallocation

Fuzzy reallocation is a decision-making process for selecting appropriate sites for deploying fragments by evaluating their values in the FUPM matrix. Abdalla's [4,7] approach deploys only the fragment into the alpha site with the maximum cost. For scenarios with multiple sites with a near-maximum cost, this technique does not satisfy the runner servers. To overcome this setback and to select the optimal number of servers to deploy fragments, we used a fuzzy mechanism.

To select the number of servers in a normally distributed list, we used the fuzzy set values of the fuzzy variables F, described as {IN, ID, MRI, MSI} belonging to Inconvenient, Ideal, More Ideal, and Most Ideal levels, all shown in Figure 4a. Population intensities are labeled with the number of servers exceeding the mean over the population as Rare, Normal, and Intense, as shown in Figure 4b. We then specified the fuzzy rule base. We can write fuzzy associations as antecedent–consequent pairs of "if, then" statements, as shown in Table 2. Lastly, we determine the output action given the input conditions.



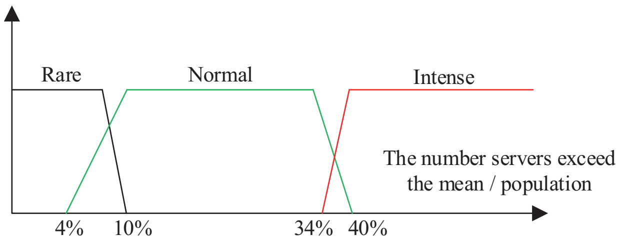


**Figure 4**. a) Fuzzy membership function for cost distribution over population. b) Fuzzy membership of workload intensity for servers.

**Table 2**. Fuzzy associative memory matrices for fragment reallocation decisions.

| | Rare | Normal | Intense |
|---|---|---|---|
| IN | Deploy | Deploy | Deploy |
| ID | Deploy | Deploy | Abort |
| MRI | Deploy | Abort | Abort |
| MSI | Abort | Abort | Abort |

### 4.7.2. Reallocation process

To create the initial allocation matrix, we considered initial client locations. First, to obtain server IP addresses, we used web services (e.g., icanhazip.com), and to convert IP addresses for geolocation and obtain the longitude and latitude of a user's location, we used freegeoip.net. To identify the nearest server, we used the *GeoCoordinate* class of the *GoogleAPI* Library. Each server identified different places, whose distance to the user's location was calculated by way of the distance method. The values of UM and QFM were used to obtain FUFM values, which were in turn used to obtain CUFT values. After implementing the CUFT matrix, the CUFT and DM were multiplied to obtain the FUPM. Eventually, processing fragment reallocation was realized via the FUPM matrix, and each fragment was on a site determined by fuzzy association.

## 5. Results and evaluation

To examine the results of the tests developed for the system, all results were verified on a sample database and all recommendations taken into account. The NoSQL version of the Northwind database, familiar from Microsoft's SQL sample database, was used for testing the database. We remodeled this well-known relational database into a document-based NoSQL database form, and in our testing scenario we used it on three different scales: versions with 1000, 10,000, and 1,000,000 records. Each database version had 13 collections and 29 fragments over 4 servers.

After tests were performed and both results and recommendations were acquired, new indexes were added in light of the system's recommendations and, as applicable, the index test results, and then the tests were reperformed. Performance-related timing results from before and after indexing operations appear in Table 3. Measurements taken after modifications prove that applying system suggestions improves system performance. As Table 3 shows, with the new indexing operations, performance improved and execution times diminished. With the utilization of fragment testing, the system detects unnecessary fragments in the deployment and fragments are reorganized as needed. The outcome of this process is a stepdown in hosting and bandwidth costs.

Figure 5 shows the comparison of fragments according to our method and Abdalla's method. While Abdalla's method only assigns one fragment to a single server, our method assigns more than one as much as necessary. The solution of a single fragment may be acceptable for very rare cases. This is why our solution guarantees the minimum number of fragments while reducing the number of deployments on the servers.

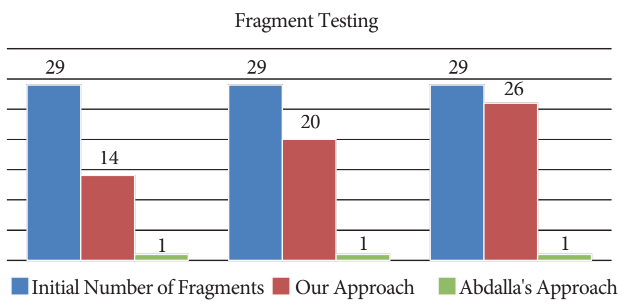


**Figure 5**. Fragment testing.

The necessary indexes were created by analyzing the queries. Depending on the queries, the number of indexes in the database with 1000 tuples has been increased from 13 to 15, as shown in Figure 6.

In Figure 7, we depicted the results of the data tests. The performance of the query, insert, update, and delete operations on 3 differently sized databases was analyzed. The rebuilding and restructuring of the indexes had a direct impact on query performance.

**Table 3**. Test results.

| Database size (total number of tuples) | | 1000 | | 10,000 | | 1,000,000 | |
|---|---|---|---|---|---|---|---|
| Average test results of both initial and modified versions | | Result of initial | Result of modified | Result of initial | Result of modified | Result of initial | Result of modified |
| Data testing | Query | 1.5 ms | 1.30 ms | 12.3 ms | 11.48 ms | 5040 ms | 3762 ms |
| | Insert | 61 ms | 60 ms | 587 ms | 579 ms | 32,157 ms | 30,674 ms |
| | Update | 6.52 ms | 6.5 ms | 92.86 ms | 90.7 ms | 10,320 ms | 10,101 ms |
| | Delete | 22.63 ms | 20.6 ms | 439.45 ms | 422.63 ms | 51,713 ms | 48,502 ms |
| Replica testing | Primary member verification | 1200 ms | 981 ms | 1525 ms | 1060 ms | 1992 ms | 1092 ms |
| | Secondary member verification | Less than 100 ms | Less than 100 ms | 998 ms | Less than 100 ms | 8997 ms | Less than 100 ms |
| | Arbiter member verification | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms |
| | Hidden member verification | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms |
| | Replica setting verification | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms |
| | Replica consistency verification | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms | Less than 100 ms |
| | Equate replicas | 103 ms | 101 ms | 10301 ms | 10100 ms | 20112 ms | 19101 ms |
| Metatesting | | 4.77 ms | 4.03 ms | 52.13 ms | 14.77 ms | 6 s | 4 s |
| System testing | Server verification | 1023 ms | 897 ms | 8069 ms | 7128 ms | 10,027 ms | 9928 ms |
| | Database verification | 102 ms | 92 ms | 1028 ms | 963 ms | 15,959 ms | 9633 ms |
| Index testing | Number of index | 13 | 15 | 13 | 19 | 13 | 27 |
| Fragment testing | Number of fragments / servers | 29 / 4 | 14 / 2 | 29 / 4 | 20 / 4 | 29 / 4 | 26 / 4 |

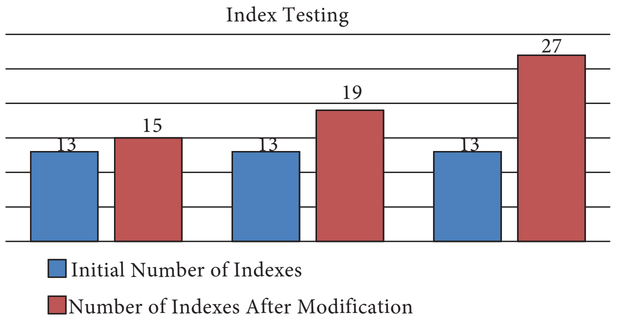


**Figure 6**. Index testing.

## 6. Conclusion

This study presented a web-based framework designed to perform database tests on NoSQL platforms, by which structural, deployment, and index faults of MongoDB-based NoSQL databases can be corrected. Test results showed that significant evaluation can be gained with the proposed framework, since improving the fragment reallocation improves the performance among distributed systems, hence our focus on the reallocation module.

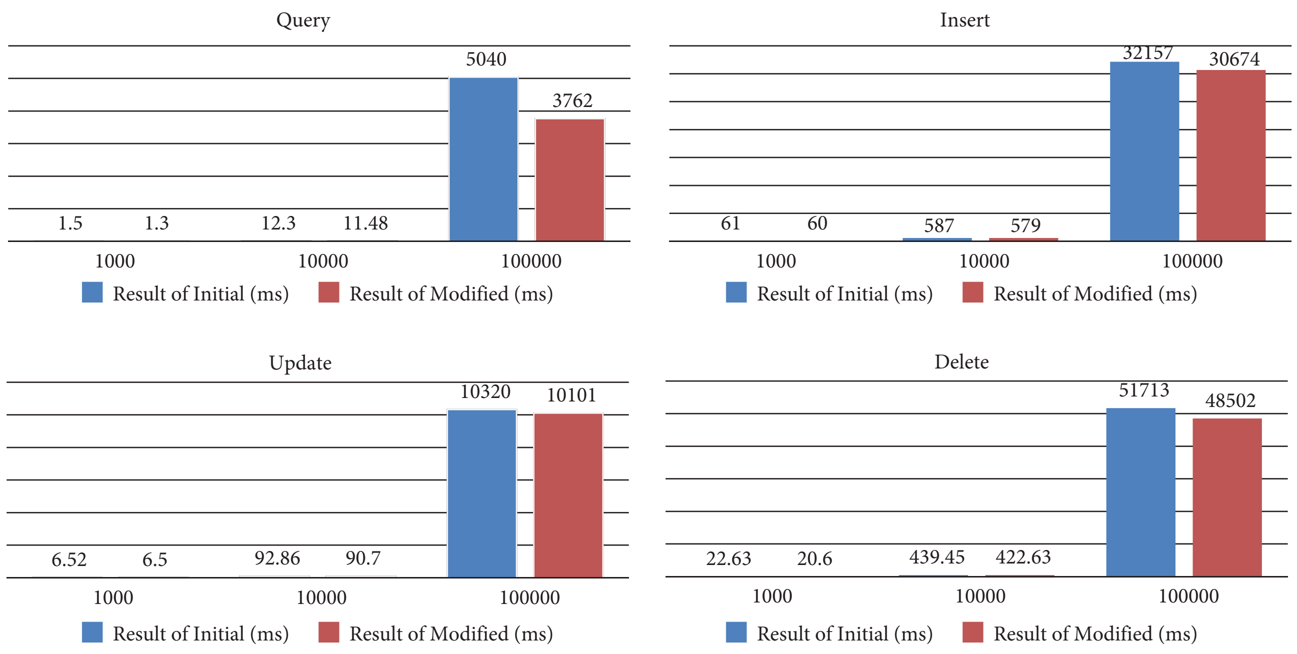


**Figure 7**. Data testing.

In that module, we optimized the well-known technique of Abdalla's approach with a fuzzy decision-making process, and our new technique allows users to deploy fragments among more than one site with necessary conditions.

As a continuation of this study, tests should continue to be developed for other NoSQL ecosystem members such as Couchbase, Cassandra, and HBase. Ultimately, more users should be able to use our tests and improve their database systems.

## References

[1] Grier DA. The relational database and the concept of the information system. Ann Hist Comput 2012; 34: 9-17.

[2] Moniruzzaman ABM, Hossain SA. NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison. arXiv: 1307.0191

[3] Li X, Ma Z, Chen H. A query-oriented data modeling approach for NoSQL DBs. In: Advanced Research and Technology in Industry Applications; 29–30 September 2014; Ottawa, Canada. New York, NY, USA: IEEE. pp. 338-345.

[4] Abdalla HI, Tounsi M, Marir F. Using a greedy-based approach for solving data allocation problem in a distributed environment. In: International Conference on Parallel and Distributed Processing Techniques and Applications; 14–17 July 2008; Las Vegas, NV, USA. pp. 975-980.

[5] Bhogal J, Choksi I. Handling big data using NoSQL. In: Advanced Information Networking and Applications Workshops; 24–27 March 2015; Washington, DC, USA. New York, NY, USA: IEEE. pp. 393-398.

[6] Srivastava PP, Goyal S, Kumar A. Analysis of various NoSQL database. In: Green Computing and Internet of Things; 8–10 October 2015; Delhi, India. New York, NY, USA: IEEE. pp. 539-544.

[7] Abdalla HI. An efficient approach for data placement in distributed systems. In: 5th FTRA International Conference on Multimedia and Ubiquitous Engineering; 28–30 June 2011; Loutraki, Greece. New York, NY, USA: IEEE. pp. 297-301.

[8] Zahid A, Masood R, Shibli MA. Security of sharded NoSQL databases: a comparative analysis. In: Information Assurance and Cyber Security; 12–13 June 2014; Rawalpindi, Pakistan. New York, NY, USA: IEEE. pp. 1-8.

[9] Gilbert S, Lynch N. Perspectives on the CAP theorem. Computer 2012; 45: 30-36.

[10] Li Y, Manoharan S. A performance comparison of SQL and NoSQL databases. In: Communications, Computers and Signal Processing; 27–29 August 2013; Victoria, Canada. New York, NY, USA: IEEE. pp. 15-19.

[11] Benefico S, Gjeci E, Gomarasca RG, Lever E, Lombardo S, Ardagna D, Di Nitto E. Evaluation of the CAP properties on Amazon SimpleDB & Win Azure Table storage. In: 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing; 26–29 September 2012; Timisoara, Romania. New York, NY, USA: IEEE. pp. 430-435.

[12] Tudorica BG, Bucur C. A comparison between several NoSQL databases with comments and notes. In: Roedunet International Conference RoEduNet International Conference, 10th Edition: Networking in Education and Research; 23–25 June 2011; Iasi, Romania. New York, NY, USA: IEEE. pp. 1-5

[13] Brewer E. CAP 12 years later: How the rules have changed. Computer 2012; 45: 23-29.

[14] Stonebraker M. SQL databases v. NoSQL databases. Commun ACM 2010; 53: 10-11.

[15] Dragomir G, Ignat I. Distributed database environment testing. In: International Conference on Automation, Quality and Testing, Robotics, Vol. 2; 25–28 May 2006; Cluj-Napoca, Romania. pp. 90-95.

[16] Liu Y, Wang Y, Jin Y. Research on the improvement of MongoDB auto-sharding in cloud environment. In: 7th International Conference on Computer Science & Education; 14–17 July 2012; Melbourne, Australia. New York, NY, USA: IEEE. pp. 851-854.

[17] Chodorow K. MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2013.

[18] Khoroshilov AV, Rubanov VV, Shatokhin EA. Automated formal testing of C API using T2C framework. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation; 13–15 October 2008; Porto Sani, Greece. Berlin, Germany: Springer. pp. 56-70.

[19] Louridas P. JUnit: unit testing and coiling in tandem. IEEE Software 2005; 22: 12-15.

[20] Dudney B, Lehr J. Jakarta Pitfalls: Time-Saving Solutions for Struts, Ant, JUnit, and Cactus (Java Open Source Library). Hoboken, NJ, USA: John Wiley & Sons, 2013.

[21] Dickheiser M. Using CppUnit to implement unit testing. In: Dickheiser M, editor. Game Programming Gems 6 (Book & CD-ROM) (Game Development Series). Newton Centre, MA, USA: Charles River Media, Inc., 2006. pp. 45-49.

[22] Sen A. A quick introduction to the Google C++ testing framework. IBM DeveloperWorks 2010; 20: 1-10.

[23] Wu X, Sanghvi C, Wang Y, Zheng Y. Privacy aware data generation for testing database applications. In: 9th International Database Engineering and Application Symposium; 25–27 July 2005; Montreal, Canada. New York, NY, USA: IEEE. pp. 317-326.

[24] Marcozzi M, Vanhoof W, Hainaut JLA. Relational symbolic execution algorithm for constraint-based testing of database programs. In: 13th IEEE International Working Conference on Source Code Analysis and Manipulation; 22–23 September 2013; Eindhoven, the Netherlands. New York, NY, USA: IEEE. pp. 179-188.

[25] Ammann P, Offutt J. Introduction to Software Testing. Cambridge, UK: Cambridge University Press, 2016.

[26] Ibrahim H, Alwan AA, Udzir NI. Checking integrity constraints with various types of integrity tests for distributed databases. In: Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies; 3–6 December 2007; Adelaide, Australia. pp. 1-2.

[27] Younas M, Tuya J. A new model for testing CRUD operations in a NoSQL database. In: 30th International Conference on Advanced Information Networking and Applications; 23–25 March 2016; Crans-Montana, Switzerland. New York, NY, USA: IEEE. pp. 79-86.

[28] Truica CO, Radulescu F, Boicea A, Bucur I. Performance evaluation for CRUD operations in asynchronously replicated document oriented database. In: 20th International Conference on Control Systems and Computer Science; 27–29 May 2015; Bucharest, Romania. New York, NY, USA: IEEE. pp. 191-196.

[29] Siemer A, Kimber R. ASP. Net Mvc 2 Cookbook. Birmingham, UK: Packt Publishing Ltd., 2011.

[30] Chen Z, Yang S, He L, Tan S, Zhang L, Yang H, Zhang G. Load-aware fragment allocation strategy for NoSQL database. In: Proceedings of International Conference on Internet Multimedia Computing and Service; 10–12 July 2014; New York, NY, USA. New York, NY, USA: ACM. p. 416.

[31] Varghese PP, Gulyani T. Region-based fragment allocation in non-replicated distributed database system. International Journal on Advanced Computer Theory and Engineering 2012; 1: 62-70.

[32] Chen Z, Yang S, Tan S, He L, Yin H, Zhang G. A new fragment re-allocation strategy for NoSQL database systems. Front Comput Sci-Chi 2015; 9: 111-127.

[33] Abdalla HI. A new data re-allocation model for distributed database systems. Comm Com Inf Sc 2012; 5: 45-60.

[34] Ross TJ. Fuzzy Logic with Engineering Applications. Hoboken, NJ, USA: John Wiley & Sons, 2009.