Replication, Sharding and Consistency in MongoDB

Chandan Kumar Mishra
cmishra@sfu.ca

Abstract

In this experiment, I explored how replication and sharding features of MongoDB can be used for development of highly available and scalable application. Also, it is well known fact that MongoDB only support document level locking means single document atomicity and consistency can be claimed, so I tried an approach where transaction containing multi document can also be handled without losing atomicity and consistency using two phase commit concept [2].

1. Introduction

MongoDB is an open source document-oriented database system. It is part of the NoSQL family of database systems. It provides high performance, high availability, and easy scalability by compromising few of transactional concepts of relational database. In this experiment, I explored replication, sharding and a way to achieve multi document atomicity and consistency in MongoDB.

Replication is a basic technique to keep data safe (by providing redundancy) and highly available all the time by providing multiple instances serving the exact copy of the data. Replication helps a lot to recover from hardware failure and prevent service interruptions. Also, it is being used to off-load some work for example reporting or backup from the primary servers to secondary replicas.

Sharding (horizontal scaling) is the process of storing data records across multiple machines. It partitions data and store in different shards. Each shard can be either single Mongo DB instance or a group of MongoDB as a replica set. Each shard is an independent database, and collectively, the shards make up a single logical database. This whole process enhances read and write throughput of database.

MongoDB claim to support only document level atomicity and consistency. However, lot of real world applications involves multi-document transactions and require atomicity and consistency in such a scenario for example banking systems, booking systems etc. In order to achieve multi-document atomicity and consistency, we need to design an application in such a way which work on concept of two phase commit [2]. Using two-phase commit, we can ensure that data is consistent and, in case of an error, the state that preceded the transaction is recoverable. Here, two-phase commits can only offer transaction-like semantics, it is possible for application to read intermediate data at intermediate point of time, however in case of rollback and recovery, it is guaranteed to bring back in consistent state.

Rest of this report is organized, as Section 2 will contain overview followed by Section 3 Environment setup. In Section 4, analysis and result followed by conclusion in Section 5 and Section 6 will have references.

2. Overview

First, I will try to relate some basic components of NoSQL database to relational database.
Just like any relational database such as MS SQL Server, Oracle etc., MongoDB server can contain multiple databases. Each database has collections, which are

analogous to tables in relational databases. Each of these collections contains documents that are analogous to rows in relational databases. Each document contains fields that are relevant and actually resemble the objects of the application. Data in MongoDB is schemaless.
The implications of this is

- documents in the same collection may not necessarily have the same set of fields or structure
- common fields in a collection's documents may hold different types of data.

2.1 Replication

There are several categories of replication: **Master – Master** and **Master – Slave**. MongoDB support **Master – Slave** replication (only one node can accept write operations at a time) with automatic **master (**primary) election in case of failure.
There are some criteria, which we need to take care while setting up replica set in MongoDB

1. It support master slave model, where write will be only on primary, whereas read can be from either secondary or primary. By default, only primary is read enabled and manually we need to enable read from secondary using rs.serverOk() command.
2. As selection of primary in the case of failover is based on voting, so odd number of mongod instance in replica is must for having a fair voting. (Secondary can be arbiter also which need not contain any business data only participate in election)
3. Asynchronous update between primary and secondary, however using write concern properties of MongoDB, we can enable synchronous update between primary and secondary by compromising performance.
4. When a primary is not available to other members of the replica set for more than 10 seconds, the replica set will attempt to promote one of the secondary instances to become a new primary by starting the election process: the first secondary which receives a majority of the votes becomes primary.

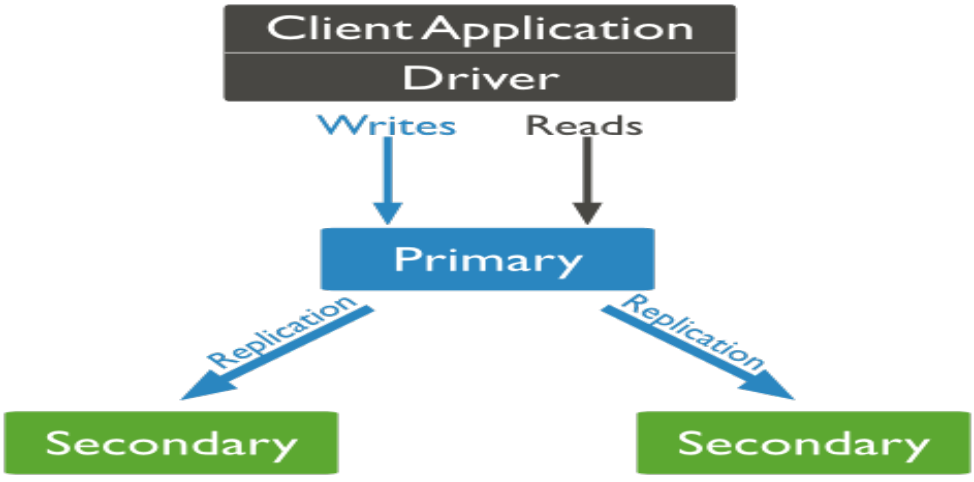

Figure 1: Basic Replication in MongoDB

2.2 Sharding

The basic building block of sharding are - Routers, Config Servers and Shards as shown in Figure 2.

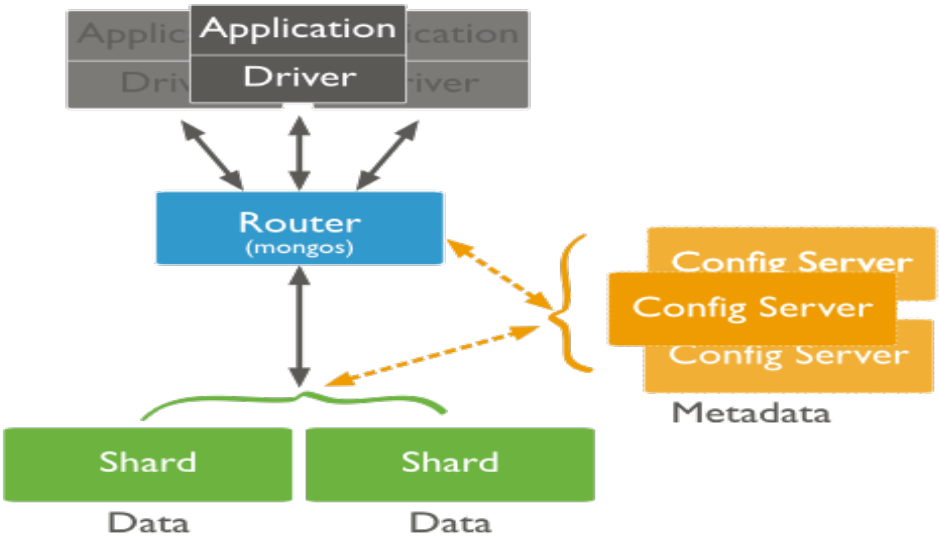

Fig 2 : Basic sharding architecture

Shard : It holds a subset of the collections data. It is either an instance of mongod (the MongoDB Server) or replication set.

Config Servers: It holds metadata of the cluster. It maps metadata to the chunks that the shard holds. These could be single or multiple instances of mongod. Config. server won't store any application specific collection data. The query router uses this metadata to target operations to specific shards.

Mongos instances (Routers): It routes the reads and writes to application transparently and to shards. It does not persist the data by itself.
Consider a database collection that is larger than the existing storage. This single collection is divided into chunks that are provided with a shard key as shown in figure 2. A process called balancer will distribute the chunks among the shards thus balancing the load. The Config. servers will hold the metadata of the shard key and the mapping to the corresponding shard. The monogos will route the reads and writes to appropriate shards by consulting Config. servers.

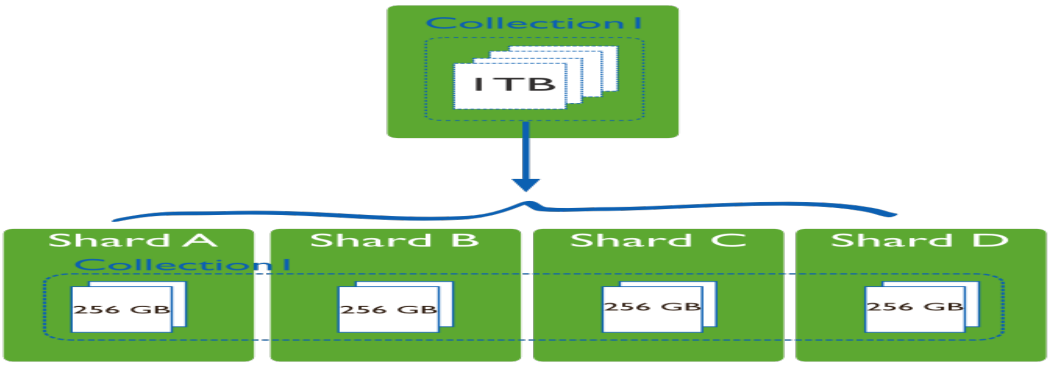

Figure 3 : Simple sharding example

Advantages of Sharding
- Automatic failover
- Auto balancing of load
- Facilitates additional write capacity

2.3 Multi - document atomicity and consistency in MongoDB

Up to very limited extent, MongoDB support ACID properties as explained below
- Atomicity: Ensured at single document level.
- Consistency: Eventually consistent reads, from a replica set are only possible with a write concern that permits reads from secondary members.
- Isolation: The multi update/write to multiple documents is not atomic and may interleave with other write operations. The isolation operator isolates the update/write operation and blocks other write operations during update.
- Durability:
    - MongoDB uses write ahead logging to an on-disk journal to guarantee write operation, durability and to provide crash resiliency.
    - before applying a change to the data files, MongoDB writes the change operation to the journal.
    - there is an up-to 100-millisecond window between journal commits where the write operation is not fully durable.
    - requiring journaled write concern in a replica set only requires a journal commit of the write operation to the primary of the set regardless of the level of replica acknowledged write concern.

Multi document atomicity and consistency is not supported in MongoDB, hence this experiment provides a pattern for doing multi-document updates or "multi-document transactions" using a two-phase commit approach for writing data to multiple documents and if it is successful in both document then only do confirm final commit. I explored this approach by taking a example of simple banking transaction system, which involves fund transfer from source to target. This two-phase commit design of application helps in rollback and recovery operation as well.

3. Environment Setup

This section will highlight steps, which I used for environment setup. I used MongoDB v3.2 and performed experiment in three independent module as listed below

1. Replication
2. Sharding
3. Multi-Document atomicity

3.1 Replication Enviornment

I setup replica set having one primary and two secondary, total of three instances in one replica set . To test replication, used a simple application and program to insert user record.

Sample script for setting up replica set :

```
mkdir -p /data/demo1 /data/demo2 /data/demo3
mongod --replSet rs1 --logpath "1.log" --dbpath /data/demo1 --port 27017 --oplogSize 64 --fork --smallfiles
```

```
mongod --replSet rs1 --logpath "2.log" --dbpath /data/demo2 --port 27018 --oplogSize 64 --smallfiles --fork
mongod --replSet rs1 --logpath "3.log" --dbpath /data/demo3 --port 27019 --oplogSize 64 --smallfiles --fork
```

3.2 Sharding Environment

For sharding setup, I made use of three shards having three mongod instance as a replica set in each shards, three config. servers and mongos router which is provided with installation of MongoDB. Total of twelve mongod and one mongos instance.

Sample bash script for setup:

```
# MongoDB
# script to start a sharded environment on localhost
# clean everything up
echo "killing mongod and mongos"
killall mongod
killall mongos
echo "removing data files"
rm -rf /data/config
rm -rf /data/shard*
# start a replica set and tell it that it will be shard0
echo "starting servers for shard 0"
mkdir -p /data/shard0/rs0 /data/shard0/rs1 /data/shard0/rs2
mongod --replSet s0 --logpath "s0-r0.log" --dbpath /data/shard0/rs0 --port 37017 --fork --shardsvr --smallfiles
mongod --replSet s0 --logpath "s0-r1.log" --dbpath /data/shard0/rs1 --port 37018 --fork --shardsvr --smallfiles
mongod --replSet s0 --logpath "s0-r2.log" --dbpath /data/shard0/rs2 --port 37019 --fork --shardsvr --smallfiles
sleep 5
# connect to one server and initiate the set
echo "Configuring s0 replica set"
mongo --port 37017 << 'EOF'
config = { _id: "s0", members:[
        { _id : 0, host : "localhost:37017" },
        { _id : 1, host : "localhost:37018" },
        { _id : 2, host : "localhost:37019" }]};
rs.initiate(config)
EOF
# start a replicate set and tell it that it will be a shard1
echo "starting servers for shard 1"
mkdir -p /data/shard1/rs0 /data/shard1/rs1 /data/shard1/rs2
mongod --replSet s1 --logpath "s1-r0.log" --dbpath /data/shard1/rs0 --port 47017 --fork --shardsvr --smallfiles
mongod --replSet s1 --logpath "s1-r1.log" --dbpath /data/shard1/rs1 --port 47018 --fork --shardsvr --smallfiles
mongod --replSet s1 --logpath "s1-r2.log" --dbpath /data/shard1/rs2 --port 47019 --fork --shardsvr --smallfiles
sleep 5
echo "Configuring s1 replica set"
mongo --port 47017 << 'EOF'
config = { _id: "s1", members:[
        { _id : 0, host : "localhost:47017" },
```

```
        { _id : 1, host : "localhost:47018" },
        { _id : 2, host : "localhost:47019" }]};
rs.initiate(config)
EOF
# start a replicate set and tell it that it will be a shard2
echo "starting servers for shard 2"
mkdir -p /data/shard2/rs0 /data/shard2/rs1 /data/shard2/rs2
mongod --replSet s2 --logpath "s2-r0.log" --dbpath /data/shard2/rs0 --port 57017 --
fork --shardsvr --smallfiles
mongod --replSet s2 --logpath "s2-r1.log" --dbpath /data/shard2/rs1 --port 57018 --
fork --shardsvr --smallfiles
mongod --replSet s2 --logpath "s2-r2.log" --dbpath /data/shard2/rs2 --port 57019 --
fork --shardsvr --smallfiles
sleep 5
echo "Configuring s2 replica set"
mongo --port 57017 << 'EOF'
config = { _id: "s2", members:[
        { _id : 0, host : "localhost:57017" },
        { _id : 1, host : "localhost:57018" },
        { _id : 2, host : "localhost:57019" }]};
rs.initiate(config)
EOF
# now start 3 config servers
echo "Starting config servers"
mkdir -p /data/config/config-a /data/config/config-b /data/config/config-c
mongod --logpath "cfg-a.log" --dbpath /data/config/config-a --port 57040 --fork --
configsvr --smallfiles
mongod --logpath "cfg-b.log" --dbpath /data/config/config-b --port 57041 --fork --
configsvr --smallfiles
mongod --logpath "cfg-c.log" --dbpath /data/config/config-c --port 57042 --fork --
configsvr --smallfiles
# now start the mongos on a standard port
mongos --logpath "mongos-1.log" --configdb
localhost:57040,localhost:57041,localhost:57042 --fork
echo "Waiting 60 seconds for the replica sets to fully come online"
sleep 60
echo "Connnecting to mongos and enabling sharding"
# add shards and enable sharding on the test db
mongo <<'EOF'
db.adminCommand( { addshard : "s0/"+"localhost:37017" } );
db.adminCommand( { addshard : "s1/"+"localhost:47017" } );
db.adminCommand( { addshard : "s2/"+"localhost:57017" } );
db.adminCommand({enableSharding: "demo"})
db.adminCommand({shardCollection: "demo.users", key: {user_id:1}});
EOF
```

Running above script will set up sharded environment with three shards, three confg servers and one mongos. For testing I used simple Java program to insert 10000 user record in user collection and performed find operation to analyse result and sharding process.

collection name : users
database name : demo
Shard_key : user_id

3.3 Multi-document atomicity

Consider a sample application for fund transfer between source and target involving multi-document transaction. In a relational database system, we can subtract the funds from source and add the funds to target in a single multi-statement transaction. In MongoDB, we can emulate a two-phase commit to achieve a comparable result.
This examples uses the following two collections:

1. A collection named accounts to store account information.
2. A collection named transactions to store information on the fund transfer transactions.

Steps :

1. Initialize source and target accounts with initial balance of 1000

```
db.accounts.insert(
  [
    { _id: "A", balance: 1000, pendingTransactions: [] },
    { _id: "B", balance: 1000, pendingTransactions: [] }
  ]
)
```

2. On fund transfer from source to target, initialize transactions record, with state = initial; possible state include pending, applied, done, cancelled, cancelling.

```
db.transactions.insert(
        { _id: 1, source: "A", destination: "B", value: 100, state: "initial", lastModified: new Date()
```

3. Fund transfer

3.1 Fetch record from transactions collection having state=”initial”
3.2 Update state=”pending”
3.3 Apply transaction to both accounts, subtract fund from source and apply fund to target. Also, push transaction _id in pendingTransactions of both source and target.
3.4 Update transaction state=”applied”
3.5 pull transaction _id from pendingTransactions of source and target.
3.6 Update transaction state =”done”

Above steps to perform fund transfer is quite useful to handle recovery and rollback features as discussed below.

NOTE: All individual record/operation will persist in collection after system crash as journal will be enabled “j”=true, so every intermediate step will be recorded in journal log and after system crash journal will put all collection’s operation to disk. Recovery step will fetch record from transactions collection and resume operation.

Recovery

Suppose system crash after transaction state updated to pending, on restart of system we can fetch pending record from transactions collection and start from step 3.2.
Similarly, possibility of failure at state=”applied” can be handled by resuming operation from step 3.5 onwards. In this way, system will be in consistent state after recovery.

Rollback
There are mainly two place where rollback possible; first transaction which is in "pending" state and other which is in "applied" state.
For "applied" state, need to reverse the transfer by creating transaction from target to source with same amount.
For "pending" state, remove pendingTransactions _id from source and target accounts and change this transaction state to "cancelled".
To demonstrate multi-document consistency, I used a Tomcat 7 server, Java and Spring framework.

4. Analysis and result

Before starting analysis, I will briefly explain few important concepts which we need to know about how updates propagates among master and slave of replica set. By default, replication between master-slaves is asynchronous and based on prolog (log maintained by master). When write happen at master (primary), it will also capture in its prolog. After gap of few second, changes captured in prolog will be applied to secondary.

This shows there will be gap in write between primary and secondary, which may cause stale read from secondary. However, this won't be problem for many applications, as after few seconds of time primary and secondary will come to eventual consistency.

Write Concern
Write concern describes the level of acknowledgement requested from MongoDB for write operations to a standalone mongod or to replica sets or to sharded clusters.
Write Concern basically contains three flags w, j, wtimeout.

"w"
Possible values are w = 1,w = 2, w = 3 or w = "majority" etc. For example, suppose a replica set having three mongod instance with w = 2 (Figure 3) implies application will get acknowledgement only after write operation performed on primary and one of secondary. Similarly for w=3 means, one primary and two secondary should acknowledge write. w =" majority" implies two out of three.

"j"
As already mentioned MongoDB uses Write ahead Logging (WAL) using journal.
The j option requests acknowledgement from MongoDB that the write operation has been written from journal to disk.
j = true; means it will write from journal to disk then only acknowledgement will be sent back to user or application.
j = false ; otherwise, however with this option we may lose data in case of system crash.

"wtimeout"
This option specifies a time limit, in milliseconds, for the write concern. wtimeout is only applicable for w values greater than 1.

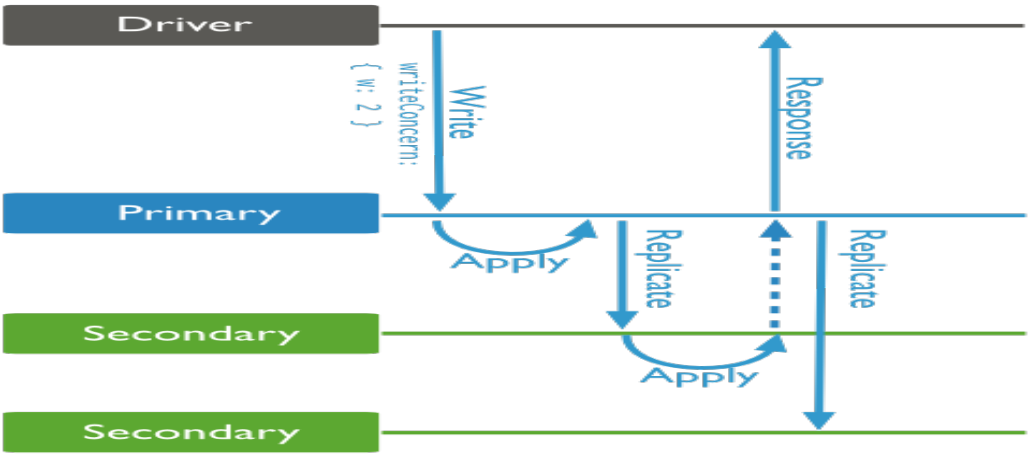

Figure 4: Replication with “w” = 2

Insertion of single user record with various combination of “w” and “j” shown in below table:

w	j	Time (in millisec)
1	false	862
1	true	932
2	true	1402
3	true	1815

Table 1 : Time taken for replication of single record

From Table 1, I can conclude only compromising on performance can do synchronous update. With w = 3 and j=true, write update will be applied synchronously on all three instances of replica by taking approx. twice time as compare to w=1 and j = true.

NOTE: Here, all three mongodb instances are running on same machine which is significantly reducing database to database communication time, however in real world when individual instance of replica will be on different machine communication time will also come into picture and will hit performance.

Sharding :

For testing sharding, I developed a simple Java program, which will insert 10000 user records.

Sample user document in demo collection

{

user_id:1, name:”TestName_1”, zipCode :50001

}

After insertion of 10000 users using simple java program, tried performing find operation and used explain command to debug how find uses sharding information from config. server

Command :
sh.status()

databases:
{ "_id" : "demo", "primary" : "s2", "partitioned" : true }
demo.users
shard key: { "user_id" : 1 }
unique: false
balancing: true
chunks:
s0 1
s1 1
s2 1
{ "user_id" : { "$minKey" : 1 } } -->> { "user_id" : 2 } on : s1 Timestamp(3, 0)
{ "user_id" : 2 } -->> { "user_id" : 16 } on : s2 Timestamp(3, 1)
{ "user_id" : 16 } -->> { "user_id" : { "$maxKey" : 1 } } on : s0 Timestamp(2, 0)

db.users.find().explain() -> it shows all three shards participated in find operation
db.users.find({user_id:1}).explain() -> it will only require to look into shard 1.
db.users.find({user_id:10}).explain() -> it will only require to look into shard 2.
db.users.find({user_id:50}).explain() -> it will only require to look into shard 0.

In above described shared environment each shared_key (user_id) is mapped to particular shard, mongos router will fetch that information from config. server and redirect any request to that specific shard. Hence, it is increasing overall read and write throughput of database.

Multi-Document atomicity:
Developed a sample banking application covering scenario explained in section 3.3. It is developed using Java spring web framework, MongoDB v3.2 and running on Tomcat server v7.0. Tested all scenarios using various options (button) available in below UI.

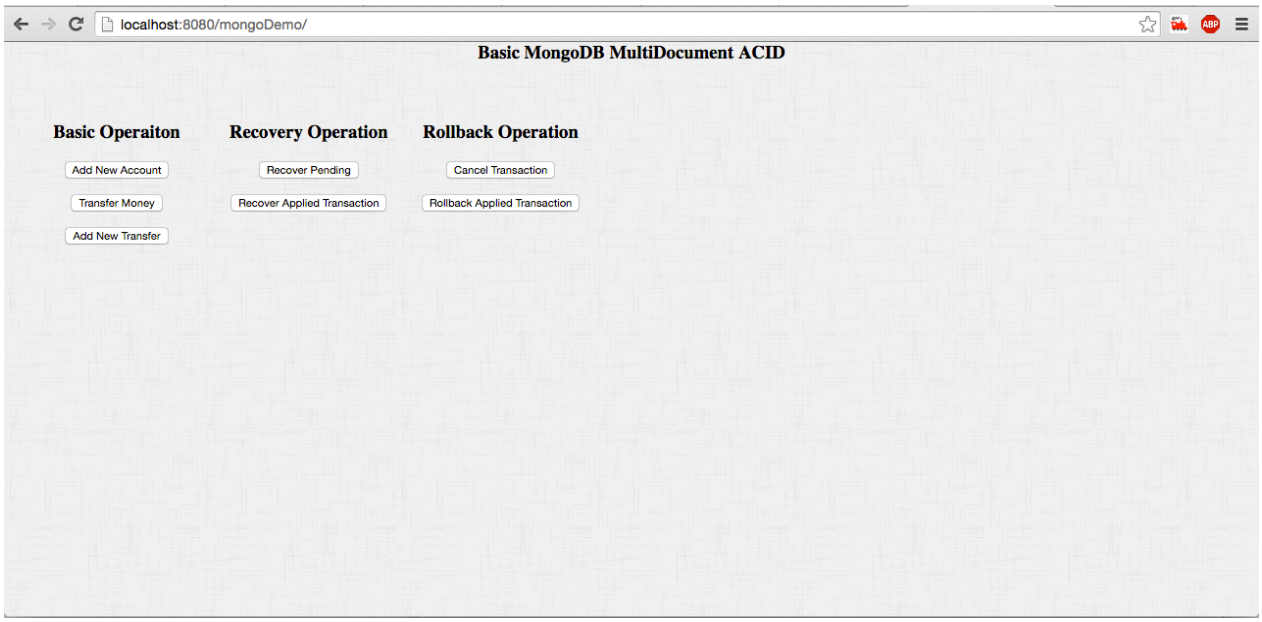

Figure 5 : UI for multi-document atomicity test

5. Conclusion

MongoDB is quite good for replication and sharding because of its ease to use and well documented manual. Setting up cluster environment is also very easy as compare to traditional relational database because of lightweight mongod instances. Also, sharding and replication almost work together in parallel to enhance database read and write throughput. Using write and read concern, we can adjust durability and performance as per need of application.

However, application containing transactions need extra care in designing phase itself. For handling a transaction, application has to design in such a way that either it does not involve multi-document transaction or if it involves than it should be handled as shown in section 3.3 two phase commit design approach.

6. References

[1] Official MongoDB 3.2 Documentation https://docs.mongodb.org/manual

[2] https://docs.mongodb.org/manual/tutorial/perform-two-phase-commits

[3] http://stackoverflow.com/questions/7149890/what-does-mongodb-not-being-acid-compliant-really=-mean

[4] Database slides from CMPT 740 course