Journal of King Saud University – Computer and Information Sciences (2016) **28**, 133–145

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

A middle layer solution to support ACID properties for NoSQL databases

Ayman E. Lotfy [a,*], **Ahmed I. Saleh** [b], **Haitham A. El-Ghareeb** [c], **Hesham A. Ali** [d]

[a] *Information Technology Institute, Egypt*
[b] *Computer Engineering and Systems Dept., Faculty of Engineering, Mansoura University, Egypt*
[c] *Information Systems Dept., Faculty of Computers and Information Sciences, Mansoura University, Egypt*
[d] *Computer Engineering and Systems Dept., Faculty of Engineering, Mansoura University, Egypt*

Received 27 January 2015; revised 10 May 2015; accepted 26 May 2015
Available online 3 November 2015

KEYWORDS

NoSQL;
ACID;
Consistency;
BASE;
Transactions;
Concurrency

Abstract The main objective of this paper is to keep the strengths of RDBMSs as consistency and ACID properties and at the same time providing the benefits that inspired the NoSQL movement through a middle layer. The proposed middle layer uses a four phase commit protocol to ensure: the use of recent data, the use of the Pessimistic technique to forbid others dealing with data while it is used and the data updates residing in many locations to avoid the loss of data and disappointment. This mechanism is required, especially in distributed database application NoSQL based environment, because allowing conflicting transactions to continue not only wastes constrained computing power and decreases bandwidth, but also exacerbates conflicts. The middle layer keeps tracking all running transactions and manages with other layers the execution of concurrent transactions. This solution will help increase both of the scalability, and throughput. Finally, the experimental results show that the throughput of the system improves on increasing the number of middle layers in scenarios and the amount of updates to read in a transaction increases. Also the data are consistent with executing many transactions related to each other through updating the same data. The scalability and availability of the system is not affected while ensuring strict consistency.

1. Introduction

Relational Database Management Systems (RDBMs) are used to store and manage data. RDBMSs maintain the relations between data through the use of constraints such as primary and foreign keys. In case the size of data is not big, the users of the system are not many, and the requirement of data consistency is mandatory, RDBMs will be the perfect solution. As data are stored on one machine there is no challenge to manage relations between them. Some of today's web applications face a challenge of serving millions of users who are distributed

* Corresponding author.
E-mail addresses: Eng5ayman@gmail.com (A.E. Lotfy), aisaleh@yahoo.com (A.I. Saleh), helghareeb@mans.edu.eg (H.A. El-Ghareeb), K_hesham71@yahoo.com (H.A. Ali).
Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2015.05.003

all over the world and who expect the service to be always available, reliable, and with a high degree of consistency. With the increase in the number of users and the increase in the amount of generated data, data have to be stored in many servers which may be distributed over different locations. Distributing data over many servers makes it difficult to maintain relations between data. Web applications require the ability to scale in many servers (Orend, 2010). This challenge led to the appearance of a new trend of DBMS called NoSQL DBMS. NoSQL DBMSs have the ability to distribute data over many nodes providing the needed levels of availability, while keeping scalability within accepted levels, and ignoring data consistency.

The problem with distributed database systems is that we have to overcome one of the three basic properties which are: Availability, Consistency, and Partitioning according to the CAP theorem proved by professor Eric Brewer (Gilbert and Lynch, 2002). The CAP theorem states that any DBMS can provide only two of three properties: consistency (C) which means that DBMS supplies users with the same version of data at the same time, high Availability (A) system responds to users with data at any moment, and tolerance to network Partitions (P) data divided over many computers (Wei, 2012; Gilbert and Lynch, 2002).

This paper is organized as follows: Section 2 presents Background and Basic Concepts about Traditional RDBMS, ACID properties, NoSQL DBMS and BASE properties. Section 3 presents Previous Efforts. Section 4 describes the proposed frame work for improving ACID properties of NOSQL DBMS. Section 5 presents performance metrics. Section 6 presents experimental results and Section 7 is the conclusion.

2. Background and basic concepts

In this section, the basic concepts related to the traditional RDBMs, ACID properties as well as NoSQL database and BASE properties will be introduced. The challenges associated with RDBMS and NoSQL are also introduced.

2.1. Traditional RDBMS and ACID properties

A relational database is a system that stores data in a collection of relations. The data saved in relations have relations between each other through primary and foreign keys. Relations of typical applications are called tables. Examples of such application are Microsoft SQL Server, Oracle DBMS, and IBM DB2. SQL language is employed to manage data in the relational database system. The scope of SQL Language includes query data from multiple joints, insert, delete, update, and other operations. Because of their rich set of features such as query capabilities and transaction management, they seemed to be fit for almost every possible task. One of the important features of RDBMs is to provide ACID properties in order to execute transactions (Wei, 2012) as transactions are divided into sub-transactions. RDBMs maintain ACID properties, which are difficult to be maintained over distributed data.

RDBMs ensure ACID for every transaction handled such as concurrent transactions; moreover they ensure consistency and availability over scalability according to the CAP theorem (Gilbert and Lynch, 2002). On the other hand, RDBMs are not the most appropriate solution in some scenarios such as big data and large scale web applications (Milanović and Mijajlović, 2012; Valer, 2013). Big data require 3 Vs Volume, Velocity and Variety (Valer, 2013). Big data entail scalability and flexibility which are not provided by RDBMs (Valer, 2013). Particularly, it is difficult to make transactions and joint operations in a distributed system using RDBMSs. As the exponential growth of data the system has to scale horizontally this means to divide data over many machines. Horizontal scalability is difficult to obtain by RDBMs. RDBMs require fixed table structure, which is not required by big data.

2.2. NoSQL DBMS and BASE properties

NoSQL is an umbrella term that includes a group of non-relational DBMS. It means "Not Only SQL" (Orend, 2010). It provides good horizontal scalability for simple read/write database operations distributed over many servers, in contrast to traditional RDBMs that have little or no ability to scale horizontally (Cattell, 2010). NoSQL databases do not need a fixed table structure and does not provide a full ACID support. It provides eventually consistency, which means that data will be consistent over a period of time (Orend, 2010). Some of the usually used NoSQL databases are CouchDB, Riak, Cassandra, Mnesia, BerkeleyDB, HamsterDB, MongoDB, and Redis (Muhammad, 2011).

There have been various approaches to classify NoSQL databases, each with different categories and subcategories (Kriha, 2011). However, the basic classification that most would agree on is the one that is based on data models. We can classify NoSQL databases according to the data model to the column, document, *K*-value, and Graph (Kriha, 2011). NoSQL databases are based on nonfunctional categories and the evaluation of their feature coverage. Table 1 summarizes these features.

In the document oriented database each database consists of a number of collections (as table in RDBMs) each collection contains a number of documents written in the Binary Structured Object Notation (BSON) format Cattell, 2010. Each document has an ID by which we can access this document. Each document is composed of key–value pairs. We can get or update a specific key–value in a specific document in a specific collection. SimpleDB, mongoDB, and coachDB are some examples of document oriented NoSQL database. In key/value store data are stored on a hash (Cattell, 2010). The key is a unique identifier and value is the respective data. Data are structured similar to a dictionary. Insert, delete and update operations are applied on each given key. A number of key–value pairs can be grouped in buckets and the key has a part that identifies the bucket. Update and get operations are done on keys. Project Voldermort, Riak, Redis, Scalaris and Tokyo Cabinet are some examples of key–value NoSQL database (Franco and Nogueira., 2011). Column Store keeps data in rows. Each row has a unique identifier called key and one or more columns. Column is themselves key–value pairs. The column names need not be predefined so the structure is not fixed. Columns in a row are kept in a sorted order according to their keys (names). Examples are Google BigTable, HBase and Cassandra from Facebook (Cattell, 2010).

NoSQL DBMSs provide high throughput (Kriha, 2011). For example, the column-store Hypertable, which pursues Google's Bigtable approach allows the local search engine

Table 1 Database categories and features.

Data model	Performance	Scalability	Flexibility	Complexity	Functionality
Key – value store	High	High	High	Low	Variable (none)
Column store	High	High	Moderate	Low	Minimal
Document store	High	Variable (high)	High	Low	Variable (low)
Graph database	Variable	Variable	High	High	Graph theory
Relational database	Variable	Variable	Low	Moderate	Relational algebra

Table 2 ACID vs. BASE as Brewer Say (Gilbert and Lynch, 2002).

Item	ACID	BASE
Consistency	Strong consistency	Weak Consistency – stale data
Concurrent exec.	Conservative (pessimistic) lock	Aggressive (optimistic)
Availability	Available	Availability
Horizontal Scalability	Difficult	Simpler
Evolution difficulty	Difficult evolution (e. g. schema)	Easier evolution
Commit	Focus on "commit"	Approximate answers OK

Table 3 The differences between RDBMS and NoSQL Database.

Item	RDBMS	NoSQL
ACID/BASE	Provide ACID Prosperities	Provide BASE Properties
Table Structure	Need Fixed Table Structure	Does not need fixed table structure
Scaling up	Vertical Scale up	Horizontal Scale up
Consistency	Strong Consistency	Weak Consistency
Availability	Available	Available
Query capabilities	Rich Query Capabilities	Low Query Capabilities

Zvent to store one billion data cells per day. NoSQL DBMs provide efficient horizontal scalability which means that data can reside in more than one machine and if space is not enough for data, other machines can be added easily. While in vertical-scaling the data reside in a single machine and scaling is done through multi-core I.E. spreading the load between the CPU and RAM resources of that machine, with horizontal-scaling it is often easier to scale dynamically by adding more machines into the existing pool – Vertical-scaling is often limited to the capacity of a single machine, scaling beyond that capacity often involves downtime and comes with an upper limit. A good example for horizontal scaling is Cassandra, MongoDB. A good example for the vertical scaling is MySQL – Amazon RDS (The cloud version of MySQL) provides an easy way to scale vertically by switching from small to bigger machines; this process often involves downtime (Kriha, 2011).

In distinction to RDBMs, which provides ACID properties, NoSQL introduces what is known as BASE properties. The BASE approach, according to Brewer, who proved the CAP theorem forfeits the ACID properties of consistency and isolation in favor of "availability, graceful degradation, and performance". The acronym BASE is composed of the following characteristics: Basically Available, Soft-state, Eventual consistency. Brewer contrasts ACID with BASE as illustrated in Table 2 which summarizes the BASE properties in the following way: an application works basically all the time (basically available), don't have to be consistent all the time (soft-state) but will be in some known state eventually (eventual consistency) in contrast ACID properties provide strict consistency. Strict consistency means that all read operations must return the same data from the latest completed write operation. Such a strict consistency cannot be achieved together with availability and partition tolerance according to the CAP theorem.

Eventual consistency means that all read operations may return different data from the latest completed write operation, but as time goes on: "In a steady state", the system will eventually return the last written value. Clients therefore may face an inconsistent state of data as updates are in progress. For instance, in a replicated database updates may go to one node which replicates the latest version to all other nodes that contain a replica of the modified dataset so that the replica nodes eventually will have the latest version. Table 3 summarizes the similarities and differences between RDBMs and NoSQL.

2.3. Problems associated to NoSQL

There are many of the challenges that face NoSQL DBMS. One of the important challenges is how to add some degree of data consistency as well as providing ACID properties. There are some researches in that point such as CloudTPS, Megastore, WAS, COPS, Percolator and others. Also, as data stored in many servers, there is a challenge on how to make a fast search on such distributed data stored in NoSQL DBMS and how to get business intelligence information from distributed data. "Business Intelligence and NoSQL Databases" and "Hive: a warehousing solution over a map-reduce framework and as Distributed Search on Large NoSQL Databases" are some of articles that address the previous challenge (DUDA, 2012; Thusoo, 2009). How to distribute data over many nodes is one of the challenges that face NoSQL DBMS. Articles such as "NoSQL and Hadoop Technologies On Oracle Cloud" and "10 rules for scalable performance in simple operation datastores" address such challenges (Sharma, 2013; Stonebraker, 2011).

2.4. Distributed data and concurrency control

There are two styles of distributing data: Sharing the distributed different data across multiple servers, so each server

Table 4 Summary of the Previous Efforts.

Research	Goal	Performance Metrics	Results	Disadvantages
Megastore (2011) Baker et al. (2011)	– Provides fully serializable ACID semantics over distant replicas with low enough – Latencies to support interactive applications	– According to clients used the megastore – availability for read and write operations. – average read and write latencies	– Provides availability (at least five nines 99.999%) – average write latencies of 100–400 ms depending on the distance between data centers and the size of the data being written	– It requires a manual partition of data into groups and only provide ACID within this groups through MVCC MultiVersion Concurrency Control
Scalable Transactions across Heterogeneous NoSQL KeyValue Data Stores (2013) Kanwar (2013)	– Providing multi-item transactions across heterogeneous data stores, using only a minimal set of features from each store	– Throughput (transactions per second (TPS)) – while varying the ratio of reads to writes from 90:10, 80:20, to 70:30 – using 1, 2, 4, 8, 16, 32, 64, and 128 client threads with 10,000 – records accessed	– The number of transactions scales linearly up to 16 client – threads (this gives approximately 491 transactions per second with a 90:10 mix of read and write transactions respectively using a single WAS data store container) – With 32 threads, the number of transactions remains roughly the same as with 16 threads	– Depend on the assumption that data store returns at all the final version of data – Depend on global ordering of transactionsDepend on central clock
CloudTPS (2012) Wei (2012)	– Providing full ACID properties for multi-item transactions issued by web applications, even in the presence of server disappointments and network partitions – CloudTPS contains number of LTMs. Each LTM responsible for a subset of data – when web application submits a transaction a LTM act as a coordinator for it	– Throughput – Run CloudTPS over HBase and SimpleDB – Maximum TPS measured while increasing number of LTM from 5-10 to 15.40	– With HBase TPS scales linearly against number of LTM and reach 8000 tps with 40 LTM – With SimpleDB TPS reaches 3000 tps with 80 LTM	– Does not address problem of dead lock – Manually divide data over LTM – Requires machine for each LTM – Using 2pc in a network can delay execution of transaction – Depend on time stamp in distributed system provide single point of disappointment

acts as a single source for a subset of data. While the other is the replication which copies data across multiple servers, each bit of data can be found in multiple places. A system may use either or both techniques. Replication comes in two forms (Muhammad, 2011): master–slave replication, which makes one node the authoritative copy that handles updates while slaves synchronize with the master and may handle reads. The other one is called multi-master replication. In contrast to master–slave, it allows writes to any node and the nodes coordinate to synchronize their copies of the data. Master–Slave replication reduces the chance of update conflicts but multi-master replication avoids loading all writes onto a single point of disappointment.

Two operations conflict if they belong to different transactions, they access the same data item and at least one of them is a write operation. Write–Write conflicts occur when two clients try to write the same data at the same time, while read–write conflicts occur when one client reads data in the middle of another client update the same data. There are two approaches to assure consistency. The first are the Pessimistic approaches, which lock data records to prevent conflicts. The second is the optimistic approach, which executes any transaction even if it violates integrity rules such as serializability and after it detects conflicts and fixes them.

3. Related work

Recently, relational databases have been challenged by the emergence of NoSQL storage systems which typically relax consistency guarantees in favor of more scalability and availability. Different solutions introduced for highly consistent and providing ACID transactions based on NoSQL systems, key–value stores. One way is to implement transaction support in the data store itself (Baker et al., 2011). This is complicated and is difficult to implement without compromising scalability and availability. Another approach which is used in our proposed solution is to use a middle layer which acts as the interface between clients and DBMs. This layer will support ACID properties and concurrency control. Another approach is to define a transactional access protocol for each data store. This protocol provides a transaction and data store abstraction API to enable the client applications to access the data with transactional semantics. Also the proposed protocol maintains the

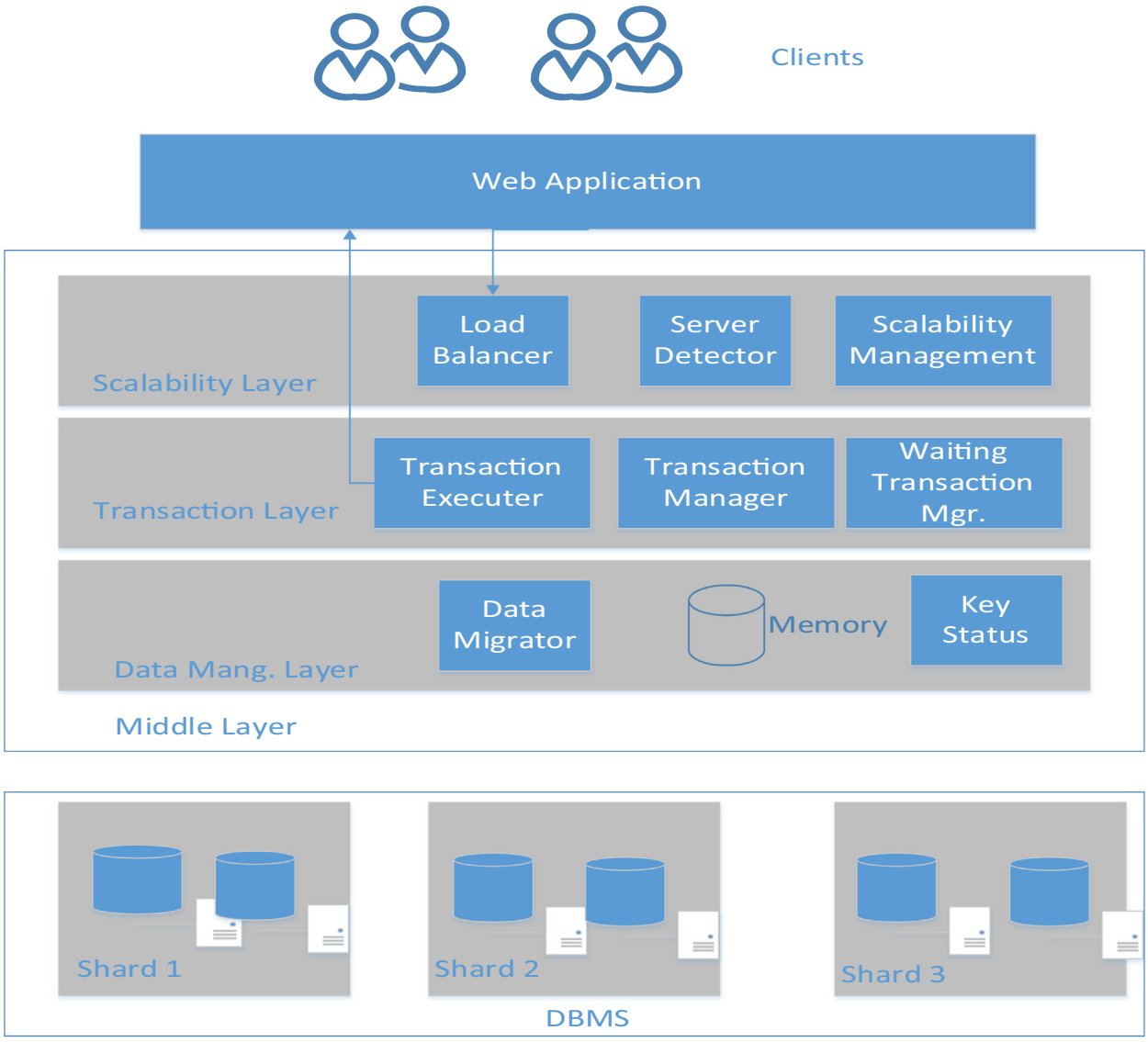

Figure 1 An overview of the proposed middle layer.

advantages of scalable and reliable access to the data store. But this approach requires to send metadata to the API from DBMS and client itself. Data given from clients cannot be accurate.

Google Megastore (Baker et al., 2011) is a transactional indexed record manager on top of Big Table. Megastore supports ACID transactions across multiple data items. However, programmers have to manually link data items into hierarchical groups, and each transaction can only access a single group. Spinnaker (Rao et al., 2011) uses a PAXOS protocol to build a Scalable, Consistent, and Highly Available Data store but it only provides a single item consistency guarantees.

COPS (Lloyd et al., 2011) is a key–value store that delivers this consistency model across the wide-area. A key contribution of COPS is its scalability, which can enforce causal dependencies between keys stored across an entire cluster, rather than a single server. The central approach in COPS is tracking and explicitly checking whether causal dependencies between keys are satisfied in the local cluster before exposing writes. COPS uses the get transactions in order to obtain a consistent view of multiple keys without locking or blocking. It uses replication protocol optimizations to achieve greater performance while supporting native multi-item transactions.

Granola (Cowling and Liskov., 2012) is a transaction coordination infrastructure for building reliable distributed storage applications. It provides a strong consistency model, while significantly reducing transaction coordination overhead. Granola introduces a specific support for a new type of independent distributed transactions, which can serialize with no locking overhead and no aborts due to writing conflicts. Granola uses a novel timestamp-based coordination mechanism to order distributed transactions, offering low latency and high throughput. It uses replication protocol optimizations to achieve a greater performance while supporting native multi-item transactions.

Scalable Transactions across Heterogeneous NoSQL Key Value Data Stores (Kanwar, 2013) defines a client API that defines a client coordinated transaction management protocol

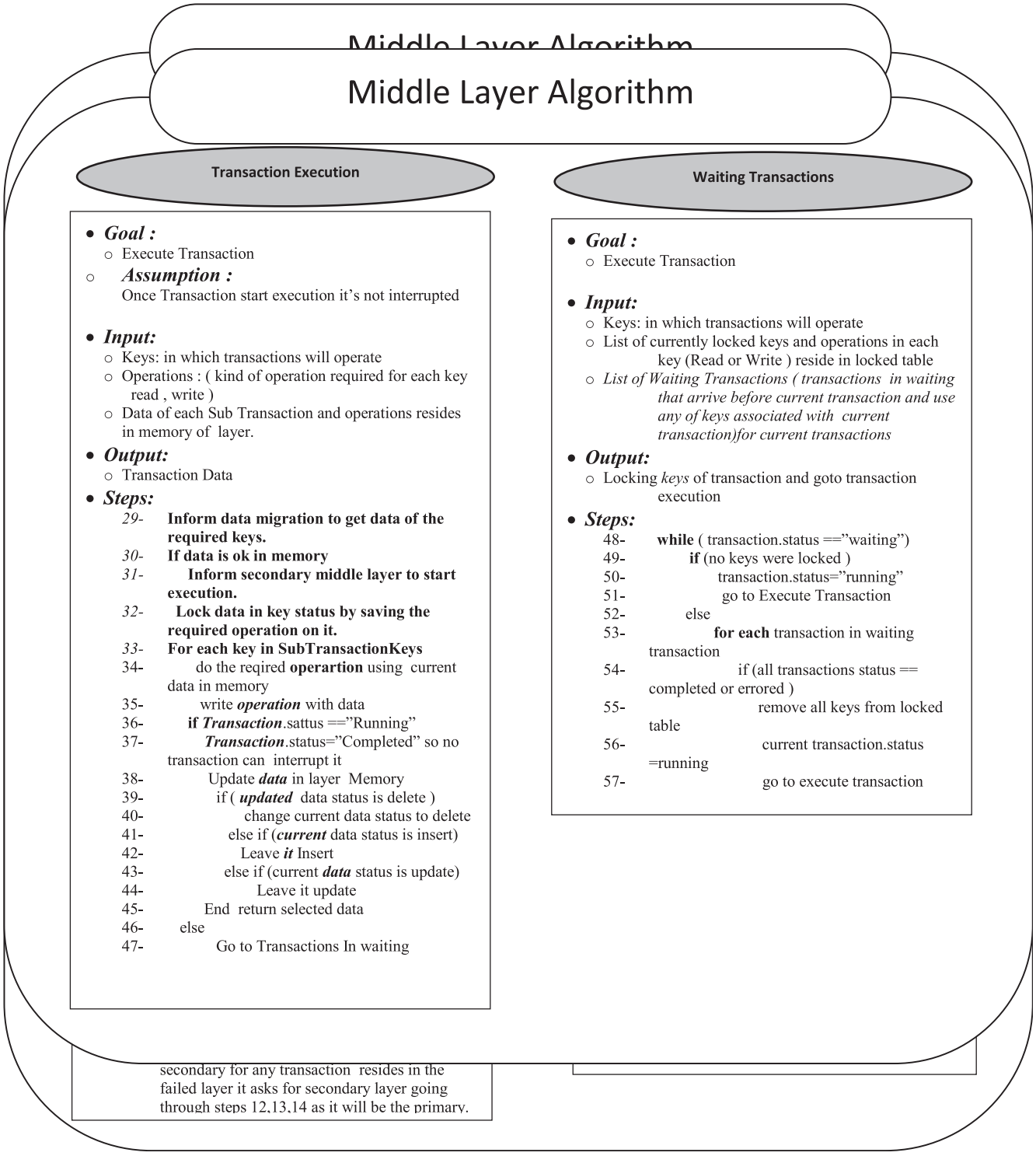

Figure 2 Middle layer algorithm.

with a pluggable data store abstraction layer enabling it to handle transactions across more than one data stores. It defines a client coordinated transaction protocol to enable efficient multi-item transactions across heterogeneous key–value stores by distributed applications. It also defines a data store implementation that provides a corresponding interface to support multi-item transactions. But this approach depends on a client clock which will give non accurate results and also it defines a central layer from which all clients get there data. This layer is a single point of disappointment. Percolator (Peng and Dabek., 2010) implements multi-key transactions with snapshot isolation semantics. It depends on a central fault-tolerant timestamp service called a timestamp oracle (TO) to generate timestamps to help coordinate transactions and a locking protocol to implement isolation. The locking protocol relies on a read-evaluate-write operation on records

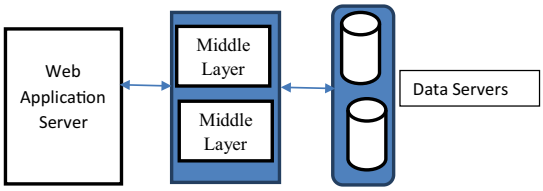

Figure 3 Experiment construction.

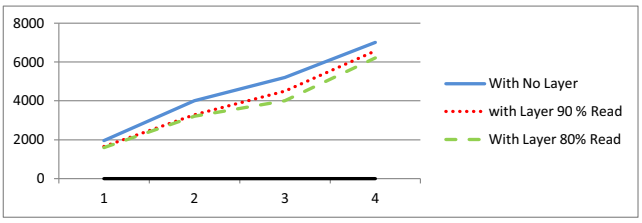

Figure 5 Throughput against number of requests: one middle layer.

to check for a lock field associated with each record. It does not take advantage of test-and-set operations available in the most key value stores making this technique unsuitable for client applications spread across relatively high-latency WANs. No deadlock detection or avoidance is implemented further limiting its use over these types of networks.

Cloud TPS (Wei, 2012) is a middle layer between client web application and cloud storage. This layer related to our work as it provides transaction ACID properties, but it does not address the problem of dead lock. Transactions are divided into sub transactions. Each sub transaction deals only with one key for reading or writing. Transaction executed if it's all sub transactions are ready to be executed. If one or many sub transactions are not ready the transaction is aborted and given another time stamp and restarted again from the first. So it can happen that a transaction will restart from the first for many times as another transaction operates in the same data. It does not give a schedule algorithm in order in which transactions are executed. Also, it uses in-memory data stores for all data in the underlying storage. Table 4 summarizes the previous efforts that tried to solve consistency and ACID properties of NoSQL.

4. Proposed solution

According to the CAP theorem proved by Eric Brewer any database management system can fulfill only two of the three properties, which are Consistency, Availability and Partitioning. Relational database provides consistency and availability with ACID properties while NoSQL databases provide Availability and Partitioning with BASE properties. The goal of this paper is to improve Consistency and provide ACID properties of NoSQL databases in concurrent transaction executions.

One of the solutions is to modify the NoSQL database engine itself, but this solution will depend on the modified database engine so it will not be suitable for other database engines. It also requires getting the source code which is not available for some engines and also it will take efforts for code understanding, like Megastore which is built for Big Table and Windows Azure Storage and Citrus leaf.

Another solution is based on building a layer over the NoSQL database engine that acts as an interface between the user application such as web application and the NoSQL database engine like cloud TPS. This middle layer will support transactional properties for the clients such as ACID and consistency and maintains the motives of NoSQL databases such as availability and Partitioning provided already by the underlying NoSQL database engine. The middle layer will depend only on how we can get or update data on the underlying database engine. So with minor modifications, we can make it suitable for many NoSQL database engines. Also, we can control the middle layer through turning it ON in situations when transaction management is important and turning it OFF in situations when transaction management is not important.

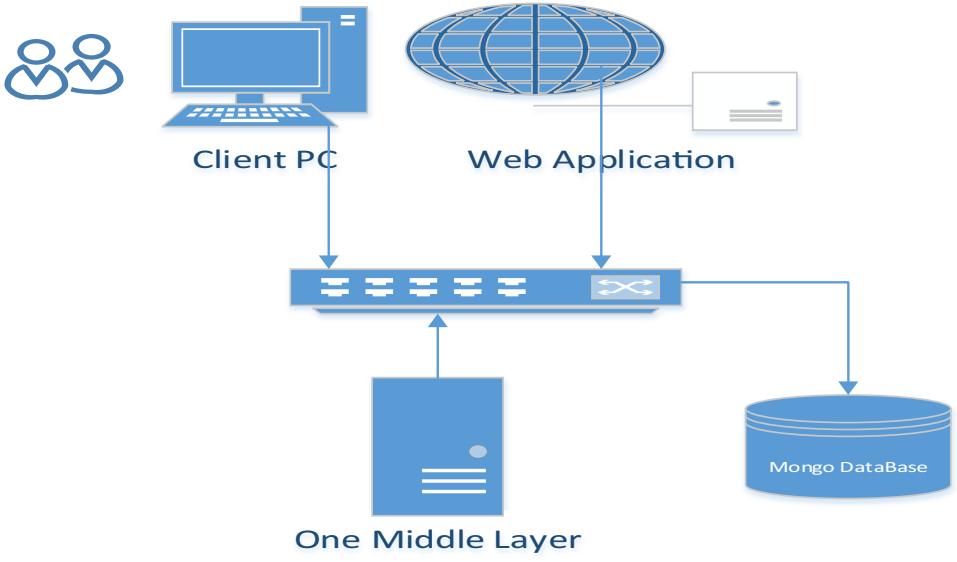

Figure 4 Experiment 1 construction.

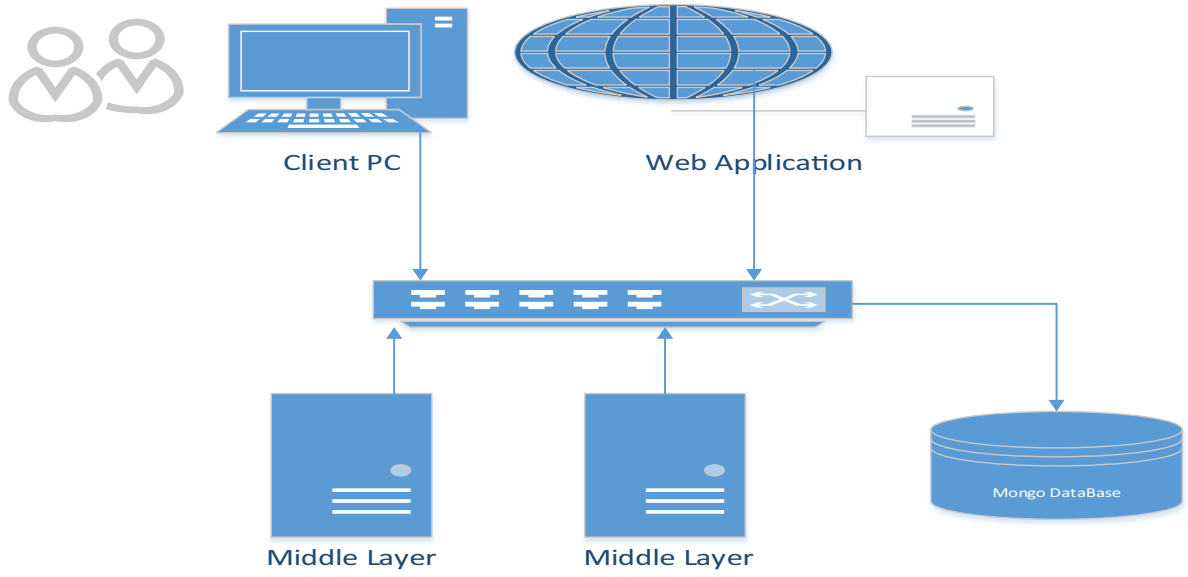

Figure 6 Experiment 2 construction.

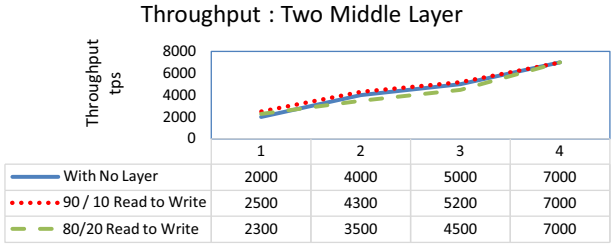

Figure 7 Experiment 2 result with two layers.

4.1. Proposed middle layer

Fig. 1 shows the architecture of the whole system with the proposed middle layer. Fig. 2 shows the algorithm of the main components of the middle layer. Clients issue requests to a web application using HTTP. The web application issues transaction to the middle layer. There is a multiple middle layer to avoid the center point of failure. The web application can submit a transaction to any of the existing middle layers.

The middle layer receiving the transaction will detect the available middle layer that will execute the transaction. The

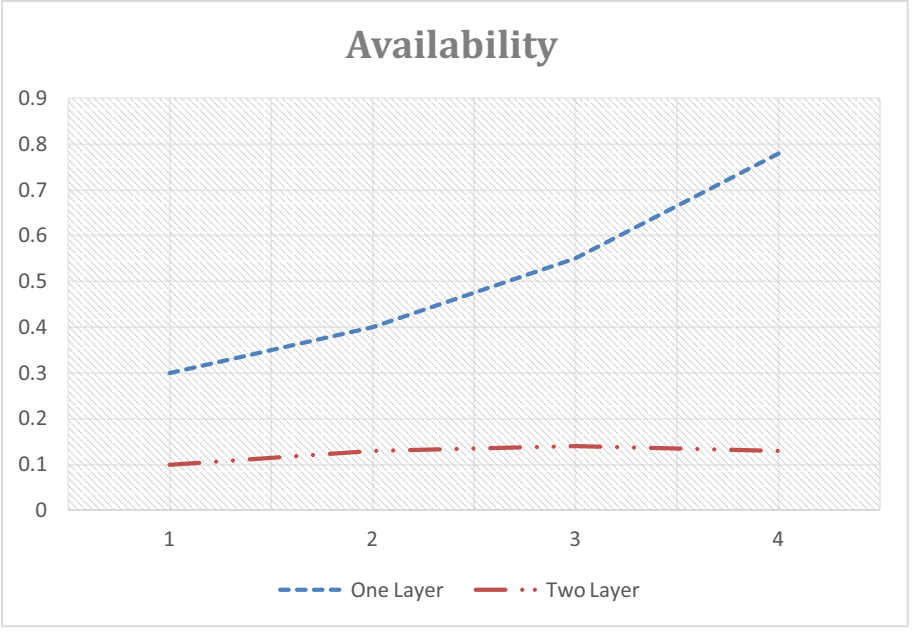

Figure 8 Availability of the system with one and two layers.

detected middle layer will act as a coordinator for the transaction. The coordinator (middle Layer) will collect data, execute transaction, migrate updates and return data to the web application. The coordinator initially loads data from NoSQL storage into the layer memory. The coordinator executes transaction in layer memory. The operation on data read or write is kept with the data (data status). Data status is used for concurrent transaction execution. Data updates resulting from transactions are kept in layer memory with new versions and these updates are replicated to other layers to prevent data loss due to middle layer server disappointments.

The transaction submitted by the web application is executed using 4-Phase Commit protocol. In the first phase the coordinator collects data from other middle layer memories with its version or from the underlying NoSQL database if data do not exist in any of the middle layers. In the second phase the coordinator selects another coordinator to execute transaction in parallel with it to make results available even in the case of the primary coordinator disappointment. In the third phase primary and secondary coordinator executes transactions. During The fourth phase the primary coordinator sends results to the web application and to the underlying NoSQL database engine while the secondary coordinator detects if a disappointment occurs; disappointment happens in the primary one or not. If it detects disappointment, the secondary coordinator returns the result to the web application and updates the underlying NoSQL database.

A load balancer will be used to detect which layer is responsible for acting as a coordinator for the transaction. The middle layer has in-memory copy of data initially loaded from NoSQL data storage through data migrator component. Data updates result from transactions migrated to another layer in order to get data in the case of layer disappointment and to NoSQL data storage and update keys-status component with the new version of this update. As NoSQL DB is horizontally scalable, we maintain the layer scalability manager which is responsible for scaling the middle layer. The server detection component responsible for detecting which node (data or layer) is connected to the system.

When a layer acts as a coordinator for a transaction the transaction management sends the keys in which it will operate to the data migration component to prepare data of such keys. Data Migration gets the last version of the key data according to Key Status component. The Key Status component maintains the location of the last version of data for each key and asks other layers to detect if any other layer operates or has a new version of the key. If the keys are available in key status component and no operation on it, it informs data migration to get data from the specified location into layer-memory and the operation (read or write) is saved with the key in the key-status component. When another layer asks for these keys it takes these keys with the version and operation and compares the operation, it wants with the current operation if any one of them (current or requested) is write operation the transaction is sent to the waiting transaction which periodically checks the key-status component in order to run the transaction. When the keys are available, the transaction is executed by the transaction executer component. During the execution of the transaction, the keys residing in key-status component have version and operation status which is read or write. When the transaction execution ended, the version of keys increases and operation set to free. Data migration component propagates changes to other layers and to No SQL DB.

4.2. Middle layer components

The middle layer is composed of three functional layers. The first layer is responsible for scalability and load balance, the second is responsible for transaction management and the third responsible for data and data status and version. The following sections explain the importance of each component.

4.2.1. Server detector

The server detector component is a critical component since it determines which middle layer and data server are currently connected to the network. The server detector maintains the currently connected server and periodically sends out a signal to these servers, informing them that it is alive and running. If the server detector does not receive any signal from a specific server it attempts to send a signal to this server and waits for the response. If the server detector does not receive any response it detects a server disappointment. If the server detector receives a signal from a server that is not currently in its list it detects a new server connected to the system.

4.2.2. Scalability manager

As NoSQL DB provides a good horizontal scalability for data storage or for the increasing in the incoming requests. We maintain this property in our layer by providing the scalability management component. This layer is responsible for increasing the number of layers when work load increases or layer memory data increase. So the scalability manager can be used to increase the size of the middle layer memory or to enter a new layer in the system if the workload increases.

4.2.3. Load balancer

When a layer is added or removed from the network, the Load Balancer component is invoked to rebalance the work within the Middle layer. The rebalancing occurs only on the waiting transactions. The rebalancing occurs on the number of the waiting transactions. Only one layer is responsible for making this balance.

The load balancer receives the transaction from the web application and detects the primary coordinator for the transaction. It also makes a rebalance to the waiting transactions if the system layers increase or decrease. Also it is responsible of detecting the secondary coordinator.

4.2.4. Data migrator

The data migration component is responsible for getting data from other middle layer memory. If data do not exist in any of the middle layer memories the data migration component gets it from the underlying NoSQL DB. If data are updated, the data migration component migrates these updates to the other middle layers and to the underlying NoSQL DB. Also, this component is also used to accomplish the residence of data inside the middle layer memory to make a room for other data. The data should be removed from the middle layer memory when there is another higher version of the same data in another middle layer memory. Also data with a higher version are removed from the memory when there is a need to location

in memory and the updates migrated to the underlying NoSQL database and its age exceeds the limit set by the admin. The age of data in memory is determined from the last time it was used. The system can modify the age of data. The removal of data starts automatically when the data size in memory becomes bigger than a threshold size identified by the admin. The removal of data starts by removing the old version of data and then by the old age data if the updates migrated to the underlying NoSQL database.

4.2.5. Keys status

Key status component is responsible for maintaining the location of the last version of the data and the current operation on it. The version of data is an incremental number that increases when any portion of this key data updated. The status of the data indicates the current operation on data such as read write or no operation. If a layer makes a write operation on a specific key the middle layer is stored in the key status with key and operation write with this key.

4.2.6. Transaction management

Transaction management component is responsible of providing ACID properties for the transaction. It contains two main components. One component that executes the transaction by applying 4PC 4 phase commits protocol explained in the next section while the other contains the waiting transactions. Hence, it periodically checks the availability of transaction running.

4.3. How the system provides ACID properties

The goal of the proposed layer is to provide the ACID properties of RDBMs in the NoSQL databases. In this section, we discuss how the proposed layer provides atomicity, consistency, isolation and durability properties in NoSQL databases through the use of 4PC.

4.3.1. Atomicity

For multi key transactions, which consist of multiple keys we divide such transactions into a number of sub transactions according to the number of keys. So each sub transaction only deals with one key for read or write. If one or more sub transaction failed, the whole transaction will be failed, leaving the database without any changes. To ensure Atomicity, we perform 4 phase commit protocol. Tacking into consideration the short life of a transaction, as soon as data are available for operations in the local memory of the coordinator, the transaction is executed without interruption.

In the first phase of the commit protocol, the coordinator brings all data required for transaction from other transaction managers or from the underlying NoSQL database. When the coordinator is ready to execute the transaction, it replicates transaction with data to other transaction manager selected by the load balancer to act as a secondary coordinator. The primary and secondary coordinators start to execute transaction. If the primary coordinator succeeded it returns the updated data to the web application and commits the transaction in the fourth phase with replicating updated data to N transaction managers. If there is a hardware disappointment in the primary coordinator detected by server detection of the secondary coordinator, the secondary coordinator executes the transaction and returns data to the web application and replicates data to the other layer to commit the transaction.

In the proposed protocol there is only one node (coordinator) responsible for executing the whole transaction after data are available in the first phase. We provide also a secondary coordinator that executes the same transaction with the same data sent to it in the second phase of the protocol. The secondary coordinator acts as an alternative for the primary coordinator if disappointment occurs in the primary coordinator while it executes the transaction. Also secondary coordinator acts as another source of data if a hardware disappointment occurs in the primary coordinator after the primary coordinator executes the transaction and updates do not migrate to the underlying database.

4.3.2. Consistency

Consistency means that execution of transaction brings the database from consistent state to another consistent state. This is done by checking the accurate data types used for declaring variables used to hold data.

4.3.3. Isolation

For the concurrence execution of transactions, isolation means that execution of a transaction has to use only committed data. So if multiple transactions operate on a number of the same key data we have to schedule (order) transactions to ensure serializability. In our protocol the execution of transactions in the third phase does not start unless all used keys are free to be used.

We use a combination of locking and time stamp mechanisms. When a coordinator executes a transaction, it divides the transaction into sub transactions. Each sub transaction makes read or write only in one key. When any coordinator executes a transaction, it marks all keys with the operation needed in keys status such as K1: R, K2: W and so on. When the execution is finished the operation is removed from the keys and the updated keys take a new version. When another transaction wants to operate on any of the keys that are being used by another already running transaction, if the two operations on the key are read, the second transaction is executed, otherwise the second sub transaction takes a number N, which is determined from the currently running coordinator. This number is incremental, so, if another transaction wants to run with the previous transaction status it takes another number $N + 1$. When a transaction starts execution and there is no conflict with any other running transactions, it takes number 1. If another transaction is received on another coordinator and there is a conflict with transaction number 1 on key K the key K in the current transaction takes the number 2 and the whole transaction is put in the waiting status. When the execution of the first transaction is finished the execution of the lowest number transaction should start.

When a coordinator wants to update a key, it locks this key by the operation writes and any other incoming transaction wants to operate on this key, it takes a number from the coordinator currently locking the key. When the coordinator currently locking the key commits the update, the coordinator with the lowest number manages the key by starting executing the operation and gives the other incoming transaction a new number starting from the last number $+1$.

4.3.4. Durability

Durability means that committed data have not to be undone. In the third phase of the protocol, we commit data after data are replicated in more than one layer and use data migration and data evictor to migrate changes to the underlying NoSQL database. If there is any hardware disappointment in the second phase of our protocol, we make a secondary coordinator that executes the transaction and through its server detection it determines if the primary coordinator has hardware disappointment or not. If there is hardware disappointment in the primary coordinator the secondary coordinator migrates updates to another layer and to the underlying NoSQL database. We make assure that the updates migrate to multiple servers before returning data to the client.

5. Performance metrics

The performance of the middle layer can be evaluated in terms of numbers of metrics such as throughput, availability of the layer, consistency and scalability. In our case of the proposed middle layer, we will measure performance in terms of Transaction throughput and availability of the layer.

Transaction throughput is the number of transaction executed per second (TPS). Concurrent transactions will be sent to the middle layer through a number of threads. The time at which transaction completed is written in the database. And the number of transactions executed per second is evaluated. We will check transaction throughput against the number of client requests. Each request contains a transaction. The transaction read to write ratios is varying from 100% read to 0 write, from 90% read to 10% write and from 80% read to 20% write.

Availability refers to the ability of the user community to obtain a service or good access to the system, whether to submit new work, update or alter existing work, or collect the results of the previous work. If a user cannot access the system, this is – from user points of view – unavailable. The term downtime is used to refer to periods when a system is unavailable. Transaction throughput will be measured during failover one of replicated servers. To increase the availability of our system consistency layer is repeated over two servers' Examination: We will examine the system performance in existence of the two servers and we will examine also its performance in existence of only one server.

The steady state availability of any component of the system can be calculated according to the equation (Teorey and Teck, 1998):

$$Ai = MTTFi/(MTTFi + MTTRi)$$

where MTTF is the mean time for the component I to fail and MTTR is the mean time for component I to repair. This function will be applied on our layer in case of a transaction executed and the layer failed and returned to run after repair. The failure occurs in case the system contains one layer and the failing occurs in this layer, and in case there are two layers and failure occurs in one layer.

There is a short note about consistency which means that all users will deal with the same data at the same time. In the traditional sense, a consistency property is something that a system either provides or fails to provide. Thus, the property can be verified, but not measured. We verify these properties by executing multiple transactions sent from many users in a specific order and we obtained the correct data.

6. Experimental results

This section provides the experimental results to validate the proposed middle layer. Experimental results show that the system throughput approximately increases linearly against the No. of requests and is promoted by increasing the number of middle layers. We build the middle layer over mongo NoSQL DB. Engine that runs on the server with web application hosted on another server and make several requests to the middle layer through the web application. Fig. 2 shows the construction of the experiment. Clients issue request to web servers which make use of the middle layer to get data from the underlying mongodb database. A quick description of the procedure used in the experiment is illustrated below

- We build the underlying NoSQL database using mongodb.
- Data stored in database represent a 10,000 record of student data in *Data Server 1*.
- Student data include key, fname, lname, birthdate, faculty, choice1, choice 2, IQ, English, Email.
- These data (10,000) record is replicated to another *Data server 2*.
- The data are stored in bson format as following {key:1; fname: "ahmed"; lname: "aly"; IQ:60; English:40}.
- There are other *two Middle Layer Servers* each one containing the algorithm that maintains ACID properties. The layer is programed using c#.
- There is one server that contains a web application; we will call it *web application server*.
- All servers are connected using switch.
- Each server contains inter core i5 processor – 4 G Ram.
- Fig. 3 shows the logical connection between servers.

6.1. Experiment 1: Evaluating the system using one middle layer

We measured the throughput in the existence of only one middle layer. The web application can issue one request, two requests at the same time, three requests at the same time or four requests at the same time. Each request demands to execute many transactions approximately 3000 transactions. Fig. 4 shows the experiment architecture.

Fig. 5 shows the experiment 1 results. The vertical axis demonstrates the average throughput (TPS) transaction per second and horizontal axis shows the number of requests issued to the middle Layer. Performance measurements were taken while varying the ratio of reading to writing in transactions from 90 R: 10 W, 80 R: 20 W using 1, 2, 3and 4 requests at the same time all operate on different keys. We increased the ratio of writing to read each time to increase workloads as writing takes more time than read. Also we used web applications to send one request at first. The transactions were executed sequentially. Also as the number of requests at the same time increases the number of the required transactions to be executed and controls the interference between data. The general structure of transaction includes

(update(fname where id = 1),read(fname where id = 1), read(fname where id = 2), read(fname where id = 1), read

(fname where id = 1), read(fname where id = 1), read(fname where id = 1), read(fname where id = 1), read(fname where id = 1), read(fname where id = 1)).

The previous transaction includes 90%R:10%W and when we make update two statements and read 8 statements this is 80%R:20%W.

Approximately the number of transactions in all cases scales linearly. As the number of requests increases the number of transactions executed per second increases as all requests operate on a different thread. With no layer the transaction throughput is greater as no extra processing is needed in the middle layer but data are eventually consistent. Increasing ratio of writing makes throughput to decrease using one layer.

6.2. Experiment 2: Evaluating the system with two middle layer

In this experiment we increase the number of middle layers to two middle layers and we measured the throughput. Fig. 6 shows the architecture of the experiment. The client can issue one request, two requests at the same time, three requests at the same time or four requests at the same time. Each request demands to execute many transactions approximately 3000 transactions. The result showed that under two numbers of requests the throughput in the first experiment is high compared to that in the second experiment. As the amount of requests increases the throughput in the second experiment is high compared to that in the first experiment. Fig. 7 shows the result.

6.3. Experiment 3: Evaluating the system availability

The availability of the system is calculated in existence of one and two layers. In the case of one layer the time that system takes to complete the execution of 1, 2, 3 and 4 transactions is calculated and in the case of the middle layer the time taken when it fails and returns is calculated. Also the time that system takes to execute the transaction in case there are two layers and one of them is failed is calculated in ms. The experimental result is shown in Fig. 8.

7. Conclusion and future work

Many applications such as bank application transactions and real time systems require strong data consistency. This application can get strong data consistency if it is operated over RDBMs. But there are some limitations that RDBMs face such as scalability. This leads us to use another type of databases called NoSQL databases. NoSQL databases provide only weak consistency which makes it not suitable for applications that require strong consistency. This paper has introduced a new framework that we can use between applications that require strict consistency and NoSQL databases. This frame work can provide strict consistency without affecting scalability and availability of NoSQL databases.

This work relies on 4PC (4 phase commit) protocol to ensure atomicity, consistency, isolation and durability. In the first phase we obtain data in which transaction will operate from layers or NoSQL database. The second phase selects the primary and secondary coordinators that are used to execute transaction. The third is the execution of the transaction. The fourth is replicating data to other layers and updating the underlying NoSQL database. If the primary coordinator fails while executing transaction the secondary one continues executing the transaction without any delays. The throughput of the systems maintains the same when there is a failure in one layer as there is another layer executing the same transaction at the same time.

One of the drawbacks of the system is that, multiple layers are used to execute the same transaction to avoid central point of failure. This approach is the computational power of the system. As a future work we want to improve framework to use only one layer and when there is failure in this layer another layer starts to execute the transactions executed by this layer. Also the framework has to be built to be more intelligent to be used over many different kinds of NoSQL databases.

References

Baker J., Bond C., Corbett J.C., FurmanJ., Khorlin A., Larson J., Leon J.-M., Li Y., Lloyd A., Yushprakh V., 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: Proc. Conf. Innovative Data Systems Research (CIDR).

Cattell Rick, 2010. Scalable SQL and NoSQL Data Stores, ACM, vol. 54, No. 6.

Cowling J., Liskov B., 2012. Granola: low-overhead distributed transaction coordination. In: USENIX ATC'12.

DUDA JERZY, 2012. Business Intelegence and NoSQL Databases, Information Systems in Management.

Franco M., Nogueira M., 2011. Using NoSQL Database to Persist Complex Data Objects, Conference, Instituto de Informática Universidade Federal de Goiás (UFG).

Gilbert S., Lynch N., 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services, SIGACT News, vol. 33, No. 2.

Kanwar Renu, Trivedi Prakriti, Singh Kuldeep, 2013. NoSQL, a Solution for Distributed Database Management System, International Journal of Computer Applications (0975–8887) vol. 67, No.2.

Kriha, Walter., 2011. NoSQL databases, course of studies. Hochschule der Medien, Stuttgart University.

Lloyd, W., Freedmand, M.J., et al, 2011. Don't settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. ACM.

Milanović, A., Mijajlović, M., 2012. A Survey of Post-relational Data Management and NOSQL movement. Department of Computer Science, Faculty of Mathematics University of Belgrade, Serbia.

Muhammad, Yousaf, 2011. Evaluation and Implementation of Distributed NoSQL Database for MMO Gaming Environment Master's thesis. Uppsala university, Department of Information Technology.

Orend, Kai, 2010. Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer Master's thesis. Faculty of informatics, Technology University, Munkh.

Peng D., Dabek F. 2010. Large-scale incremental processing using distributed transactions and notifications. In OSDI'10.

Rao J., Shekita E.J., et al., 2011. Using paxos to build a scalable, consistent, and highly available datastore. In: Proc. VLDB Endow Conf.

Sharma Vatika, 2013. Meenu Dave, NoSQL and Hadoop Technologies On Oracle Cloud, IJETTCS.

Stonebraker, Michael, Cattell, Rick, 2011. 10 Rules for Scalable Performance in 'Simple Operation' Datastores. ACM.

Toby J. Teorey, Wee Teck Ng, 1998. Dependability and Performance Measures for the Database Practitioner. IEEE.

Thusoo, Ashish et al, 2009. Hive: A Warehousing Solution Over a Map-Reduce Framework. ACM.

Valer, Henrique, 2013. XQuery Processing Over NoSQL Stores Master thesis. Technical University, Kaiserslautern.

Wei Zhou, Pierre Guillaume, Chi Chi-Hung, 2012. CloudTPS: Scalable Transactions for Web Applications in the Cloud. In: IEEE Transactions on Services Computing.

Further reading

Calder, B. et al, 2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. ACM.

Dewitt, D., Gerber, B., Graefe, G., Heytens, M., Kumar, K., Muralikrishna, 1986. GAMMA – A High Performance Dataflow Database Machine. In: Proceedings of the 1986 VLDB Conference.

Furman J.J., Karlsson J.S., Leon J.M., Newman S., Lloyd A., Zeyliger P., 2008. Megastore: A Scalable Data System for User Facing Applications, In: Proc. SIGMOD Int'l Conf. Management of Data.

Rys Michael, 2011. Microsoft Corp. Scalable SQL, ACM, vol. 9, No. 4.