1. Summary

In this project, I design a Load Balancer for Key-value Database (LBKVD for short). LBKVD is written in java, and uses network to conmmunicate with database instance through database defined protocol. The main goal is to scale database service at least linearly as more database instance as added.

2. Why does LBKVD matters

Think about thi senario. Assume you develop a social website for pets, especially for cats and dogs. Because you don't have money, you decided to use a light-weight open source database in backend. At the beginning, you website is not well-known, only pets of your friends use you service, so pressure on database is not that critical. However, you website becomes popular and users grow exponentially.

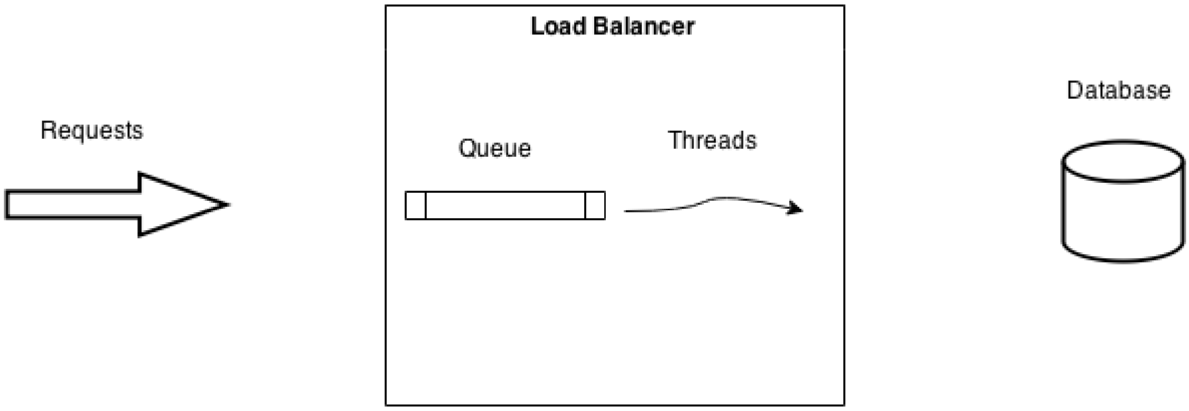
At first, you are very happy because you will become next zark muckerberg. Then you realize a problem: you need to scale your database to maintain your website, before you can get money from investors, and become a billionaire. For now, you are poor, so still need to use open-source database that used in the past. In the case, how can you scale your website?

Then answer is: Load Balancer for Key-Value Database!

3. Experiment

3.1 Description

The basic structure of my LBKVD is fixed, no matter what kind of schedule policy I use. Please see below:



So basicly, LBKVD creates queues to contain coming requets, and also creates worker threads to fetch requests from queues and send to databases. Worker threads are also responsible to response requests.

Therefore, based on structure of LBKVD, load balancer schedule policy is actually a strategy to decide how to utilze queues and worker threads, to assgin jobs to database instances.

3.2 Trace file, software and hardware

The trace file I created contains 300,000, 4KB insert operations with unique keys.

The cluster that I used for testing, has four nodes. Each node has two, four core Xeon E5345 processor (2.33GHz, 8M L2 cache, no hyper-threading), 16GB memory and hard disk(15000RPM).

The Key-Value database, is acutally a document database, named EmeraldDB.

Following is my policy design.

3.3 Navie assignment

There is only one queue in LBKVD, and one worker thread for each database node. LBKVD assign requests to each worker thread one by one. So every database node will be assigned equal amount of work.

The problem of this assignment, is that it is so simple. There is no multi-threading to hidden latency, and there is no flexible schedule on work assignment. What if a node is idle while another one is busy? So I acutally never seriously implement this policy.

3.4 Equal assignment

There is one queue and one worker thread for each database node. Work assgiment is dynamic, i.e. LBKVD assgin a job based on current workload of each node. LBKVD can monitor length of queue of each node, and pick up the one with shortest queue depth.

	Baseline	1 Node	2 Node	3 Node	4 Node
Speed up	1x (149s)	0.88x (169s)	1.86x (80s)	2.64x (56s)	3.35x (44s)
throughput (ops / sec)	2063	1827	3925	5700	7111

note: Baseline here is trace file tested on one database instance.

I got pretty good result here, except for Equal assignment on 1 Node. The reason is clear, there are queues in equal assignment, but not exist in baseline. Packet requests in queue. fetch and unpacketing costs a lot. Things get better when the number of nodes increase.

However, this policy still cannot do the best. For each thread, it read data from disk, put it into memory buffer, and then put to network buffer, and send out, there exists lots of change that worker thread is waiting and do nothing,while lots of requests are in the queue. So multi-threading is possible to improve throughput.

3.5 Equal assignment with multi worker threads

There is one queue but multi worker threads for each database node. Work assgiment is still the same as Equal assignment. Each database node establish multi connection with LBKVD, and receive multi requests at the same time. There are locks on queues, in order to provide concurrency control under multi-threading situation.

	Baseline	1 Node	2 Node	3 Node	4 Node
Speed up	1x (169s)	1.13x (129s)	4.89x (34s)	6.76x (25s)	9.69x (17s)
throughput (ops / sec)	1827	5011	9062	12445	17065

Note: Each database instance correspondes to 3 worker threads. Baseline here is Equal assignment with 1 node.

When create multi-threads for a database instance, things get better compared to one worker thread. Data in the table is using 3 worker threads and one queue for each node. I tried create more threads for each node, however, they cannot beat 3 threads per node. I guess the reason is that, the Load Balancer I ran on the machine that only has two, four core Xeon E5345 processors. This CPU doesn't have hype-threading, so the maximum threads it can execute, is 8. CPU cannot afford too many threads, and it seems like too many threads acutally reduce the throughput of Load Balancer. 4 nodes with 3 threads each in Load Balancer has already achieve peek rate.

Still, this assignment has a problem, it releases consistency. Assume here is two operations coming one by one: "INSERT a", "QUERY a". These two operations has the same key, so they are assigned to same node. Because there is a queue and multi-threading, these two operations may be sent to node at the same time. It is very likely that "QUERY a" arrives before "INSERT a", then query operation returns nothing.

In order to fix this **weak consistency problem**, finally I come up with a multi threads with multi queues approach.

3.6 Equal assignment with multi worker threads and multi queues

There are multi queues and multi workder threads for each database node. In this policy, because of increase number of queues and worker threads, maybe thirty or more, it is hard to monitor status of each queue and worker threads, so I use **consistenct hashing** here, in order to have a relatively balanced workload assignment under complexity situation. queues for database nodes are mapped to a circle, as well as keys of records.

This approach guarantees that operation serialization. Operations with same keys always sent to the same, only queue. So operations are serialized based on the time they arrive at Load Balancer, while throughput still holds as the same as last policy.

4. In the end...

	Trace file on one database instance	4 database instances, each one corresponding to a queue and 3 worker threads
Speed up	1x (149s)	8.52x (17s)
throughput (ops / sec)	2063	17065

note: Baseline here is trace file tested on one database instance.

Here is best implementation vs baseline, and I believe this should be not the end…...

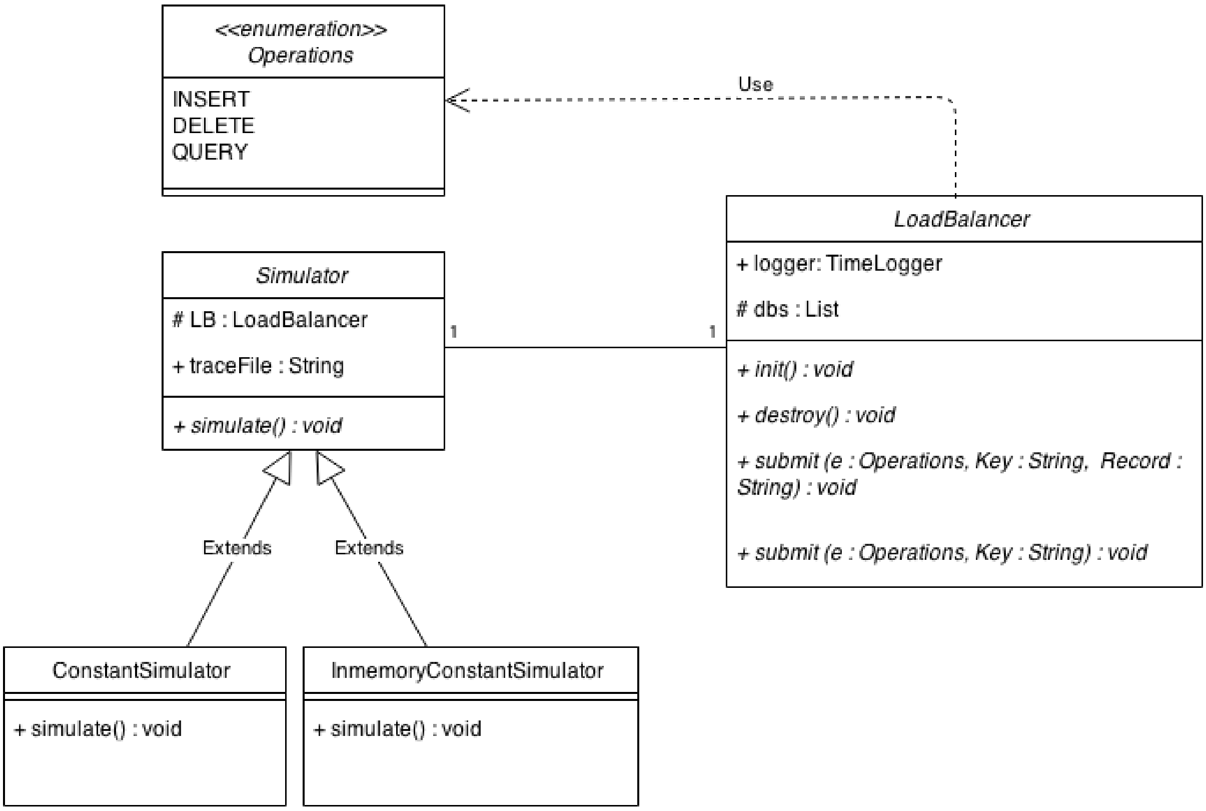
5. Retrospective

Although I have achieved big breakthrough by incrementally improving design and implementation of LBKVD, I still have ideas haven't tried, which I think can keep improving performance of LBKVD. Here is some idea.

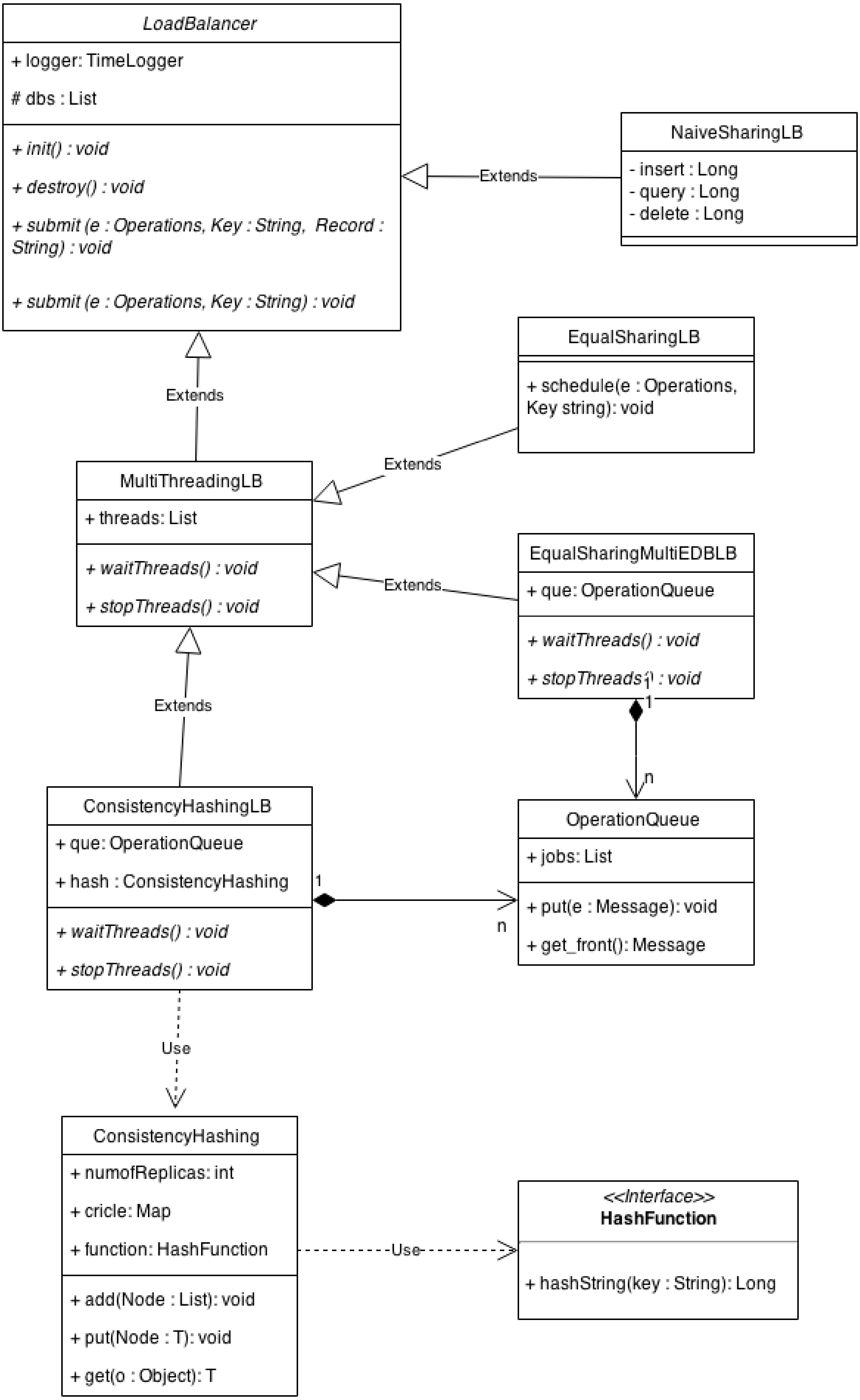
a. It turnes out best number of worker threads is 12 on my testing machine. Increasing this number will not bring any benefit. So this is one of the potential bottlenecks of LBKVD. So, why I need put all the logic of LBKVD on one machine? Why cannot I use client-server model for LBKVD? So new design of LBKVD sounds like this: server of LBKVD is in charge of receiving requests. In the meantime, each node will not only install database instance, but also install client of LBKVD. Server can send requests to clients, and each client can lanuch 12 worker threads and handle requests. Client, instand of server, commnunicates with database instance.

b. One of the problem of idea a, is that server may become new bottleneck, e.g. network becomes bottleneck. So new idea is peer to peer system. Client can talk to each other, and reduce pressure on server. For example, server may avoid complicated scheduling policy, instead, server sends request to a random client. Client can decide the right node, to which this request shoud be sent.

6.UML



<<enumeration>>
Operations
INSERT
DELETE
QUERY
Use
LoadBalancer
+ logger: TimeLogger
# dbs : List
+ init() : void
+ destroy() : void
+ submit (e : Operations, Key : String, Record : String) : void
+ submit (e : Operations, Key : String) : void
Simulator
# LB : LoadBalancer
+ traceFile : String
+ simulate() : void
1
1
Extends
Extends
ConstantSimulator
+ simulate() : void
InmemoryConstantSimulator
+ simulate() : void


LoadBalancer
+ logger: TimeLogger
# dbs : List
+ init() : void
+ destroy() : void
+ submit (e : Operations, Key : String, Record : String) : void
+ submit (e : Operations, Key : String) : void
NaiveSharingLB
- insert : Long
- query : Long
- delete : Long
Extends
EqualSharingLB
+ schedule(e : Operations, Key string): void
Extends
MultiThreadingLB
+ threads: List
+ waitThreads() : void
+ stopThreads() : void
Extends
EqualSharingMultiEDBLB
+ que: OperationQueue
+ waitThreads() : void
+ stopThreads() : void
Extends
1
1
n
Extends
ConsistencyHashingLB
+ que: OperationQueue
+ hash : ConsistencyHashing
+ waitThreads() : void
+ stopThreads() : void
1
n
OperationQueue
+ jobs: List
+ put(e : Message): void
+ get_front(): Message
Use
ConsistencyHashing
+ numofReplicas: int
+ cricle: Map
+ function: HashFunction
+ add(Node : List): void
+ put(Node : T): void
+ get(o : Object): T
Use
<<Interface>>
HashFunction
+ hashString(key : String): Long

7. Reference

[1] Intel® Xeon® Processor E5345 (8M Cache, 2.33 GHz, 1333 MHz FSB), link.

[2] Document of EmeraldDB, a light-weight NoSQL Database.

[3] Wikipedia: Consistent hashing, link.