# Sharded-Key-Value-Store

Github Link: https://github.com/yzlucas/Sharded-Key-Value-Service

# Abstract

A key-value database is a type of non-relational database that uses a simple key-value method to store data. A key-value database stores data as a collection of key-value pairs in which a key serves as a unique identifier. Both keys and values can be anything, ranging from simple objects to complex compound objects. Key-value databases are highly partitionable and allow horizontal scaling at scales that other types of databases cannot achieve. For example, Amazon DynamoDB allocates additional partitions to a table if an existing partition fills to capacity and more storage space is required.

Amazon DynamoDB and Bigtable/HBase (Google) are two examples that apply key/value service.

Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It's a fully managed, multiregion, multimaster, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications. Also, Cloud Bigtable is a sparsely populated table that can scale to billions of rows and thousands of columns, enabling you to store terabytes or even petabytes of data. A single value in each row is indexed; this value is known as the row key. Cloud Bigtable is ideal for storing very large amounts of single-keyed data with very low latency. It supports high read and write throughput at low latency, and it is an ideal data source for MapReduce operations.

# Introduction

This project mainly focuses on completing and analyzing the lab4 from mit 6.824 -- Sharded Key/Value Service. This project is a Linux system based environment and implements a Sharded key/value database service, where each shard replicates its state based on Paxos algorithm, which handles network partitions and is fault tolerant, scalability, and availability. The project is to build a key/value storage system that "shards," or partitions, the keys over a set of replica groups. A shard is a subset of the key/value pairs; for example, all the keys starting with "a" might be one shard, all the keys starting with "b" another, etc. The reason for sharding is performance. Each replica group handles puts and gets for just a few of the shards, and the groups operate in parallel; thus total system throughput (puts and gets per unit time) increases in proportion to the number of groups.

There are two main components: shard master and shard group. The entire system has a master and multiple groups. The master is a raft cluster, and each shard group is a cluster composed of kvraft instances. The shard master is responsible for scheduling. The client sends a request to the shard master. The master will tell the client which group is serving this key according to the configuration (config). Each group is responsible for part of the shard.

For each group, we need to realize the shard data transfer between the groups. Because our group changes dynamically, some are newly added, and some have to be withdrawn. Sometimes load balancing needs to reconfigure the shards of each group.

The sharded key/value database, where each shard replicates its state using Paxos. This key/value service can perform Put/Get operations in parallel on different shards, allowing it to support applications such as MapReduce that can put a high load on a storage system. Also, it has a replicated configuration service, which tells the shards for what key range they are responsible. It can change the assignment of keys to shards, for example, in response to changing load.

We will show how we complete part a (the shardmaster) and part b (shard key/value server) in the next part.

# Implement / related work / experimental methodology

We implement the part a (the shard master) in eight main steps:

## 1. Read the document:

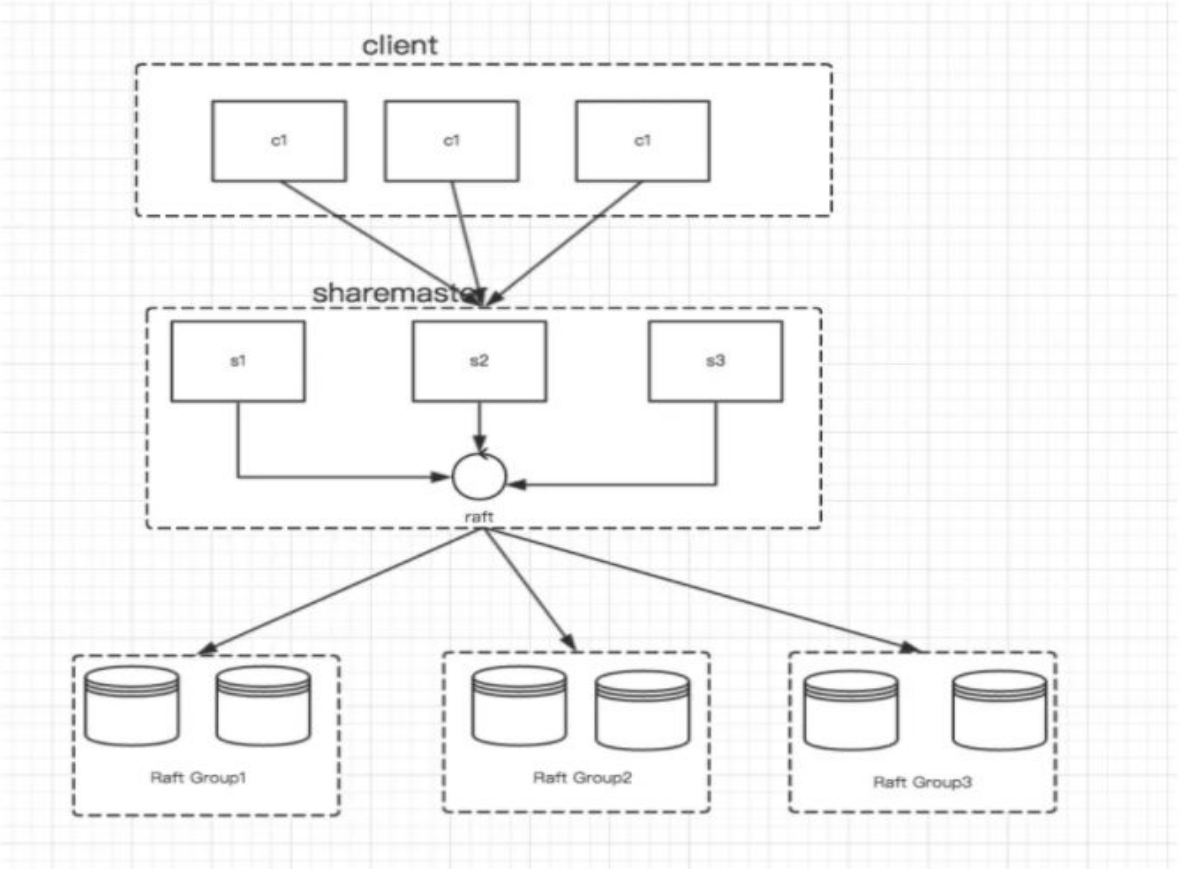
https://pdos.csail.mit.edu/6.824/labs/lab-shard.html
We read everything before 4B and understand the meaning of each paragraph.

## 2. Understand the structure:

It is a classic M/S structure: a configuration service and a set of replica groups. However, it is fairly basic, some functions are not implemented: (1) The transmit between shards is slow and concurrent client access is not allowed. (2) the member in each raft group will never be changed.

**Configuration service:**
1, Use the raft agreement to maintain the consistency for shardmasters.
2, Manage the order of configurations: The configuration describes the replica group; the group store shares.
3, Respond to the Join/Leave/More/Query request, and make related changes to configuration.

**Replica group:**
1, Use the raft agreement to maintain the consistency for shardkv.
2, Storing a part of specific data, combining all data in each group which is data in the whole database.
3, Respond to the Get/PurAppend request, and make sure it is linearized.
4, Query shardmasters periodically to get configuration, and do the migration and update.




Sharemaster consists of many machines, they use the agreement of the raft to maintain consistency. It stores data in different replica groups according to the related client.

Each replica group consists of many machines, they use the agreement of the raft to maintain consistency.

## 3. Understand the structure of code

The client part is the same pattern as 3A (which we already covered in the previous lab)
We need to implement the server part.

First, we need to understand the common part.

```go
// The number of shards.
const NShards = 10

// A configuration -- an assignment of shards to groups.
// Please don't change this.
type Config struct {
    Num    int              // config number
    Shards [NShards]int     // shard -> gid
    Groups map[int][]string // gid -> servers[]
}
```

The config struct contains the config number, shard -- gid and gid -- servers.
Then the master server contains a group of config (number increased).

```go
configs []Config // indexed by config num
```

The next thing is four API:

JOIN: It gives a group of maps: GID → SERVER. It actually adds the group of gid to the area that is managed by MASTER. Then we have the new GROUP, each machine could give a part of it to SHARD

LEAVE: It gives a group of GID, which indicates the related SERVER machine will leave. Then SHARDs that they managed will give to the GROUP that has not left yet.

QUERY: It finds the related SHARD's rule of CONFIG according to the CONFIG NUM.

The master side is actually similar to the lab 3a. The mean difference is "REBALANCE". (Make the GROUP increase/decrease) to make SHARD even be placed.

**Implement Client side:**
Based on the hint, the strategy is similar to 3A.

*"Start with a stripped-down copy of your kvraft server.*

*You should implement duplicate client request detection for RPCs to the shard master. The shardmaster tests don't test this, but the shardkv tests will later use your shardmaster on an unreliable network; you may have trouble passing the shardkv tests if your shardmaster doesn't filter out duplicate RPCs."*

The codes of client.go and common.go could be checked at github.

**Construct the Server structure:**

The code is basically the same as KV SERVER.

The code could be checked at github.

Importantly, two things are different compared to KV SERVER. In KV server, we can simply update STRING STRING MAP. However, in SM SERVER, we need to do something different. Therefore, we encapsulate it in updateConfig. We need to REBALANCE the JOIN and LEAVE in it. Also, the QUERY part is different: if successfully returned, we need to GET CONFIG and send it back.

**Implement QUERY HANDLER:**

As the hint says:

*"The Query RPC's argument is a configuration number. The shardmaster replies with the configuration that has that number. If the number is -1 or bigger than the biggest known configuration number, the shardmaster should reply with the latest configuration. The result of Query(-1) should reflect every Join, Leave, or Move RPC that the shardmaster finished handling before it received the Query(-1) RPC."*

When QUERY, if it is not -1 or not greater than LEN(CONFIG), we get the CONFIG of LEN -1. It is the same as the GET of the KV SERVER, we use the raft to maintain consistency.

```
func (sm *ShardMaster) Query(args *QueryArgs, reply *QueryReply) {
    reply.WrongLeader = true;
    originOp := Op{ OpType: "Query", Args: args.copy(), Cid: Nrand(), SeqNum: -1}
    reply.WrongLeader = sm.templateHandler(originOp)
    if !reply.WrongLeader {
        sm.mu.Lock()
        defer sm.mu.Unlock()
        if args.Num >= 0 && args.Num < len(sm.configs) {
            reply.Config = sm.configs[args.Num]
        } else {
            reply.Config = sm.configs[len(sm.configs) - 1]
        }
    }
}
```

## 4. Implement the structure of updateConfig:

Only MOVE, JOIN and LEAVE need to be UPDATED CONFIG.

The UPDATE means: Make a copy based on the most recent CONFIG and update it in the copy.

The MOVE is simple, only needs to update Shards in CONFIG.

Then JOIN is to put the new mapping of GID→ server to the new CONFIG.

LEAVE is get a group of GID from the new CONFIG.

```
func (sm *ShardMaster) updateConfig(op string, arg interface{}) {
    cfg := sm.createNextConfig()
    if op == "Move" {
        moveArg := arg.(MoveArgs)
        if _,exists := cfg.Groups[moveArg.GID]; exists {
            cfg.Shards[moveArg.Shard] = moveArg.GID
        } else {return}
    }else if op == "Join" {
        joinArg := arg.(JoinArgs)
        for gid,servers := range joinArg.Servers {
            newServers := make([]string, len(servers))
            copy(newServers, servers)
            cfg.Groups[gid] = newServers
        }
    } else if op == "Leave"{
        leaveArg := arg.(LeaveArgs)
        for _,gid := range leaveArg.GIDs {
            delete(cfg.Groups,gid)
        }
    } else {
        log.Fatal( v...: "invalid area",op)
    }
    sm.configs = append(sm.configs,cfg)
}
```

## 5. Implement rebalance

First, we need to calculate the number of shards that each replica group will be allocated. For example 10, 4 groups, it is 3,3,2,2. Then we put shards from the group (contain more shards) to the group(contain less shards). For example, 4,3,3; there is a new group, we get 1 from “4”, 3,3,3,1, we can 1 from “3”, it will be 3,3,2,2.

If the origin situation is 4,3,3 now it turns to 2 groups. Then we traverse Shards[NShards] int // shard → gid, we will see which group disappears, and add related SHARD to the group that contains less SHARD.

The above two strategies are hard to write in one function. We need to write them separately.

Follow the above pattern, the time-complexity is moveElements*gidNumber. To implement our idea, we calculate the previous CONFIG, the number of SHARD in each GID. Then JOIN: we find the group that most SHARD and move them to the new group until they reach to the average value. LEAVE: we find the smallest and give the LEAVE to the smallest one. Then we find the new smallest one until the LEAVE's GROUP does not contain SHARD.

The code could be checked at github.

We implemented 4b in 21 steps:

## 6. Read the document

Understand each paragraph.

Base on the lab3, we could pass the test with single group:

*"Your first task is to pass the very first shardkv test. In this test, there is only a single assignment of shards, so your code should be very similar to that of your Lab 3 server. The biggest modification will be to have your server detect when a configuration happens and start accepting requests whose keys match shards that it now owns."*

## 7. Increase CONFIG, and refuse SHARD that does not belong to itself:

We could do this from hint:

*"Add code to server.go to periodically fetch the latest configuration from the shardmaster, and add code to reject client requests if the receiving group isn't responsible for the client's key's shard. You should still pass the first test."*

The code could be checked in github.

## 8. Think:

We need to think two things: how to move SHARD after CONFIG changed and the rpc for transferring SHARD.

**How to move SHARD after CONFIG changed:**

If a REPLICA GROUP A gets a SHARD 1, corresponding B loses a SHARD 1. If A detects too much, it will be more passive to wait for someone to send it to me. Because we don't know how long to wait. Secondly, B also needs to take the initiative to send it to A after discovering that he has lost SHARD 1, which increases B's workload. Because when we are doing SHARD MIGRATION here, we cannot respond to requests. But for B, he can update CONFIG immediately, even if SHARD 1 is not sent out, he can still respond to the request.But for A, it must get SHARD 1 before it can continue to serve.

Based on the above thinking, we decided to let A ask B for SHARD, which would simplify the design. Because of this, if B finds a new CONFIG, it can be directly updated to make it effective immediately. A needs to wait for the PULL to succeed, then update CONFIG to make it take effect.

Reconfiguration will affect PutAppend/Get, so it is also necessary to use raft to ensure consistency within the group to ensure that reconfiguration is performed at the same time after the previous operations are completed in the cluster.

**The rpc for transferring SHARD:**

First of all, SHARD DATA must be sent.But just sending SHARD DATA is not enough. For example, when an APPEND REQUEST is sent to A SERVER, it is TIMEOUT. At this time, the A server has already done this update operation. After this point, Reconfiguration occurs, and CLIENT asks B SERVER to send APPEND REQ. If only SHARD DATA passed and it will cause APPEND2 times.Therefore, we also need to send the deduplicated MAP together. In addition to telling which SHARD we want, the parameters sent in the past also need to add a CONFIG NUM, because it is possible that the CONFIG NUM we sent is larger than that, indicating that the CONFIG over there has not been synchronized. Based on the above ideas. The RPC we designed is as follows.

```go
type MigrateArgs struct {
    Shard     int
    ConfigNum int
}

type MigrateReply struct {
    WrongLeader bool
    Err         Err
    DB          map[string]string
    Cid2Seq     map[int64]int
}
```

## 9. Implement MIGRATE SHARD RPC HANDLER

Each RAFT GROUP is responsible for sending and receiving RPC by LEADER. FOLLOWER is only responsible for going from APPLY MSG and LEADER SYNC state.

Also, we cannot directly fetch data from the DB. If we do not achieve the premise of cleaning the data, because the data is not cleaned. So we have a lot of data, imagine. We accept SHARD1 first, then we don't accept it, and then we accept SHARD1 again. At this time, the migration will be a union. And we just hope it is the part that is re-accepted. Based on the above considerations. We need to extract the data to be migrated separately based on each CONFIG. Do migration according to CONFIG in this way.

```go
func (kv *ShardKV) ShardMigration(args *MigrateArgs, reply *MigrateReply) {
    reply.WrongLeader, reply.Err, reply.Shard, reply.ConfigNum = true, OK, args.Shard, args.ConfigNum
    _,isLeader := kv.rf.GetState()
    if !isLeader {return}
    kv.mu.Lock()
    defer kv.mu.Unlock()
    if args.ConfigNum > kv.cfg.Num {return}
    reply.WrongLeader = false
    reply.DB, reply.Cid2Seq = kv.deepCopyDBAndDedupMap(args.ConfigNum,args.Shard)
}
func (kv *ShardKV) deepCopyDBAndDedupMap(config int,shard int) (map[string]string, map[int64]int) {
    db2 := make(map[string]string)
    cid2Seq2 := make(map[int64]int)
    for k, v := range kv.toOutShards[config][shard] {
        db2[k] = v
    }
    for k, v := range kv.cid2Seq {
        cid2Seq2[k] = v
    }
    return db2, cid2Seq2
}
```

## 10. How to pull data

If we choose to let LEADER interact, we must fail the HANDLER RAFT Leader, and a new LEADER must be responsible for PULL DATA. Therefore, it must be stored on all nodes to ask where to go to PULL DATA. If PULL arrives, we need to make sure that LEADER will send CMD to RAFT (this CMD is for nodes to synchronize data, and at the same time delete the place where the maintenance goes to PULL DATA). And we must open an additional background process and loop to do this. Otherwise, there will be no one to PULL DATA after the LEADER is transferred. Because of PULL DATA, there is no CLIENT timeout to retry.

Because we need to loop to PULL DATA in the background, we get the DATA, send it to RAFT, and then enter the APPLY CH, all nodes need to be able to synchronize this data. Once the synchronization is successful, we need to clean up this waiting data. In this way, background threads can save a lot of useless RPC. At the same time, we need to know the target REPLICA GROUP when we ask for data.

Now, we have already put the new received CONFIG and MIGRATION DATA to RAFT'log to do the linearly sortion in the leader side.

Therefore, when the two messages are sent from APPLY MSG, we need to do something. We write a function for Apply.

```
func (kv *ShardKV) apply(applyMsg raft.ApplyMsg) {
    if cfg, ok := applyMsg.Command.(shardmaster.Config); ok {
        kv.updateInAndOutDataShard(cfg)
    } else if migrationData, ok := applyMsg.Command.(MigrateReply); ok{

    }else {
        op := applyMsg.Command.(Op)
        kv.mu.Lock()
        maxSeq,found := kv.cid2Seq[op.Cid]
        if !found || op.SeqNum > maxSeq {
            switch op.OpType {
            case "Put":
                kv.db[op.Key] = op.Value
            case "Append":
                kv.db[op.Key] += op.Value
            }
            kv.cid2Seq[op.Cid] = op.SeqNum
        }
        kv.mu.Unlock()
        if kv.needSnapShot() {
            go kv.doSnapShot(applyMsg.CommandIndex)
        }
        if notifyCh := kv.put(applyMsg.CommandIndex, createIfNotExists: false); notifyCh != nil {
            send(notifyCh,op)
        }
    }
}
```

## 11. Implement APPLY MAS is MIGRATION DATA REPLY:

Here we are receiving the CONFIG change, we will refresh the CONFIG. But after the CONFIG is refreshed, we will update COME IN SHARD, and then the background thread will go to PULL. From the update of COME IN SHARD to the arrival of data SHARD, during this period, we must reject all requests for the SHARD. So we can't judge whether it is WRONG GROUP directly from CONFIG. At this point, we need to maintain an additional data structure of which SHARD I can HANDLER.

```
if migrationData.ConfigNum != kv.cfg.Num-1 {return}
delete(kv.comeInShards, migrationData.Shard)
//this check is necessary, to avoid use  kv.cfg.Num-1 to update kv.cfg.Num's shard
if _, ok := kv.myShards[migrationData.Shard]; !ok {
    kv.myShards[migrationData.Shard] = true
    for k, v := range migrationData.DB {
        kv.db[k] = v
    }
    for k, v := range migrationData.Cid2Seq {
        kv.cid2Seq[k] = v
    }
```

Then we could delete the sended SHARD and add it when SHARD comes.

```
toOutShards     map[int]map[int]map[string]string "cfg num -> (shard -> db)"
comeInShards    map[int]int "shard->config number"
myShards        map[int]bool    "to record which shard i can offer service"
```

## 12. Implement UpdateInAndOutDataShard

We could decide which data we want to send and come according to the new CONFIG.

```go
func (kv *ShardKV) updateInAndOutDataShard(cfg shardmaster.Config) {
        if cfg.Num <= kv.cfg.Num { //only consider newer config
            return
        }
        oldCfg, toOutShard := kv.cfg, kv.myShards
        kv.myShards, kv.cfg = make(map[int]bool), cfg
        for shard, gid := range cfg.Shards {
            if gid != kv.gid {continue}
            if _, ok := kv.myShards[shard]; ok || oldCfg.Num == 0 {
                kv.myShards[shard] = true
                delete(toOutShard, shard)
            } else {
                kv.comeInShards[shard] = oldCfg.Num
            }
        }
        if len(toOutShard) > 0 { // prepare data that needed migration
            for shard := range toOutShard {
                outDb := make(map[string]string)
                for k, v := range kv.db {
                    if key2shard(k) == shard {
                        outDb[k] = v
                        delete(kv.db, k)
                    }
                }
                kv.toOutShards[oldCfg.Num][shard] = outDb
            }
        }
}
```

## 13. Decide WRONG GROUP:

In the previous version, when we received the request on the SERVER side, we directly judged the WRONG GROUP based on CONFIG. Now we changed to look at MYSHARD, but this is still insufficient. we still remember that when we were working on LAB 3, we decided to de-duplicate, and we had to read it again when the news came back. Because the data may still be in REPLICA GROUP 1 when the request is sent. But when the message returned from RAFT, CONFIG was updated. The data is no longer in GROUP 1. Therefore, the logic of judging WRONG GROUP must be added to the data return layer.

At the same time, because the data will receive a new CONFIG in APPLY CH, part of the data to be TO OUT will be DELETE out of the DB. To ensure that the change of this DB will not affect the actual return value of GET during the transmission of NOTIFY CH. We need to inject the results into the OP when we receive APPLY CH. Otherwise, wait for the OP to send it and then take it from the DB, there is a certain probability that another thread has already DELETE DB at this time.

The code could be checked in github.

We then update POLL NEW CONFIG each by each.

From the hint:

*“Process re-configurations one at a time, in order.”*

We cannot go and get the next CONFIG, if the current CONFIG has nor finish sending SHARD yet.

## 14. How to delete unnecessary status:

In the above implementation, we opened 3 data structures, one is TO OUT, one is COME IN, and the other is MY SHARD; The third one is a fixed size. Don't think about it. Second, we will delete it after we have received the DATA.

The only one that is not recycled is the first one.

The most NAIVE implementation is that when we send the data as a REPLY, we delete it directly. This is dangerous. Because it is very likely that this message will be lost and rejected by the server over there, causing the data to never be recovered.

The approach is to wait until the other server successfully receives the DATA, and then deletes the corresponding COME IN. At this time, you should send a REQUEST to tell the TO OUT party that you can safely recycle the DATA in the TO OUT. But there are still situations where RPC will be lost. The same idea as PULL. (Use a COME IN LIST+ background thread to keep retrying. When it succeeds, delete the content of COME IN LIST, and don’t go to PULL until there is a new COME IN. If it fails, because the COME IN content is still there, it will automatically retry. , Not afraid of network instability)

When the DATA of COME IN is received, we need to mark this piece of data into the Garbage List. When the background GC thread finds that the Garbage List has content, it will send a GC RPC to the corresponding GROUP. After the corresponding GROUP is successfully cleaned up, REPLY informs. We delete the content corresponding to the Garbage List. Similarly, we still only interact with LEADER and use RAFT LOG to ensure that all nodes successfully delete GARBAGE, and then RPC replies SUCCESS.

We after implementing the rest part.

The code can be checked at github.

# Result / analysis:

## In part A, there are two main bugs:

Bug 1:

```
Test: Basic leave/join ...
panic: interface conversion: interface {} is *shardmaster.JoinArgs, not shardmaster.JoinArgs

goroutine 12 [running]:
shardmaster.(*ShardMaster).updateConfig(0xc4200803c0, 0x6a4429, 0x4, 0x66dbc0, 0xc420206c00)
        /home/zyx/Desktop/mit6824/mit6.824/src/shardmaster/server.go:127 +0x87f
shardmaster.StartServer.func1(0xc4200803c0)
        /home/zyx/Desktop/mit6824/mit6.824/src/shardmaster/server.go:247 +0x315
created by shardmaster.StartServer
        /home/zyx/Desktop/mit6824/mit6.824/src/shardmaster/server.go:236 +0x5ca
exit status 2
```

```
func (sm *ShardMaster) Join(args *JoinArgs, reply *JoinReply) {
    originOp := Op{ OpType: "Join", Args: *args, Cid: args.Cid, SeqNum: args.SeqNum}
    reply.WrongLeader = sm.templateHandler(originOp)
}

func (sm *ShardMaster) Leave(args *LeaveArgs, reply *LeaveReply) {
    originOp := Op{ OpType: "Leave", Args: *args, Cid: args.Cid, SeqNum: args.SeqNum}
    reply.WrongLeader = sm.templateHandler(originOp)
}

func (sm *ShardMaster) Move(args *MoveArgs, reply *MoveReply) {
    originOp := Op{ OpType: "Move", Args: *args, Cid: args.Cid, SeqNum: args.SeqNum}
    reply.WrongLeader = sm.templateHandler(originOp)
}

func (sm *ShardMaster) Query(args *QueryArgs, reply *QueryReply) {
    reply.WrongLeader = true;
    originOp := Op{ OpType: "Query", Args: *args, Cid: Nrand(), SeqNum: -1}
```

Bug 2: OP is NIL

```
0
panic: interface conversion: interface {} is nil, not shardmaster.Op

goroutine 12 [running]:
shardmaster.StartServer.func1(0xc420078410)
    /home/zyx/Desktop/mit6824/mit6.824/src/shardmaster/server.go:243 +0x38f
created by shardmaster.StartServer
    /home/zyx/Desktop/mit6824/mit6.824/src/shardmaster/server.go:236 +0x2d6

Process finished with exit code 2
```

After research, we found that any custom STRUCT needs to be registered with LABGOB, otherwise the analysis cannot be transmitted. After we fixed all bugs, we passed the test.

```
Test: Basic leave/join ...
  ... Passed
Test: Historical queries ...
  ... Passed
Test: Move ...
  ... Passed
Test: Concurrent leave/join ...
  ... Passed
Test: Minimal transfers after joins ...
  ... Passed
Test: Minimal transfers after leaves ...
  ... Passed
Test: Multi-group join/leave ...
  ... Passed
Test: Concurrent multi leave/join ...
  ... Passed
Test: Minimal transfers after multijoins ...
  ... Passed
Test: Minimal transfers after multileaves ...
  ... Passed
PASS
ok      shardmaster     3.890s
```

## In part B:

We added below code to decrease running time:

```
func (cfg *config) cleanup() {
    for gi := 0; gi < cfg.ngroups; gi++ {
        cfg.ShutdownGroup(gi)
    }
    cfg.mu.Lock()
    for i := 0; i < len(cfg.masterservers); i ++ {
        if cfg.masterservers[i] != nil {
            cfg.masterservers[i].Kill()
        }
    }
    cfg.mu.Unlock()
    cfg.net.Cleanup()
    cfg.checkTimeout()
}
```

It took 2 minutes for testing:

```
Test: unreliable 1...
  ... Passed
Test: unreliable 2...
  ... Passed
Test: unreliable 3...
  ... Passed
Test: shard deletion (challenge 1) ...
  ... Passed
Test: concurrent configuration change and restart (challenge 1)...
  ... Passed
Test: unaffected shard access (challenge 2) ...
  ... Passed
Test: partial migration shard access (challenge 2) ...
  ... Passed
PASS
ok      shardkv 120.676s
```

The biggest challenge of this project is understanding how to deal with modification of the configuration file during the service process, which is the correspondence and relationship between group and shard. For example, the data requested by the client at this time is exactly the data being migrated. We need to confirm whether this request is before or after the migration. If it is before the migration, the new share corresponding to the requested date will be notified. If it is after the migration, we can just simply retry the request.

# Conclusion/Future work

We have met so many challenges during the development of this shard key/value project. However, we learnt a lot through the process, such as a shared storage system must be able to shift shards among replica groups. There are learning resources available on the internet and we got chances to exchange ideas with other learners. We also learnt the importances of sharding, and the benefit of it. It is necessary if a dataset is too large to be stored in a single database. Furthermore, the strategies of sharding we have used allow additional machines to be added. It allows a database cluster to scale along with its data and traffic growth. Sharding could be a great solution for those looking to scale their database horizontally.

By reading useful articles and guidelines, we have a clearer understanding of the pros and cons of sharding. In the future, we plan to use this insight to make a more informed decision about whether or not a shared database architecture is suitable for applications, and knowing the process to go through to develop a sharded key/value service. Also, as the functions of this project are fairly basic, some functions are not implemented: (1) The transmit between shards is slow and concurrent client access is not allowed. (2) the member in each raft group will never be changed. We will try to complete those functions in the future.

# Reference:


1, The project/Lab4 description: 6.824 Lab 4: Sharded Key/Value Service:

https://pdos.csail.mit.edu/6.824/labs/lab-shard.html

2, Tutorial for: MIT 6.824: Lab 4 Sharded KeyValue Service Implementation:

https://wiesen.github.io/post/mit-6.824-lab4-sharded-keyvalue-service/

3, SIDNEY PRIMAS, 2018, A FAULT-TOLERANT, SHARDED KEY-VALUE STORAGE SERVICE:

https://www.sidneyprimas.com/sharded-storage-system

4, From Wikipedia: Key–value database:

https://en.wikipedia.org/wiki/Key%E2%80%93value_database

5, “bysoul”, 2018, 6.824 Lab4: Sharded Key/Value Service:

https://zhuanlan.zhihu.com/p/144995713