Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A scalable generic transaction model scenario for distributed NoSQL databases

Ramesh Dharavath[*], Chiranjeev Kumar

Department of Computer Science and Engineering, Indian School of Mines, Dhanbad 826004, Jharkhand, India

ARTICLE INFO

Article history:
Received 21 February 2014
Revised 22 September 2014
Accepted 20 November 2014
Available online 28 November 2014

Keywords:
Atomic transaction
Column-oriented distributed databases
NoSQL databases

ABSTRACT

With the development of cloud computing and internet; e-Commerce, e-Business and corporate world revenue are increasing with high rate. These areas not only require scalable and consistent databases but also require inter database transaction support. In this paper, we present, a scalable three-tier architecture along with a distributed middle-ware protocol to support atomic transactions across heterogeneous NoSQL databases. Our methodology does not compromise on any assumption on the accuracy of failure modalities. Hence, it is suitable for a class of heterogeneous distributed systems. To achieve such a target, our architectural model exploits an innovative methodology to achieve distributed atomic transactions. We simulate this architectural setup with different latency tests under different environments to produce reliable impact and correctness.

1. Introduction

Distributed computing infrastructures were emerged as the deployment platforms for a variety of applications. Increasingly, desktop and mobile applications are using distributed infrastructure to take advantage of high scalability characteristics. Massive usage of internet produces huge amount of data. These vast volumes of data are also called Big Data. This information can be used in many ways. This Big Data is beyond the capability of traditional databases. New type of databases have been introduced to deal with this Big Data, these are called column oriented databases and also known as NoSQL which are scalable, eventual consistent, and reliable. Due to these features NoSQL databases like Big-Table (Chang et al., 2006, 2008), Cassandra (Lakshman and Malik, 2010), HBase (Vora, 2011), MongoDB (Membrey et al., 2010), and Amazon S3 (DeCandia et al., 2007) are successfully used by almost all internet giants. NoSQL databases offer great scalability and availability for applications.

Relational database systems (RDBMSs) also dominate the environment by providing set of services which refer variety of requirements. But, these require analytical processing and decision support. However, some trepidation has recently materialized towards RDBMSs. First, it has been argued that, there are cases where RDBMSs performances are not passable. Second, the structure of the model is considered to be too rigid and not useful in some instances with the arguments that call for the semi structured data (Abiteboul et al., 1999). At the same time, the full power of relational models with multifaceted transactions and queries would not require in some perspectives, where simple operations are quite enough (Stonebraker and Cattel, 2011). Also, in some instances, ACID consistency and complete consistency assured by relational model is not essential. It has been observed that many internet application areas related to networking domain need both scalability and flexibility in construction, while being fulfilled with operations and weak forms of consistency. With these drives, a number of new systems are not following the RDBMS models. Their futures are limited and support simple operations with limited flexibility in the structure of data. According to McKinsey Global Institute (Manyika et al., 2011), the data volume is growing at a rate of 40% per year. From this, we conclude that NoSQL is better for large data volume applications, where the date rate increases rapidly. There is wide variety of systems with NoSQL (Cattel, 2010; Chang et al., 2008) exposes a different interface models and APIs. Certainly, it has been pointed that, the lack of standard is a great trepidation for organizations in adopting any of these systems (Stonebraker, 2011), applications, and data are not portable and reusable with others. Also, each of these systems has specific objectives, and it is customized on few specific prototypes. As an outcome, it might be possible that complicated applications could benefit from the use of NoSQL systems. Traditional databases are useful for those applications, where consistency and transaction support is needed in more suitable fashion. But, the current trend applications such as corporate world, e-business and e-commerce needs database support which is not only scalable as well as consistent too. With the massive success of NoSQL databases and growth of e-commerce, e-banking and corporate data, we assume

[*] Corresponding author. Tel.: +91 326 2235795; fax: +91 326 2296563/2296613.
E-mail addresses: ramesh.d.in@ieee.org (D. Dharavath), k_chiranjeev@yahoo.co.uk (C. Kumar).

http://dx.doi.org/10.1016/j.jss.2014.11.037
0164-1212/© 2014 Elsevier Inc. All rights reserved.

that, in future, NoSQL databases can be used in e-commerce and e-banking areas. In order to preserve atomicity and isolation related instances, in a previous work (Ramesh et al., 2012a,b, 2013), we have implemented an architectural methodology for multi row transactions on column oriented distributed databases using RDBMS. We have also proposed a suitable methodology that preserves accuracy of atomic transactions in heterogeneous distributed column oriented database environment (Dharavath et al., 2014). The observations above have motivated us to look for tools and methods that can improve the consequences of the heterogeneity in distributed NoSQL database systems and can also enable interoperability between them.

As a first step in this direction, in this paper, we present here a *"distributed three-tier architecture"* as well as *"distributed middleware protocol"* to support atomic transaction across heterogeneous distributed NoSQL databases. Proposed framework is entirely distributed; hence it is infinitely scalable, highly available, very flexible and cost effective. We also exemplify the correctness and effectiveness of our framework in different aspects.

2. Related literature

Frolund and Guerraoui (2001, 2002), have recently proposed a reliability framework called e-transaction (exactly once transaction), that provide atomicity for distributed transaction. An e- transaction framework is based on three tier architecture and protocol implementation for each tier is clearly explained. In this framework, there is a chance of single point failure and scalability issue. There have been several works that actively replicates database systems in this way. Pacitti et al. (2003), Whitney et al. (1997), Stonebraker et al. (2007), and Jones et al. (2010) proposed about performance of transaction processing in a distributed database without concurrency control mechanism. Moreover, these methodologies execute the transactions serially which is equivalent to a serial order i.e. where a node can be a single CPU core in a multi-core server (Stonebraker et al., 2007). By executing the transactions serially, no determinism due to thread scheduling of concurrent transactions is eliminated, and active replication is easier to achieve. However, serializing transactions can limit transactional throughput, since if a transaction stalls (e.g. for a network read), other transactions are unable to take over.

Other researchers have proposed techniques for providing multi-row transactions (Peng and Dabek, 2010), (Zhang and Sterck, 2010). The Percolator system (Peng and Dabek, 2010) addresses the problem of providing SI-based transaction for Big- Table (Chang et al., 2008). However, it does not address the problem of ensuring serializability of transactions. Moreover, it is intended for offline processing of data. The work presented in (Zhang and Sterck, 2010) does not adequately address issues related to recovery and robustness when some transaction fails. Kallman et al. (2008) proposed a technique which is related to main memory transaction processing. This technique has not explored in scaling the performance of transactions which were situated in main memory. Dadiomov et al. (2003), patents a system for distributed transaction processing with asynchronous message delivery. According to this, a transaction is combination of many database operations. These operations spans over many distributed databases. All operations in a transaction must be performed atomically. To implement this system authors use message queue (MQ) concept. This message queue (MQ) guarantee exactly-once in-order message delivery. For each transaction there is a coordinator. With the help of message queue (MQ) message is passed between coordinator and participating databases. In this proposed system, atomicity can be achieved using two-phase commit (2PC) protocol.

The problems of transaction serializability have been extensively studied in the form of different models (Fekete et al., 2005; Bornea et al., 2011; Cahill et al., 2009; Revilak et al., 2011; Jung et al., 2011). The work presented by Fekete et al. (2005) exemplifies about necessary conditions for non-serializable transaction executions. Based on this, many approaches have been suggested to avoid serialization anomalies. These approaches include static analysis (Jorwekar et al., 2007) as well as runtime detection of anomalies (Bornea et al., 2011; Cahill et al., 2009; Revilak et al., 2011). Methodology presented by (Bornea et al., 2011; Cahill et al., 2009; Jung et al. 2011) tend to be distrustful and can lead to unnecessary aborts. The PSSI approach (Revilak et al., 2011) avoids such problems and aborts only the transactions that lead to serialization anomalies. However, these approaches were developed in the context of traditional relational databases and (except in Jung et al., 2011) provided solutions only for centralized databases. Thomson et al. (2012) proposed a Calvin layer to support fast and scalable transactions. Calvin layer is concerned about replication and scheduling. Calvin has no single point failure. This transaction model is useful to get high throughput but replication relaxed ACID is an issue with this layer. Another method called MAV (Monotonic Atomic View) proposed by Peter et al. (2013), which requires disallowing reading intermediate writes of a transaction. But it incurs two writes for every client side write and achieves 75% of the throughput only. Han et al. (2011), made a survey on NoSQL database that explains distinct column-oriented databases like MongoDB, Cassandra, Hyper table etc. This paper also describes about data model used by NoSQL and explores features of NoSQL. In this paper, we describe a framework to provide atomicity for distributed transactions. Proposed framework is scalable and flexible in nature and recovers from failure instances. In novelty, this framework fulfils present as well as future requirements related to real world applications.

3. Proposed methodology

In this work, we describe and present a distributed architecture (depicted in Fig. 1) to provide atomic transaction instance across heterogeneous distributed column-oriented NoSQL databases. This architecture is scalable and flexible in transactional issues. This architecture is mainly inspired by Apache HBase data-store architecture (Lars, 2011). In HBase (Vora, 2011), master nodes are managed by zookeeper. Master is used for load balancing and managing region nodes. In the same way in our architecture, manager nodes are managed by zookeeper and manager is used for balancing the load among labour nodes. HBase is having the control to store and retrieve BigData, in other hand our architecture is used to provide atomicity for transaction across heterogeneous distributed NoSQL databases. As name implies, three-tier architecture is divided physical entities in three layers: (i) client (ii) middle-tier, and (iii) distributed database (HBase).

3.1. Client

Architecture supports multiple client nodes denoted by C_1, C_2 … C_p. User interacts with the entire architecture by utilizing the

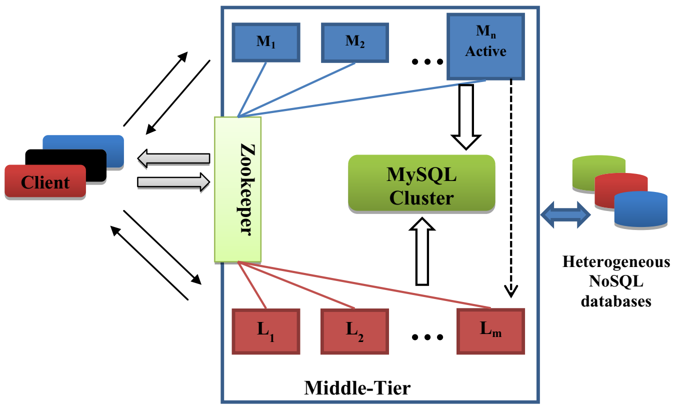

Fig. 1. Schematized three-tier architecture.

services of client node. The responsibility of client node is to wrap transaction request and deliver it to corresponding *GTM* (Global Transaction Manager) as well as request to manager node to get the address of associated *GTM.* We propose a fully transparent distributed architecture where client not only knows location of all database servers also knows syntax of delivered request to each database server as well.

3.2. Middle-tier

This is a place where atomicity is provided. Middle-tier is scalable and flexible in nature, which contains multiple nodes. The classified nodes are as follows.

3.2.1. Labour node

Multiple labour nodes ($L_1, L_2, \ldots, L_m$) provide scalability for atomic transactions. Every *GTM* runs on any labour node. These labour nodes are managed by manager node. If a new labour node joins the architecture, first, it contacts zookeeper to know about active manager node, and then register itself to manager.

3.2.2. Global transaction manager (GTM)

This architecture supports multiple global transaction manager processes ($GTM_1, GTM_2, \ldots, GTM_n$). In order to handle each transaction request, a *global transaction manager* process is created. Because, *GTM* can be regarded as *"serializing"* all the transaction processing, people may think that *GTM* can be a performance bottleneck but it ensures the integrity of the data in a high speed network like Gigabit. Moreover, coordinator backend is provided with *GTM* (i.e. client library) to obtain snapshot and to report the transaction status. GTM opens a port to accept connection from each coordinator. When GTM accepts a connection, it creates a thread (GTM thread) to handle request to GTM from the connected coordinator. The same can be preserved by using an active manager node but it cannot ensure the data integrity. This *global transaction manager* runs on any one of labour nodes and managed by manager node. Based on the progress, every *GTM* maintains a state which is stored as meta-data. *GTM* works as a coordinator for assigned transaction. Every *GTM* location and assigned work can be identified with the help of meta-data. This is an advantageous aspect if one *GTM* is crashed then other *GTM* takes place of it.

3.2.3. Manager node

Any one of multiple manager nodes ($M_1, M_2, \ldots, M_n$) is active at a time. If one goes down then another manager node is elected by bi-directional ring election algorithm (Ramesh et al., 2012a,b). Two processes run on active manager node; (1) manager process and (2) cleaner process. *Manager process* allocates *GTM* for each client request. Manager also balances the load among all labour nodes. The responsibility of *cleaner process* is to recover all failed and incomplete transactions. Cleaner process always runs on active manager node and keeps the track of failed and incomplete transactions. If cleaner process gets any failed or incomplete transaction, then it creates recovery process on one of any labour node and assigns this transaction for recovery.

3.2.4. MySQL cluster

MySQL cluster is used to store meta-data related to transactions. This meta-data is used by *GTM* process and manager node to provide atomicity for transactions. MySQL cluster is distributed and there is no single point failure.

3.2.5. Zookeeper

Zookeeper is a service for coordinating processes of distributing applications. Zookeeper provides per client guarantee of FIFO execution of requests (Chin et al., 1992). The zookeeper interface enables a high-performance service implementation. Zookeeper is used to manage multiple manager nodes. Zookeeper also provides distributed services (Manager Election, naming, distributed synchronization) for the given three-tier architecture.

Table 1
Transac_table structure.

Transaction_id	Timestamp	Write-set	Status_id
TID1	TS = 150	List of item IDs	Begin, prepare, commit, rollback, complete, fail, abort
TID2			

3.3. Distributed database

There are multiple heterogeneous NoSQL databases ($DB_1, DB_2, \ldots, DB_k$) participating with different transactions. We assume a process called *database-server* process is running corresponding to a database and handle each transaction request, which is initiated by the proposed middle-tier architecture. Our implementation is based on HBase (Lars, 2011) as NoSQL database, which meets all requirements.

3.4. Meta-data structure

Information about the atomicity of a transaction is called as meta-data. This meta-data is used to recover a transaction. In place of any distributed file system like hadoop, we use MySQL -cluster to store this meta-data. The reason is that, MySQL -cluster is not only reliable also scalable and sql compatible. It also gives easy and faster access to the stored data as well, compared to distributed file system. First, we describe the metadata that needs to be organized in the global storage for the transaction. We then describe various steps in transaction protocol performed by all the application processes.

3.4.1. Transaction data table

First, we identify the features of the system which requires transaction mechanisms. The system should provide support for tables and multiple columns per row, and primitives for managing data items with timestamps. It should support consistency for updates, i.e. when a row is updated. Moreover, for the heterogeneous distributed architecture, we require some methods for performing transactions involving in any number. For each transaction, we maintain the following information: transaction_id (TID), timestamp (TS), write_set information, and status_id. This information is maintained in a table called Transac_table in the storage as shown in Table 1. In this table, column's transaction_id is the primary key of table; with the help of transaction_id each client's transaction can be uniquely identified.

To ensure that the Transac_table does not become the bottleneck, we partition the table across all the HBase servers. The data distribution in HBase is based on sequential range partitioning. The column status_id is foreign key which refers status_id column of status table. Every transaction has an associated status value which changes gradually according to the progress of transaction. For each data table, (represented as Storage_table: Table 2) we maintain the information related to the committed data items of transactions and lock information. A transaction writes a new data item with its *TID* as the timestamp. These timestamps then need to be mapped to the transaction commit timestamp when transaction commits. This mapping is stored by writing TID in a column named *committed* with timestamp. The column "write lock" in the Storage_table is used to detect write–write conflicts, whereas columns "read lock", and "read-ts" are used to identify read-write conflicts for serializability.

Table 2
Storage_table structure.

Item	Committed	Read_Ts	Read Lock	Write Lock
Item 1	value 1 80 value 2 100 value 3 130	Time Stamp	TID/N ULL	TID/N ULL
Item 2				

Table 3
Status table.

Status_id	Description
0	Begin
1	Prepare
2	Commit
3	Rollback
4	Complete
5	Fail
6	Abort

3.4.2. *Status table*

Status table has two columns (depicted in Table 3); status_id and description. Status table is static in nature and cannot be changed.

3.4.3. *DB_Server table*

DB_Server table is used to recover failed transactions. This table has three columns; transaction_id, db_loc and transaction request. First, column transaction_id is a foreign key which refers transaction_id column of Transac_table. Second, column db_loc store address of column oriented database where transaction request has to be send. Third, column transaction request is actual request that has to be forwarded to column oriented database.

4. Middleware protocol

4.1. *Working scenario*

Proposed middleware-protocol is highly scalable and distributed in nature. The processes participating in this protocol are: (1) client, (2) zookeeper, (3) manager, (4) *GTM*, and (5) database servers. In this section, we describe working of entire protocol in brief.

Step 1: User submits a transaction to client.
Step 2: Client sends manager request to zookeeper to know about active manager.
Step 3: Zookeeper responds the client with active-manager address.
Step 4: Client sends GTM request to know about free GTM node.
Step 5: By load-balancing algorithm manager calculates free GTM process and respond the client with its GTM process address.
Step 6: Client submits transaction request to associated GTM and waits for the result.
Step 7: GTM sends prepare message to all participating database-servers.
Step 8: If a database-server is ready to commit the assigned sub-transaction respond with 'Yes' otherwise respond with 'No' vote. After that database server gets blocked until receives decision message.
Step 9: If GTM gets all votes with 'Yes' message, GTM replies to all participating database-servers with 'commit' decision. Otherwise decision should be 'rollback'. After sending decision messages, GTM waits for acknowledgement.
Step 10: Database-server gets decision message and works accordingly. Then respond GTM with acknowledgement.

Algorithm 1: Commit protocol executed by a transaction T for our model

```
Validation phase:
if status = active then
  status ← validation
end if
insert write-set information in Transac_table
for all item ∈ write-set of Ti do
begin row level transaction:
  if any committed newer version for item is
    created then abort
  if item is locked then
  if lock-holder's TID < TIDitem, then abort
     else wait
     else acquire lock on item by writing
TIDitem
     in lock column end if
    end row level transaction
end for

Commit-Incomplete phase:
if status = validation then
  status ← commit-incomplete
  else abort
  TimeStamp← get commit timestamp from
  TimestampService
for all item ∈ write-set of Ti do
  insert TStamp→ tid mapping in the
  Storage_table and release lock on item
end for
status ← commit-complete

Abort phase:
for all item ∈ write-set of Ti do
  if Ti has acquired lock on item, then
    release the lock.
delete the temporary version created for item by
Ti
end for
```

Step 11: After receiving all acknowledgements, GTM completes the transaction and respond client accordingly.
Step 12: Client receives the transaction result and forward to user and client remains wait to receive another transaction request from user.

Transaction submitted by the user contains multiple database operations. Objective of the submitted transaction is to follow ACID properties during execution. Zookeeper is used for the purpose of configuration of entire architecture. Zookeeper also provides flexibility for architecture. Availability of more than one manager provides reliability to the architecture. Multiple *GTM* processes runs on different labour nodes, these multiple labour nodes are used to provide scalability. Proposed system uses Two-Phase Commit Protocol (2PC) with some changes to provide atomicity for global transaction. The transaction execution with respect to this protocol is shown in Algorithm 1. System maintains all logs (related to global transactions) in MySQL-cluster.

4.2. *Algorithmic approach*

Proposed three-tier architecture works with multiple algorithms, those runs on distinct nodes. These algorithms are inspired by e-transaction algorithm proposed by Frolund and Guerraoui (2001) and 2PC protocol proposed by Boutros (1996).

4.2.1. *Client algorithm*

When a client process receives a transaction request; (1) it invokes *connection ()* method to contact manager node and request to create an associative *GTM* process which handles this client transaction request. (2) Client contacts the allocated *GTM* by invoking *clientExec ()* method to complete a transaction. When user requests for a transaction, a client object is created by invoking constructor of client class. After that client invokes *connection ()* method of client class. Client sends a message with its client id to allocate a *GTM* by invoking *sendToManger ()* method, after that client remain blocks until it gets acknowledgement from the manger. If client gets positive *ACK*, then it initializes the parameters of *GTM* class and invokes *clientExec ()* method to complete the transaction. If client gets *NAK* (negative acknowledgement) from the manger then client sends *unsuccessful* message to user and terminates the transaction. The *clientExec ()* method sends all sub transaction requests to allocated *GTM* and wait for the result. Each sub-transaction request is forwarded to different column oriented databases, based on database address by *GTM*.

These sub-transaction requests are stored in a list of *request objects* at the time of client object is created. Each transaction must have a lifetime that is calculated by *period* variable and that value is continuously decreases according to time. If a client receives *abort* message or *period* reached to 0, then client sends unsuccessful message to user. Otherwise, transaction is successfully committed by all cohorts and passes information to the user by sending *successful* message.

4.2.2. *GTM algorithm*

For each incoming request, active manager allocates a labour node. Then labour node creates new *GTM* process (thread). This *GTM* process works as a coordinator for associated transaction. *GTM* invokes *execGTM ()* method when it gets transaction request from client. In Entire exec*GTM* () method is divided into three phases as described below:

Prepare phase: First, *GTM* inserts a tuple in transaction table that contains associated transaction_id := tid and transaction state := 0 (begin). *GTM* fetches database server address and associated sub transaction according to client request and then redirects a *prepare request* to each participating database server. This prepare request contains actual sub-transaction request that is sent by client and *transaction id* to identify each transaction uniquely by database server. *GTM* also saves each sub transaction request with corresponding database server address and corresponding *transaction id* in *DB_Server* table for the recovery instance. At last, *GTM* updates transaction state to 1 (prepare) and move to decision phase.

Decision phase: In this, *GTM* waits for vote message from participated database servers for a particular transaction. A database-server process sends *yes vote,* if it is ready to commit all participating rows, if not then sends *no vote* to *GTM*. We consider a variable named *compCount* and assigns value of count (count variable contains number of participating database servers in this transaction). *GTM* waits for vote response from database servers. For each *yes* vote *GTM* decrements *compCount* variable by one until it reaches zero. If *compCount* value reaches to zero, then all participating databases are ready to commit their transactional instances (i.e., all databases acquired the locks on required rows and remain blocked for *GTM* decision). Now *GTM* takes commit *decision*. *GTM* sends *commit* message with *transaction id (tid)* to all participating cohorts and transaction moves in commit state by updating its entry in transaction table with transaction_id := tid and transaction state := 2 (commit). Before the *compCount* reaches to zero, if *period* expires or any database server responds with *no vote,* then *GTM* decides to *rollback* transaction and moves to *rollback* state by updating the entry in transaction table with transaction_ id := tid and transaction state := 3 (rollback). After then *GTM* informs all cohorts with *rollback* message and transaction moves in abort state by updating entry in transaction table with transaction_id := tid and transaction state := 6 (abort). Also responds to client with *abort* message and *execGTM ()* method gets terminated.

Response phase: After sending *commit* message to all participating database servers, *execGTM ()* method moves to response phase. In response mode, *GTM* waits for acknowledgement from all participating database server so that *GTM* can confirm whether all subtransactions are properly committed or not. After that, *GTM* waits until it receives acknowledgement from any participated database servers or period expires. If period is expired then transaction moves to fail state by updating entry in transaction table with transaction_id := tid, transaction state := 5 (fail) and sends *abort* message to the relative client. Then execClient *()* method gets terminated. If an acknowledgement is received by the *GTM* then *compCoumt* value increments by one until the value reach to *count* value (count value shows that number of database servers are participating in this transaction). Before starting the response phase, *compCount* value is set to zero because *GTM* moves to response phase from decision phase when *compCount* value becomes zero only. Now *GTM* checks whether *acknowledgement* is received from all participating databases or not, if yes, then *compCount* value is same as *count* variable. That means transaction is successfully completed and *GTM* sends *success* message to relative client. After that *GTM* updates its transaction state by updating entity in transaction table with transaction_id := tid and transaction state := 4 (complete) and *execClient ()* method gets terminated.

4.2.3. *Database server algorithm*

We assume that, a database-server process is running on each column oriented distributed database site in order to listen incoming requests. This database-server process handles each request of *GTM* ($GTM_1, GTM_2, \ldots, GTM_n$). When server gets a *prepare-request*, it creates a new clone process of database server and remains listen to the channel. On the other hand, newly created clone process handles the assigned transaction only. After that, this *clone database server* process creates a new instance of DB class and initializes the parameters *receivedData* and *tid* by the value which comes with prepare-request. After that, execDB () method is invoked. The responsibility of a database server process is to map incoming requests (each request must contain *transaction_id* parameter) like prepare-request or decision-request. The mapping functionality can be performed based on *transaction_id* and *clone process id*. The corresponding pseudo code is shown in Algorithm 2 (as clone distributed database server). In order to perform transaction activities like commit, rollback we adopt *sql statements* to work with column oriented NoSQL databases. This is because of each column oriented database uses its own operation called *CRUD operation* which is the part of client transaction request.

Here, we give precise information according to the pseudo code which is represented as Algorithm 2. First, *clone DB process* creates an instance of DB class and initializes the parameters by passing the values (these values come with transaction request). After that, execDB () method is invoked and waits for GTM *response* (**line-2**). If GTM responds (**line-3**) with prepare request, then *clone DB* tries to acquire locks on all participating rows because it gets prepare request first time (**lines-5 and 6**). By checking the *prepare-flag,* clone DB process conforms that, whether *prepare-request* message has been arrived or not (**line-4**).

If execDB () method is not able to acquire locks on any participating row (**line-7**) then it releases all acquired locks. By sending *no-vote* primitive it also informs to the GTM process that, database is not ready to commit this transaction *(*shown in **line-9**).After that, execDB () method sets *prepareFlag* to avoid duplicate *prepare-request* messages (**line-10**). Now (**line-11**), the execDB () method performs the computation on participating rows accordingly. After that, execDB () method makes the loop variable false and method remain gets blocked until it gets the response from GTM. ExecDB () method sends (**line-13**)

Algorithm 2: Clone distributed database server

```
Class DB {
        RequestData receivedData;
        String tid;
        Boolean flag, loop;
        Boolean prepareFlag, decisionFlag;
        Public DB(RequestData receivedData,
String tid){
                this. receivedData := receivedData;
                this. tid := tid;
                prepareFlag := false;
                decisionFlag = false;
loop = true;
        } execDB() {
1. while ( true )
2. wait till (loop or reveivedData:=[GTM
                                    response])
3. If (receivedData is [prepare-message] ) then
4.              If ( prepareFlag == flase) then
5.              For each row t_i in receivedData
6.          flag = lockIfFree(t_i);
      //return true if successfully takes lock
7.              If(flag == false)
8.              release all acquired lock
9.              [respondToGTM ( No )] and exit;
10.     prepareFlag = true;
11.     do computation on rows
12.     loop = false;
13.     [respondToGTM ( yes )];
14.If (receivedData is [decision-message] ) then
15.     If ( decisionFlag == flase) then
16.             If Decision is Commit then
17.             decisionFlag = true;
18.             Commit (tid);
19.     Else
20.     rollback (tid) and exit;
21.     release all acquired Lock
22.     [sendAckToGTM ( ack )];
            }
    }
```

yes-vote to GTM that implies database is ready to commit and again the method remains blocked. If GTM responds with *decision-message* then the execDB () method finds about the GTM decision (commit or rollback). If commit is a decision message (shown in **line-16**), then execDB () method commits this transaction (**line-18**). After that, execDB () method releases all acquired locks (**line-21**) and sends an acknowledgement (*ack*) to GTM. If decision is *rollback*, then execDB () method rollbacks this transaction and takes exit (shown in **line-20**).

4.2.4. Recovery algorithm

A process called cleaner, runs on active manager node. This cleaner process recovers all transactions which are either *incomplete* or *failed*. For each transaction (transaction which is failed or incomplete), cleaner process creates a recovery process that runs on any one of chosen labour. A transaction is said to be incomplete, if it has 0 – begin or 1 – prepare or 2 – commit or 3 – rollback status values along with crashed *GTM*. If a transaction has a status as 5 – fail then it comes under *failed transaction* category. The transactions belong to *incomplete* or *failed* transaction category is handled by recovery algorithm. Recovery algorithm makes a transaction in *abort* state by performing rollback functionality on all sub-transactions. The related pseudo code is presented as Algorithm 3.

Algorithm 3: Recovery algorithm

```
Class Recovery
{
        List of Request requests;
        List of DatabaseServer
receivedAddress;
        List of RequestData receivedData;
        String tid;
        Int status;
        TimeOut period;
        Int count, compCount;
        execRecovery(String receivedTid){
1. status := SELECT status FROM Transaction
table with transaction_id := tid
2.      If (status == 1 or 2 or 3 or 5)
3.          tid := receivedTid;
4.      requests := (SELECT * FROM
DB_Server table where transaction_id := tid)
5.     For each request r_i in requests
6.     ReceivedAddress[i]:= r_i.dbServer;
7.     Received Data[i] := r_i. rdata;
8.     For each ra_i in receivedAddress
9.         SendDecisionToGTM( rollback , tid );
10. UPDATE Transaction SETstatus_id:=
     6(abort) WHERE transaction_id:=tid
            }
}
```

4.3. Protocol description

We describe a distributed three-tier architecture with multiple nodes. Multiple processes run on these nodes. These processes communicate with each other through message passing mechanism. Each process has its own state which changes according to time. In this section, we describe the functionality of these processes.

4.3.1. Client process

When a user requests for a transaction then a client process is created. At this time, client process is in *create* state. Client process sends a *connection* request to active manager to know about associated *GTM* process and enter into *ready* state. Manager successfully allocates a *GTM* process to the client by sending *ACK* (ACK contains binded *GTM* process id and location, so client can directly communicate with *GTM*) message. After receiving *ACK* client moves to *bind* state. If a manager is not able to allocate *GTM* to client process, then it sends *NAK*. When client process receives this *NAK*, it moves into unsuccessful state. In *bind* state the client calculates transaction id (tid) and initializes period parameter (period is a parameter on which transaction lifetime depends). After that client process sends transaction request to corresponding *GTM* with [request, tid, period] message and moves to *wait* state. If period is expired or *GTM* sends [abort] message then client moves from *wait* state to *unsuccessful* state. If *GTM* sends [success] message then client process moves from *wait* state to *successful* state. Fig. 2 describes about state transition diagram of a client process.

4.3.2. GTM process

Fig. 3 depicts state transition diagram for GTM process**.** Labour node creates *GTM* process when it gets *create GTM* request from manager. Initially *GTM* process is in create state. When *GTM* process receives *connection* request (this message contains not only client original request but client id as well so that *GTM* process only accepts request from attached client only) from the manager then it moves to

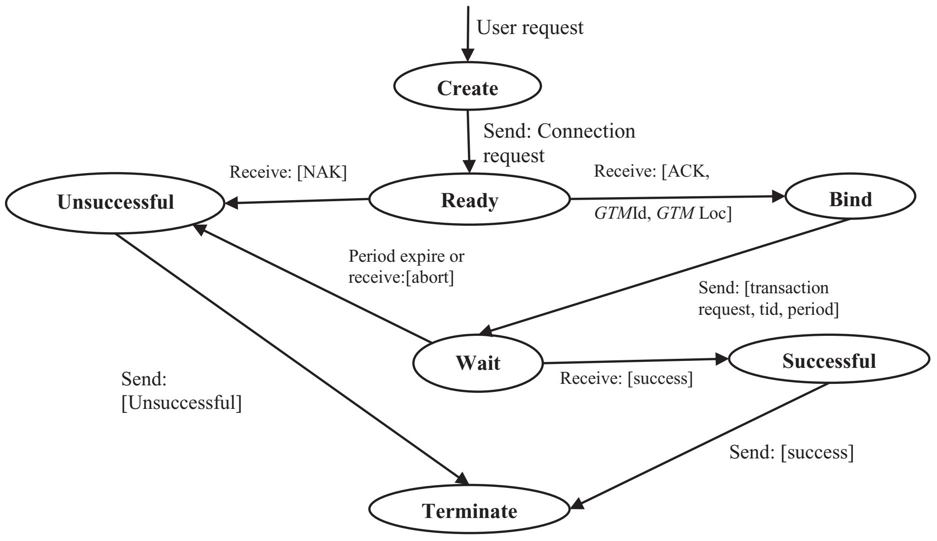

Fig. 2. State transition diagram of client process.

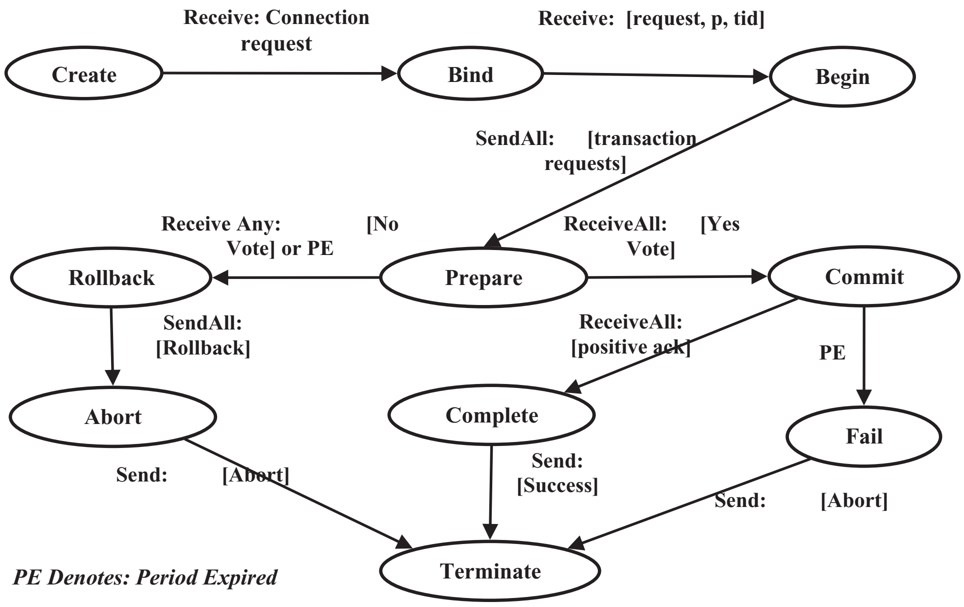
Receive: Connection request
Receive: [request, p, tid]
Create
Bind
Begin
SendAll: [transaction requests]
Receive Any: [No Vote] or PE
ReceiveAll: [Yes Vote]
Rollback
Prepare
Commit
SendAll: [Rollback]
ReceiveAll: [positive ack]
PE
Abort
Complete
Fail
Send: [Abort]
Send: [Success]
Send: [Abort]
Terminate
PE Denotes: Period Expired

Fig. 3. State transition diagram of *GTM* process.

bind state. After receiving a transaction request [requests, p, tid] from associated client *GTM* process moves from *bind* state to *begin* state. For the purpose of recovery *GTM* process also maintains state information in durable and stable storage (MySQL-cluster). In *begin* state, *GTM* redirects each sub-transaction request to the corresponding database-server and moves to *prepare* state. By receiving *yes vote* from all participating databases *GTM* process moves from *prepare* state to *commit* state. A database server sends *yes vote* to *GTM* if it acquire locks on corresponding rows. If *GTM* gets (in *prepare* state) *no vote* from any database-server or if *period* expires then transaction moves to *rollback* state. In *commit* state, *GTM* sends *commit* request to each participating database server and wait for response. If a database-server commits the sub-transaction then it sends *positive ack* to *GTM*. *GTM* moves from *commit* state to *complete* state if it gets *positive ack* from all participating database servers. If period expires then *GTM* moves from *commit* state to *fail* state. In *rollback* state *GTM* sends *rollback* message to all participating databases to rollback assigned sub-transactions and moves to *abort* state. *GTM* sends an *abort* message (from both *abort* and *fail* state) to client process and gets terminated.

4.3.3. Database-server process

Fig. 4 depicts state transition diagram for database-server process**.** Database-server process (DB process) creates a clone process when it gets a transaction request (T_i) from *GTM*. After that, it initializes the parameters (*param*) of DB object. It also invokes execDB() method and then moves to *waiting* state. In the waiting state DB process waits for *GTM* response. *GTM* can send three kinds of messages: (1) *prepare* message, (2) *rollback* decision message, and (3) *commit* decision message. If DB process gets a *prepare* message from *GTM* then it moves to *prepare* state. If DB process gets a decision message as *commit* then it moves to *commit* state. If DB process gets a decision message as *rollback* then it moves to *rollback* state. A DB process receives a decision message after receiving a prepare message only. The DB process moves from *prepare* state to *ready* state if locks are acquired on all participating rows (r_i). If not, then all acquired locks should be released and DB process responds *GTM* with *no vote* message (*no vote* means database is not ready to commit the sub-transaction). DB process moves from *ready* state to *waiting* state by sending *yes vote* to the corresponding *GTM*. In *commit* sate DB process commits the transaction. After commit, the DB process releases all acquired locks

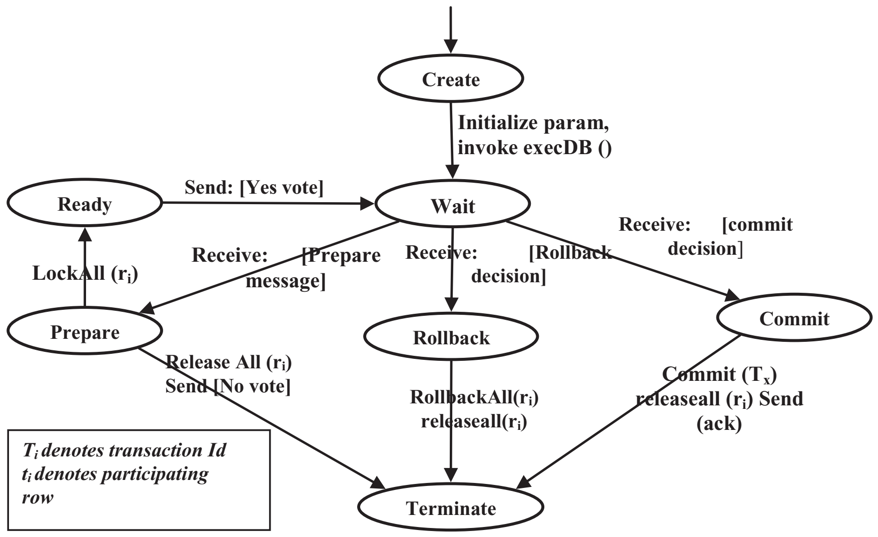
Create
Initialize param, invoke execDB ()
Ready
Send: [Yes vote]
Wait
Receive: [commit decision]
LockAll (r_i)
Receive: [Prepare message]
Receive: [Rollback decision]
Prepare
Rollback
Commit
Release All (r_i) Send [No vote]
RollbackAll(r_i) releaseall(r_i)
Commit (T_x) releaseall (r_i) Send (ack)
T_i denotes transaction Id
t_i denotes participating row
Terminate

Figure 4. State transition diagram of database server process.

and sends an acknowledgement (*ack*) message to *GTM* (assume *ack* is about successful commit of sub transactions). In *rollback* state, DB process undo the sub transaction and then releases all acquired locks.

4.3.4. Recovery process

Recovery process is running on any one of labour nodes. As name implies a *recovery process* recovers a transaction which is either incomplete or failed. *Cleaner* process creates a recovery process and put forward the transaction request with *transaction-id*. The recovery process rollbacks the transaction and keeps the transaction in abort state.

5. Protocol evaluation

We assume that, (i) channel between distinct nodes is noiseless and reliable i.e. data should be delivered without effects, (ii) there is no propagation delay and transmission delay to transfer a message from sender to receiver, (iii) clocks are synchronized among all participating nodes, and (iv) there is no clock drift among nodes. Given three-tier architecture is fully transparent and client is aware of symentic behaviour, syntatic nature and locality of participating NoSQL databases.

Tier-1 (client): A client process handles one transaction request at a time. By the above assumptions *period* variable used in algorithms explained *above* contains same value. On a client node each client process can be uniquely identified by with its *client-id.*

Tier-2 (middle-tier): We assume that, at least one manager node is working properly to take the charge of *active-manager.* In our proposed architecture, each labour node has a unique id (name). With the help of labour node id *active-manager* accesses an individual labour.

Tier-3 (HBase-NoSQL database): Each database maintains its own log record for recovery.

Protocol described in Section 4 is composition of multiple algorithms (described in Sections 4.2.1, 4.2.2, 4.2.3, and 4.2.4). These algorithms runs on different nodes and provide atomicity for transactions across distributed column oriented databases. In this section, we examine the correctness of our protocol in different aspects.

Lemma 1. *Middleware protocol never block infinitely until entire middle-tier architecture goes down.*

Proof by contradiction. Assume that, cohorts DB_i, DB_j and DB_k are participating in a transaction T_x which is coordinated by GTM_p. In first phase, GTM_p sends prepare request to all databases. All participating databases receive the request message and responds with yes vote accordingly. After responding with yes vote message all the databases will remain blocks. Our assumption by contradiction is that, a cohort DB_j gets blocked forever. Blocking of DB_j can be happened in two ways (i) decision message is not transmitted to DB_j. (ii) Before sending decision message to DB_j, coordinator process GTM_p has been crashed.

Reflection of case (i). GTM_p transmits a decision message to DB_j by invoking of exec*GTM*(), but the decision message is not received by DB_j. This case is possible because of: (a) DB_j goes down and (b) channel is not reliable. Condition (a) is not possible because DB_j is blocked forever and waits for decision message. Condition (b) is not possible because the assumption is already made that channel is reliable and noiseless. By the contradiction, condition (a) and (b) are against to case (i). Reflection of case (ii). As we describe in Section 3, if, a coordinator process (GTM_p) has crashed, then manager elects a new labour node. This labour node creates *a recovery process* to recover the transaction T_x. *Recovery* process sends decision messages to DB_i, DB_j, DB_k and DB_j gets unblocked.

Lemma 2. *A participating database DB_i does not commit a transaction T_x more than once.*

Proof by contradiction. Assume that databases DB_i, DB_j are participating in transaction T_x and DB_i *commits twice for its allocated sub-transaction* t_{xi}. DB_i only commits a transaction twice if it gets *commit-decision* message twice.

As we describe in Section 7, channel is reliable and *GTM* algorithm never sends a *commit* decision twice. By any chance if, *GTM* process goes down (after sending *commit* decision-message) then a process called *recovery process* comes in to the picture and *rollbacks* the committed sub-transaction t_{xi} (if any). By using a *decision-flag*, we can handle the case called "*if any DB_i gets a commit decision-message twice*". When a *commit* decision-message arrives at first time, a *decision-flag* is set and t_{xi} is committed by database-server algorithm. When a *commit* decision message arrives again, then t_{xi} will not be committed because *decision-flag* is already been set.

All the above explanation is contradict that, DB_i commits twice for its allocated sub-transaction t_{xi} which belongs to transaction T_x.

Lemma 3. *Each and every transaction has unique transaction id.*

Proof. For a transaction, client process allocates a *transaction_id*. This *transaction_id* is the combination of clients current time, *GTM* Id (id

Table 4
Hardware configuration of used cores.

Mach. Name	Address	RAM	Processor	OS
Host 1	172.16.4.222	3 GB	Intel-i3 3.10 GHz	UBUNTU-12.04
Host 2	172.16.5.198	4 GB	Intel-Core 2 Duo 2.99 GHz	UBUNTU-11.10
Host 3	172.16.7.30	2 GB	Intel-i3 3.10 GHz	UBUNTU-11.10

Table 5
Software configuration of used cores.

Machine name	JDK	MySQL-server	Database driver
Host 1	Oracle JDK 7	MySQL-server 5.6	MySQL-connector-java-5.1.24
Host 2	Oracle JDK 7	MySQL-server 5.6	MySQL-connector-java-5.1.24
Host 3	Oracle JDK 7	MySQL-server 5.6	MySQL-connector-java-5.1.24

of allocated *GTM*), and labour node's Id (on which *GTM* process is running). When a labour node joins the architecture, the active manager assigns an Id to labour node. With this Id, each labour node can be uniquely identified in the architecture. Every labour node supports (at a time) multiple *GTM* processes which can be uniquely identified by *GTM*-Id. Two *GTM*-Ids can be similar if both *GTM* processes belong to different labour nodes. As we describe in Section 2, a *GTM* process can handle only one transaction request. As we already describe in Section 7**,** we assume that a client can request for single transaction a time. With this explanation, we can say that combination of time, *GTM*-Id, and labour node Id gives a unique transaction id.

Lemma 4. *Either all participating cohorts decide to commit transaction T_x or none (preserving atomicity property).*

Proof. *GTM* commits a transaction when it gets *yes-vote* from all participating databases (cohorts). Here, *yes-vote* implies that, sender database is ready to commit according to its allocated sub-transaction. Then *GTM* sends a message called *commit* decision message to all participating cohorts. Cohort commits the subtransaction by receiving *commit* decision message from *GTM*.

Lemma 5. *Client delivers a transaction result to the user, which is same as the result computed by associated GTM process.*

Proof. As per the client algorithm's methodology, a client responds to user when the period expires or *GTM* replies with result. Here, period is lifetime of the transaction. Client replies with one of two messages to the user: (i) transaction is successful or (ii) transaction is unsuccessful. Here, we discuss all the cases when client responds to user. Client sends *unsuccess* message to user, because active manager does not allocate a *GTM* process successfully. If a *GTM* is not allocated, then client will not send any transaction request to any *GTM*. Client sends *unsuccess* message to user, when *period* is expired or *abort* result is received from relative *GTM* process. If *GTM* aborts the transaction, then it informs with *abort message* to the client. After receiving this abort message, client informs with *unsuccessful* message to the user. The second situation is about *expiry of period* convinced by the assumption (discussed in Section 7). This assumption implies that, if client's *period is expired* then *GTM* also calculate its *expiry of period* at the same time. When period expires, *GTM* will never decide to *commit* the transaction but moves to *fail* state. After that, cleaner process rollbacks failed transactions. Because, this client replies the user with a *unsuccessful* message, that is totally agreed by *GTM*. Client sends *success* message to user, when *GTM* successfully *commit* all subtransactions and reply with *success* message to client.

6. Background setup for simulation

In this section, we describe a prototype to simulate our proposed framework. This simulation work will help to understand the complexity of real environment distribution transaction system. Further prototype is used to evaluate the correctness and performance of proposed framework.

6.1. Hardware configuration

We simulate the proposed framework using different machines. All machines are connected through LAN/WAN. The configurations are given in Table 4.

6.2. Software configuration

In order to simulate the proposed framework, we use java and MySQL database server. Software configuration of each machine is shown in Table 5.

6.3. Conceptual model

We use *'Client-Server'* model to implement entire framework. For the implementation of *'Client-Server'* model, we have used java socket programming. In Socket programming, server process binds with an address and port number. Client process can communicate with server process with the help of this address and port number only. Port number range from 0 to 1023 are reserved and called well known port. We cannot allocate a well-known port to user process. The entire framework consist many processes. All processes are communicated with each other through sockets. Entire model is partitioned into five processes: (1) client, (2) manager, (3) global transaction manager (*GTM*), (4) database server, and (5) recovery manager.

Fig. 5 depicts Client-Server model of our prototype. This model describes communication between different processes using sockets. We are using multi-threaded server architecture, which implies for each client request a new server thread is allocated.

6.4. Implementation

With the help of various host machines, we have implemented the prototype of proposed framework. To evaluate correctness of the prototype, we started testing with standalone platform and scaled. The prototype contains following processes:

Client process: A client process can run any of host machines or many client processes can also run simultaneously. Client process represents a client node of framework. A client process submits user transaction and collects the result.

Manager process: Manager node of framework is represented by a manager process in the prototype. Prototype contains only one manager process which never goes down.

Global transaction manager (*GTM*) process: In our prototype, at max, three *GTM* processes can run simultaneously, each of which runs on different host machine. *GTM* processes provides scalability to our architecture.

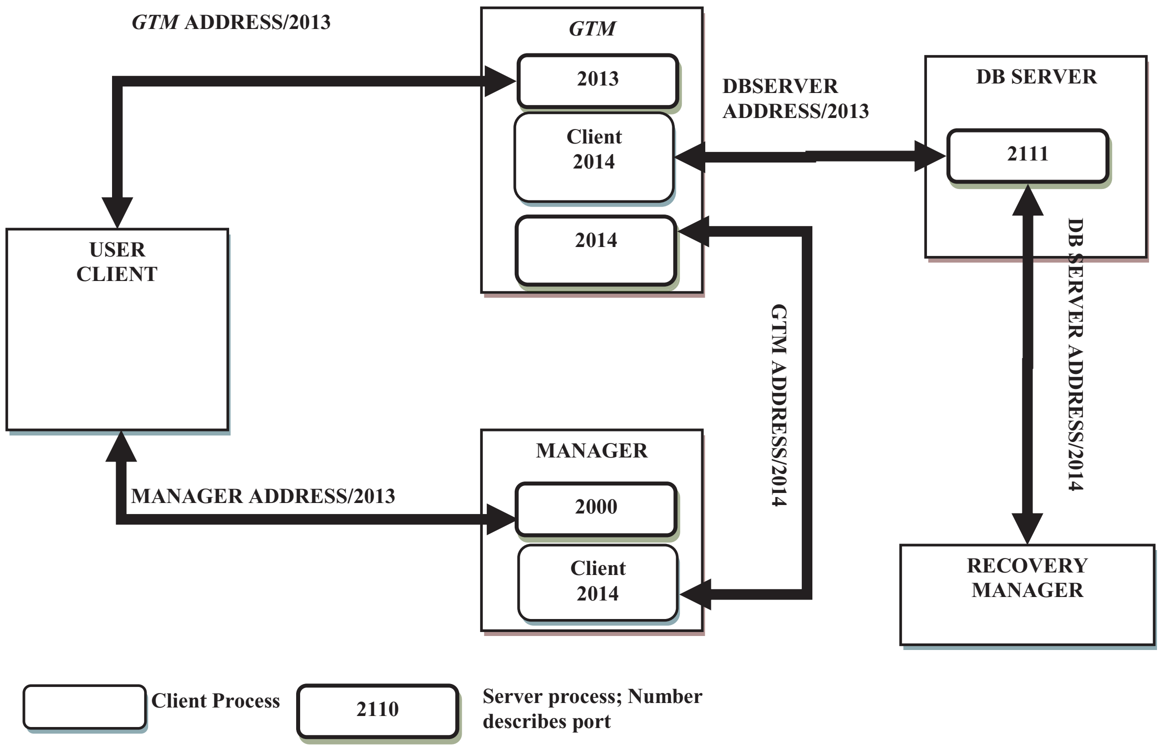

Fig. 5. Prototype for Client-Server model.

Database server process: To create a distributed database environment, we use bank database that is scattered among two host machines (Host 1 and Host 3). This bank database contains a table named *'accounts'*. A database server process runs at each database site (Host 1 and Host 3). A transaction can be performed by either all two host machines or any of one host machines.

7. Result analysis and scalability evaluations

7.1. Scalability test

Assumptions:

i. Each transaction lifetime is 30 seconds only.
ii. A GTM can serve 10 or more client requests at a time.
iii. Participated resource managers are super-computers.

Configuration of framework for this test is given below in Fig. 6. In this, we have considered the maximum global transaction managers as per the number of hosts for serializing the transaction requests and maintaining the integrity of the entire architecture. Given input with respect to the above framework is as follows:

Input: In this test we try to achieve scalability, with respect to client requests. A host (Host 1) submits 30 transaction requests simultaneously. These transaction requests are sent by interface client$_1$. Each transaction request accesses distinct rows of accounts table.

We have performed this scalability test with distinct cases. Where, case 1 represents the scenario of: one GTM is active with the transaction success rate as 10, case 2 represents the scenario of: two GTMs are active with the transaction success rate as 13, finally, case 3 represents the scenario of: three GTMs are active with the transaction success rate as 23. The corresponding output is shown in Table 6.

7.2. Atomicity test

Configuration: Prototype configuration for atomicity test is depicted in Fig. 7.

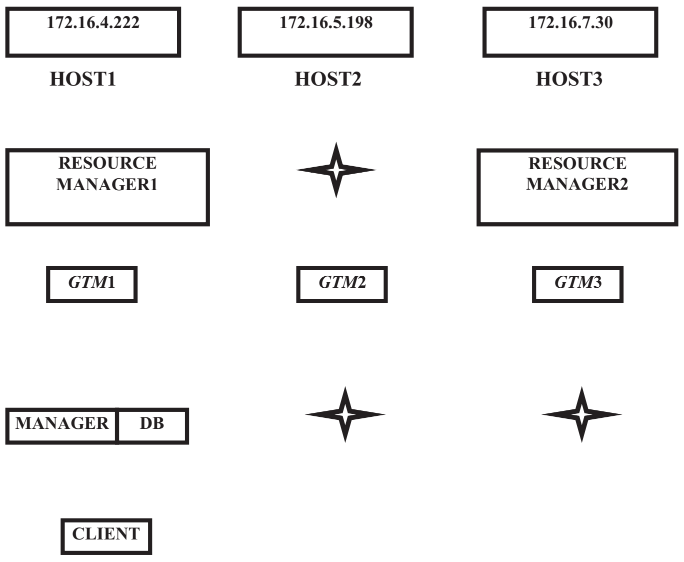

Fig. 6. Prototype configurations for scalability test.

Input: In this, a client requests (according to transaction) to *GTM* and waits for response. Transaction contains two database update operations (T$_1$ and T$_2$), which are distributed among resource manager$_1$ and resource manager$_2$. Transaction operations are given in Fig. 8.

Case 1: While executing a transaction, after sending 'ok' (yes) vote to *GTM*, resource manager$_1$ gets failed. Entire transaction scenario is shown in Fig. 9.

Result: The transaction is failed, due to resource manager$_2$ (RM$_2$) goes down after sending positive vote for *GTM*. Distributed database goes into inconsistent state. But global logs keep track this situation and recovery can be done by recovery manager as early.

Table 6
Output of scalability test in distinct scenario.

SL. no.	Case	Transaction success rate	Transaction in queue due to connec. problem	Transactions are in queue due to all *GTMs* are busy
1	One *GTM* is active	10	2	18
2	Two *GTMs* are active	13	7	10
3	Three *GTMs* are active	23	7	0

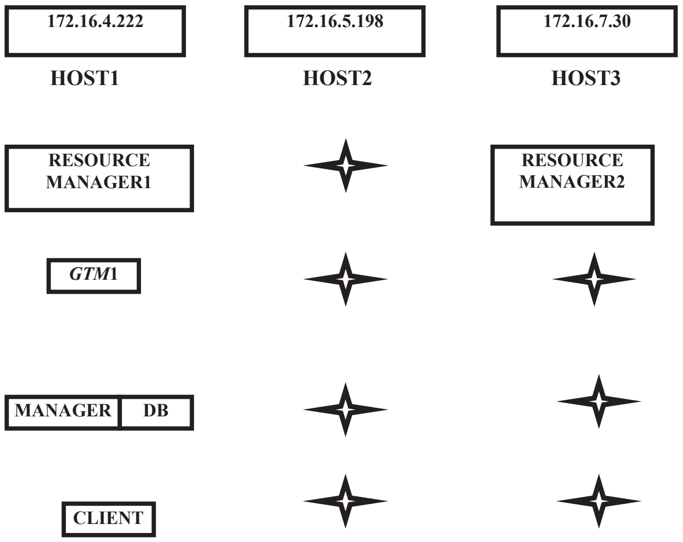

Fig. 7. Prototype configurations for atomicity test.

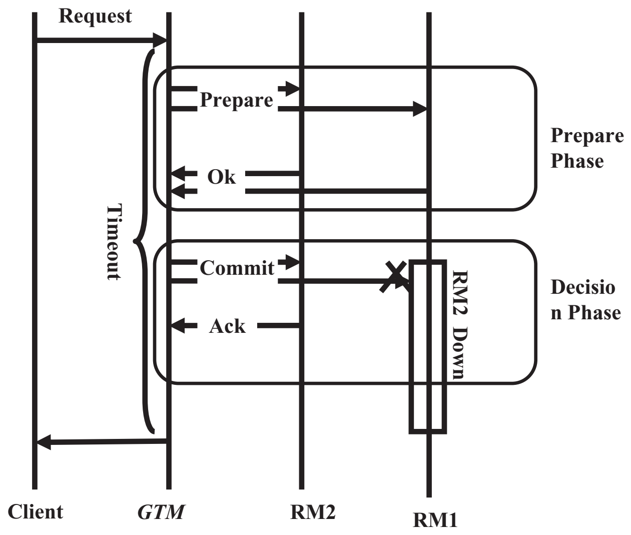

Fig. 9. Event driven sub-transaction scenario for case 1.

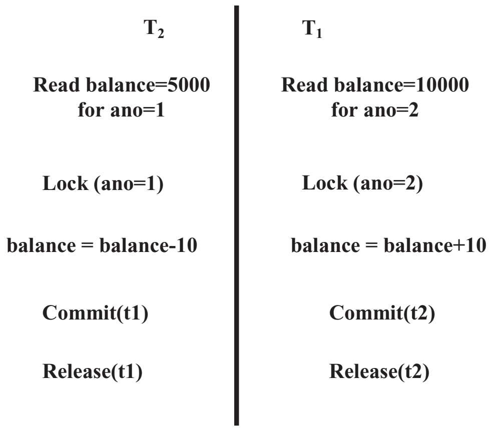

Fig. 8. Concurrent sub-transaction scenarios.

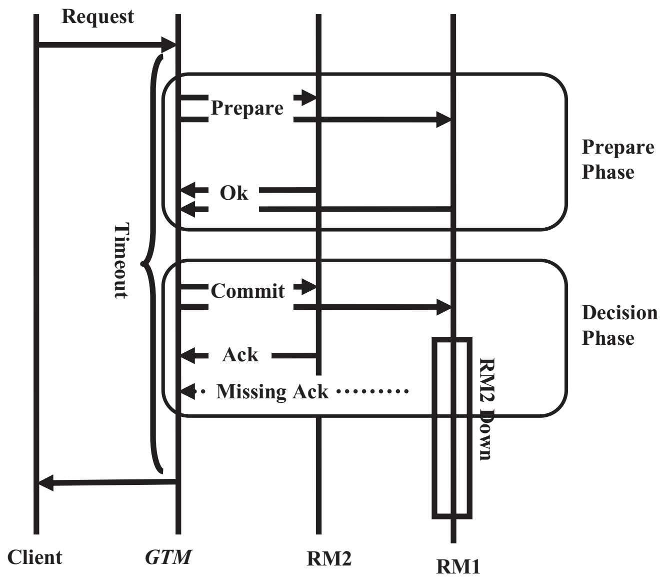

Fig. 10. Event driven sub-transaction scenario for case 2.

Case 2: After committing the transaction and before sending the '*Ack*' message to *GTM*, resource manager$_1$ goes down. Entire transaction scenario is given below in Fig. 10.

Result: Transaction is failed, due to resource manager$_2$ (RM$_2$) goes down after committing the transaction but before sending '*Ack*' message to *GTM*. Distributed database goes into inconsistent state. But, global logs keep track this situation and recovery can be done by recovery manager soon.

7.3. Transaction latency test

Latency is a measure of how long a transaction takes to complete. Latency is measured in time per transaction. Typical transaction latencies are measured in milliseconds (ms) per transaction. In this section, we perform latency test to find the results.

Case 1: Latency test in standalone environment.

Configuration: Prototype configuration for latency test in standalone environment is depicted in Fig. 11.

Input: In this test, we send client transaction requests and find out completion time of these transactions. Each transaction affects two database rows, and all consecutive transactions are concern with distinct row sets. Both rows belong to same database. We increase the client requests and verify the latency accordingly.

We have tested the number of transactions by increasing the count from 1 to 1000 and recorded the completion time in both best and worst cases. The completion time of both the cases is depicted in

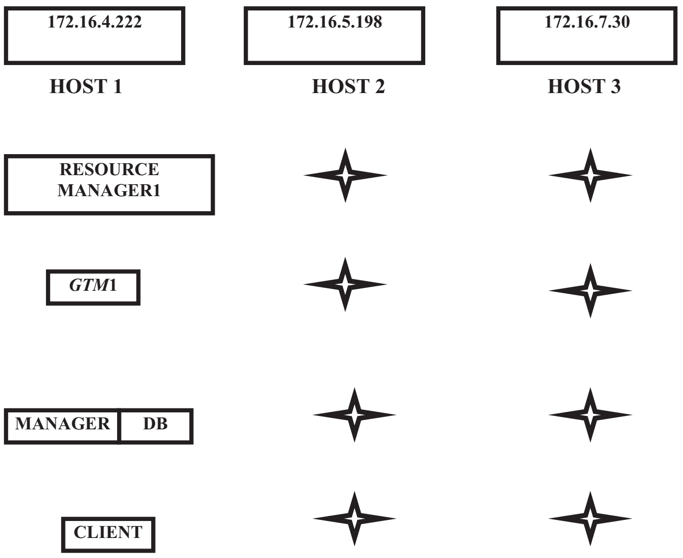

Fig. 11. Prototype configurations for latency test on standalone environment.

Table 7 as output. The completion time in worst case represents the execution of all 1000 transactions when the network speed is medium. On the other hand, completion time in best case represents the execution of all 1000 transactions when the network speed is high. The execution scenario of all transactions in standalone environment is depicted in Fig. 12.

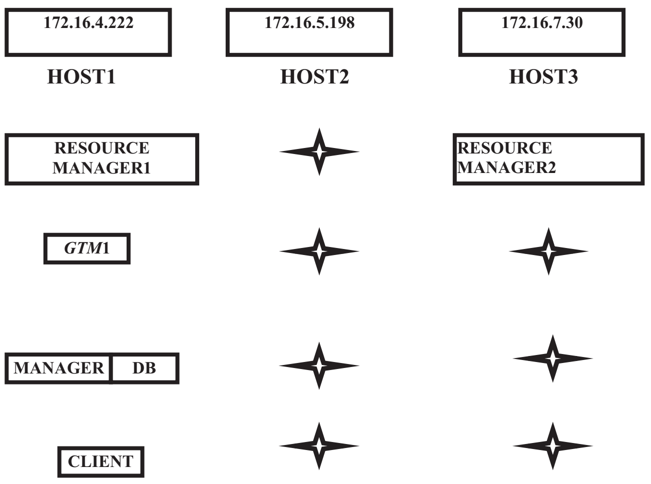

Fig. 13. Prototype configurations for latency test in distributed environment.

Output:

Case 2: Latency test in distributed environment.

Configuration: Prototype configuration for this test is depicted in Fig. 13.

Input: Same as the standalone environment, we submit client transaction requests and find out completion time of these transactions. Each transaction affects two database rows, and all consecutive transactions are concern with distinct row sets. But here, both rows

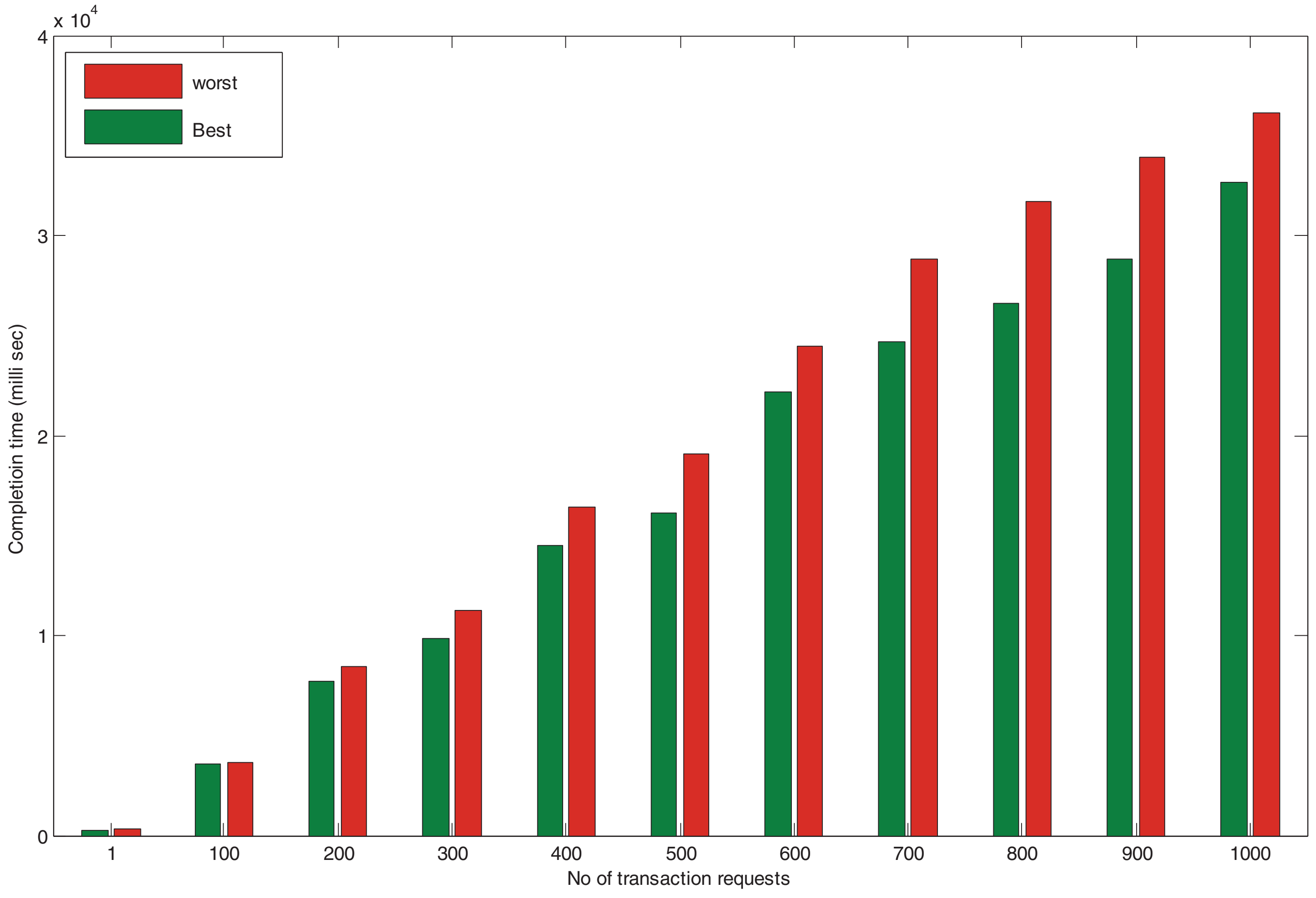

Fig. 12. Latency output in standalone environment.

Table 7
Latency test in standalone environment.

No. of transactions	Completion time (ms) (worst case)	Completion time (ms) (best case)
1	340.54880	239.61420
100	3665.59538	3599.00312
200	8425.03799	7695.47510
300	11262.30672	9858.63815
400	16422.92424	14522.68817
500	19085.67133	16160.09835
600	24485.82305	22190.18806
700	28799.65106	24682.56338
800	31668.09947	26612.64461
900	33926.35894	28850.53364
1000	36103.84988	32644.83922

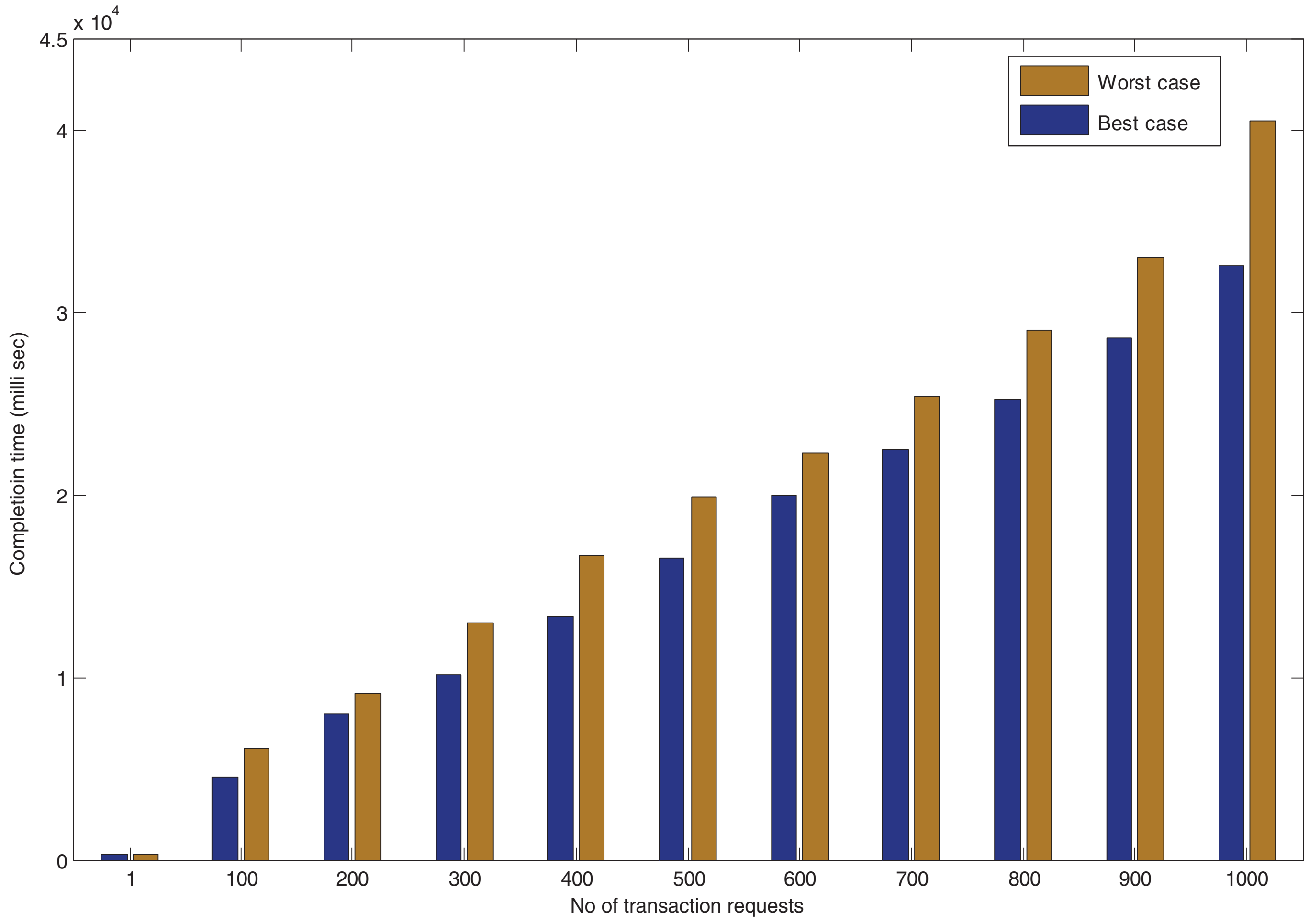

Fig. 14. Latency output in distributed environment.

are distributed among distinct databases. We increase client requests and check latencies accordingly.

We ran the number of transactions by increasing the count from 1 to 1000 and recorded the completion time in both best and worst cases as performed in standalone environment. There is no occurrence of unsuccessful transaction instance while recording the latency in best case. But, there are some unsuccessful transactions occurred while recording the latency in worst case. This is due to the channel error (i.e. network delay or packet loss) in the distributed environment. The completion time of both the cases is depicted in Table 8 as output. The completion time in worst case represents the execution of all 1000 transactions when the network speed is medium. On the other hand, completion time in best case represents the execution of all 1000 transactions when the network speed is high. The execution scenario of all transactions in distributed environment is depicted in Figs. 13 and 14 as output instances.

7.4. Scalability evaluations

In our evaluations of the proposed approaches the focus was on evaluating the following aspects: (1) the scalability of different approaches and (2) comparison of our model and the decentralized model (Berenson et al., 1995) and MAV (Peter et al., 2013) in terms of transaction throughput and scalability. During the initial phase of our work, we performed a preliminary evaluation of the proposed

Table 8
Latency test in distributed environment.

Number of	Latency in best case			Latency in worst case		
transactions	Completion time in ms	Number of unsuccessful transactions	Channel error (% of Pkt loss)	Completion time in ms	Number of unsuccessful transactions	Channel error (% of Pkt loss)
1	266.22041	0	0%	285.45720	0	0%
100	4532.48824	0	0%	6046.26482	0	0%
200	7951.83200	0	0%	9124.86423	0	0%
300	10154.46184	0	0%	12935.47721	10	12%
400	13284.96640	0	0%	16677.42130	0	42%
500	16520.84625	0	0%	19880.15584	27	37%
600	19986.12341	0	0%	22283.19954	0	0%
700	22499.99912	0	0%	25422.95822	86	42%
800	25200.22200	0	0%	29008.42200	0	0%
900	28555.50050	0	0%	33005.25510	0	0%
1000	32541.86620	0	0%	40510.85320	0	0%

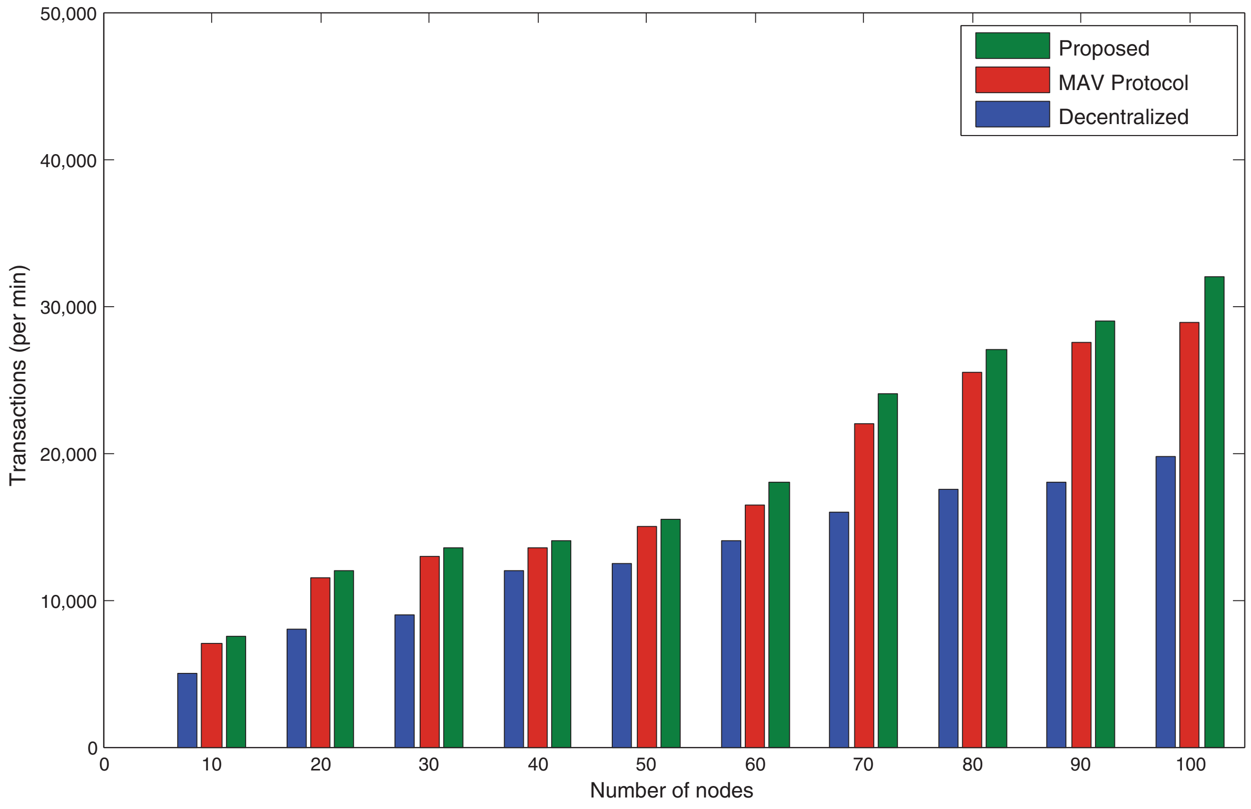

Fig. 15. Transaction throughput.

approaches to determine which approaches are more scalable and which of these need to be investigated further on a large scale system. This evaluation was done using a cluster of 16 nodes on which the number of cores on the cluster nodes varied from 3 to 6 cores, each with 3.2 GHz (and above), and the memory capacity ranged from 2 GB to 8 GB. The final evaluations in the later phase were conducted using a much larger cluster. Each node in this cluster had 8 CPU cores with 3.2 GHz capacity, and 24 GB main memory. We evaluate the scalability after performing all the tests.

7.4.1. Benchmark and preliminary evaluations

We used TPC-C benchmark to perform evaluations under a realistic workload. However, our implementation of the benchmark workload differs from TPC-C specifications in the following ways. Since our primary intension is to measure the transaction throughput we did not emulate terminal I/O. Since HBase does not support composite primary keys, we created the row-keys as concatenation of the specified primary keys. This eliminated the need of joining operations, typically required in SQL-based implementation of TPC-C. Predicate reads were implemented using scan and filtering operations provided by HBase. Since the transactions specified in TPC-C benchmark do not create serialization problems. In our experiments we observed that on average a TPC-C transaction performed 8 read operations and 6 write operations. During each experiment, the first phase involved loading data in HBase servers, which also ensured that the data was in the memory of HBase servers when the experiment started. We compared throughput and scalability of the transaction execution scenario with decentralized model architecture in Berenson et al. (1995) and monotonic atomic view (MAV) by Peter et al. (2013) which also represents HBase as NoSQL databases. Fig. 15 depicts the throughput scalability of 100 nodes, and Fig. 16 shows average response time in a 100 node environment. In these evaluations, we enabled the logging option for HBase to ensure that the writes are durable even under failures of HBase servers.

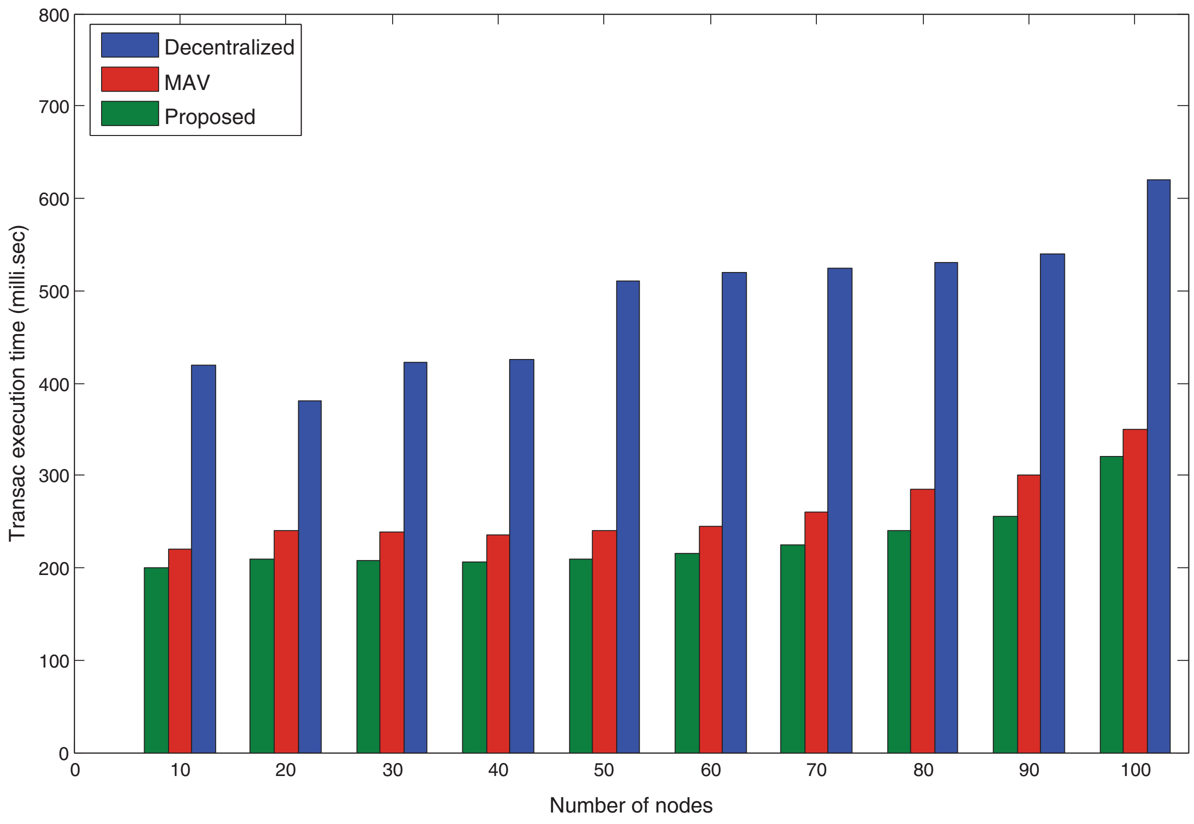

Fig. 16. Average transaction response times.

Table 9
Impact of transaction size.

Transaction size	Maximum throughput (Transactions/min)	Average response time (s)	% of aborts
10	5219	0.31	0.3
100	6988	1.69	12.2
1000	1080	5.2	41.7

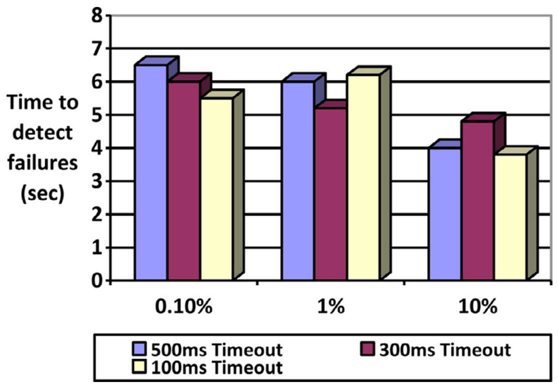

Fig. 17. Average time to detect failures.

7.4.2. *Impact of transaction size*

Another aspect that interested in evaluating is the impact of transaction size, i.e. the number of reads and writes in a transaction, on various performance measures. To analyze this, we created a custom benchmark as follows. We created a single table with 450 items. The benchmark included three classes of transactions: *small* size transactions accessing 10 items each, *medium* size transactions accessing 100 items each, and *large* size transactions accessing 333 items each.

The items to read and write were randomly selected based on uniform distribution. We performed separate evaluations for each class of transactions by generating transaction load for that class and measured the maximum throughput, average response times, and number of aborts for that transaction class. Table 9 shows the results of these evaluations.

7.4.3. *Fault tolerance evaluation results*

We induced a transaction load of approximately 50,000 transactions per minute. The insertion of faults was performed as follows. A transaction randomly stalls during the commit protocol execution with certain probability called as *failure probability*. The failure probability is calculated based on the desired failure rate. As expected, the percentage of false aborts increases as we decrease the timeout values. The percentage of valid timeout-based aborts depends on the failure rate. Note that, even though we decrease the timeout values from 500 ms to 100 ms, the percentage of total aborts increase only by approximately a factor of two (from 15% to 30%). This is because the transactions that do not conflict with any other transactions are unaffected by the timeout values. If a transaction does not conflict with any other concurrent transaction, it would not be aborted by any transaction irrespective of the timeout values. Fig. 17 shows the data related to average delays in detection of the failure of transactions that were failed.

8. Conclusions

In this paper, we have presented the scalability issue of our three-tier architecture to support atomic transaction across heterogeneous NoSQL databases. This architecture is reliable and flexible for managing the transactional instances. Labour nodes give a very high and infinitely scalable computation capability at very low cost. Manager nodes make this architecture available and also work as a cleaner to make it reliable. At last, use of MySQL cluster gives a way to store and access meta-data related to transaction without any single point failure. Zookeeper is used to configure all nodes of this three-tier architecture. In summary, our proposed model demonstrates that transactions can be supported in a scalable manner in NoSQL database systems.

Acknowledgements

The authors wish to express their gratitude and appreciation to the area editor and anonymous referees for many helpful suggestions that significantly improve this paper. This research work is supported by Indian School of Mines (ISM), Government of India under TEQIP – II project initiative. The authors would also like to express their gratitude and heartiest thanks to the Department of Computer Science & Engineering, Indian School of Mines (ISM), Dhanbad, India for providing their research support.

References

Abiteboul, S., Buneman, P., Suciu, D., 1999. Data on the Web, From Relations to Semistructured Data and XML. Morgan Kauffman, Los Altos.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P., 1995. A critique of ANSI SQL isolation levels, Proceedings of ACM SIGMOD' 95, ACM, pp. 1–10.

Bornea, M., Hodson, O., Elnikety, S., Fekete, A., 2011. One-copy serializability with snapshot isolation under the hood. In: IEEE ICDE'11, April 2011, pp. 625–636

Boutros, B.S., 1996. A two-phase commit protocol and its performance. In: IEEE 7th International Conference and Workshop on Database and Expert Systems 9–10 September 1996

Cahill, M.J., Röhm, U., Fekete, A.D., 2009. Serializable isolation for snapshot databases. ACM Trans. Database Syst. 34, 1–20:42.

Cattell, R., 2010. Scalable SQL and NoSQL datastores. SIGMOD Rec. 39 (4), 12–27.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R., 2006. Bigtable: a distributed storage system for structured data, OSDI. USENIX Association, pp. 205–218.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E., 2008. Bigtable: a distributed storage system for structured data. ACM Trans. Comp. Syst. (TOCS) 26 (2), 4.

Chin, W.-P., Simeon, N., 1992. The zookeeper route problem. Inform. Sci.: Int. J. 63 (3), 245–259.

Dadiomov, A., Raphael, R., Uri, H., 2003. Method and system for distributed transaction processing with asynchronous message delivery. U.S. Patent No. 6,529,932. 4 March 2003

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W., 2007. Dynamo: amazon's highly available key-value store. ACM SIGOPS Operating Systems Review, 41. ACM, pp. 205–220.

Dharavath, R., Jain, A.K., Kumar, C., Kumar, V., 2014. Accuracy of atomic transaction scenario for heterogeneous distributed column-oriented databases, Intelligent Computing, Networking, and Informatics. Springer India, pp. 491–501.

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D., 2005. Making snapshot isolation serializable. ACM Trans. Database Syst. 30, 492–528.

Frolund, S., Guerraoui, R., 2001. Implementing e-transactions with asynchronous replication. IEEE Trans. Parallel Distrib. Syst. 12 (2), 133–146.

Frolund, S., Guerraoui, R., 2002. e-Transactions: end-to-end reliability for three-tier architectures. IEEE Trans. Software Eng. 28 (4), 378–395.

Han, J., Haihong, E., Guan L., J. Du, 2011. Survey on NoSQL database, pervasive computing and applications (ICPCA), In: 2011 6th International Conference, pp. 363–366.

Jones, E.P.C., Abadi, D.J., Madden, S.R., 2010. Concurrency control for partitioned databases, In: SIGMOD, 2010

Jorwekar, S., Fekete, A., Ramamritham, K., Sudarshan, S., 2007. Automating the detection of snapshot isolation anomalies. In: VLDB 2007, pp. 1263–1274

Jung, H., Han, H., Fekete, A., Roehm, U., 2011. Serializable snapshot isolation for replicated databases in high-update scenarios. In: VLDB, 2011

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D.J., 2008. H-store: a high-performance, distributed main memory transaction processing system. Proc. VLDB Endowment 1 (2), 1496–1499.

Lakshman, A., Malik, P., 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 35–40.

Lars, G., 2011. Client API: The Basics in HBASE: The Definitive Guide. O'reilly Publishers.

Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxsburg, C., Byers, A.H., 2011. Big Data: The Next Frontier for Innovation, Competition, and Productivity. McKinsey Global Institute, pp. 1–20.

Membrey, P., Plugge, E., Hawkins, T., 2010. The Definitive Guide to MongoDB: the noSQL Database for Cloud and Desktop Computing. Apress.

Pacitti, E., Ozsu, M.T., Coulon, C., 2003. Preventive multi-master replication in a cluster of autonomous databases. In: Euro-Par

Peng, D., Dabek, F., 2010. Large-scale incremental processing using distributed transactions and notifications, In: USENIX OSDI'10, pp. 1–15

Peter, B., Aaron, D., Alan, F., Ali, G., Joseph, M., Hellerstein, I.S., 2013. Highly available transactions: virtues and limitations. PVLDB 7 (3), 181–192.

Ramesh, D., Chiranjeev, K., Amit Kumar, J., 2013. Atomicity and Snapshot Isolation on Column Oriented Databases – A Transaction Approach. Lap Lambert Academic Publishing, Germany.

Ramesh, D., Jain, A., Kumar, C., 2012a. Implementation of atomicity and snapshot isolation for multi-row transactions on column oriented distributed databases using rdbms. In: 2012 International Conference on Communications, Devices and Intelligent Systems (CODIS), pp. 298-301

Ramesh, D., Kumar, K.C., Ramji, B., 2012b. Design of a transaction recovery instance based on bi-directional ring election algorithm for crashed coordinator in distributed database systems, IEEE 2nd World Congress on Information and Communication Technologies (WICT -2012), pp. 721-726

Revilak, S., O'Neil, P., O'Neil, E., 2011. Precisely serializable snapshot isolation (PSSI), In: ICDE'11 pp. 482–493

Stonebraker, M., 2011. Stonebraker on NoSQL and enterprises. Commun. ACM 54, 10–11.

Stonebraker, M., Cattell, R., 2011. 10 rules for scalable performance in simple operation data stores. Commun. ACM 54 (6), 72–80.

Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P., 2007. The end of an architectural era: (it's time for a complete rewrite), Proceedings of the 33rd International Conference on Very Large Data Bases. VLDB Endowment, pp. 1150–1160.

Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J., 2012. Calvin: fast distributed transactions for partitioned database systems, Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, pp. 1–12.

Vora, M.N., 2011. Hadoop-HBase for large-scale data. International Conference on Computer Science and Network Technology (ICCSNT), 1. IEEE, pp. 601–605.

Whitney, A., Shasha, D., Apter, S., 1997. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In: HPTS

Zhang, C., Sterck, H.D., 2010. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones HBase. In: GRID, 2010, pp. 177–184

Dharavath Ramesh received the B.Tech degree from Kakatiya University, Warangal, India, in 2004 and M.Tech degree from the Jawaharlal Nehru Technological University, Hyderabad, India, in 2009. He is currently an assistant professor at the Department of Computer Science and Engineering, Indian School of Mines, Dhanbad, India. He works and publishes widely in the areas of Distributed databases, Database management systems, and Data mining. He has organized workshops, including tutorials, and actively served in the programming committees of international conferences. He is a member of the ACM and IEEE.

Chiranjeev Kumar received the M.E and Ph.D degrees from Motilal Nehru National Institute of Technology (MNNIT) and Allahabad University, Allahabad, UP, India. He is currently an associate professor at the Department of Computer Science and Engineering, Indian School of Mines, Dhanbad, India. He was a gold medallist of his M.E. batch at MNNIT, Allahabad in year 2001. His main research interests include Mobility Management in Wireless Networks, Ad Hoc Networks, Distributed Databases, and Software Engineering. He has contributed many research papers in several refereed journals and conference proceedings of national, and International reputes. He is also the guest editor of many international journals. He is a member of the IEEE and ACM.