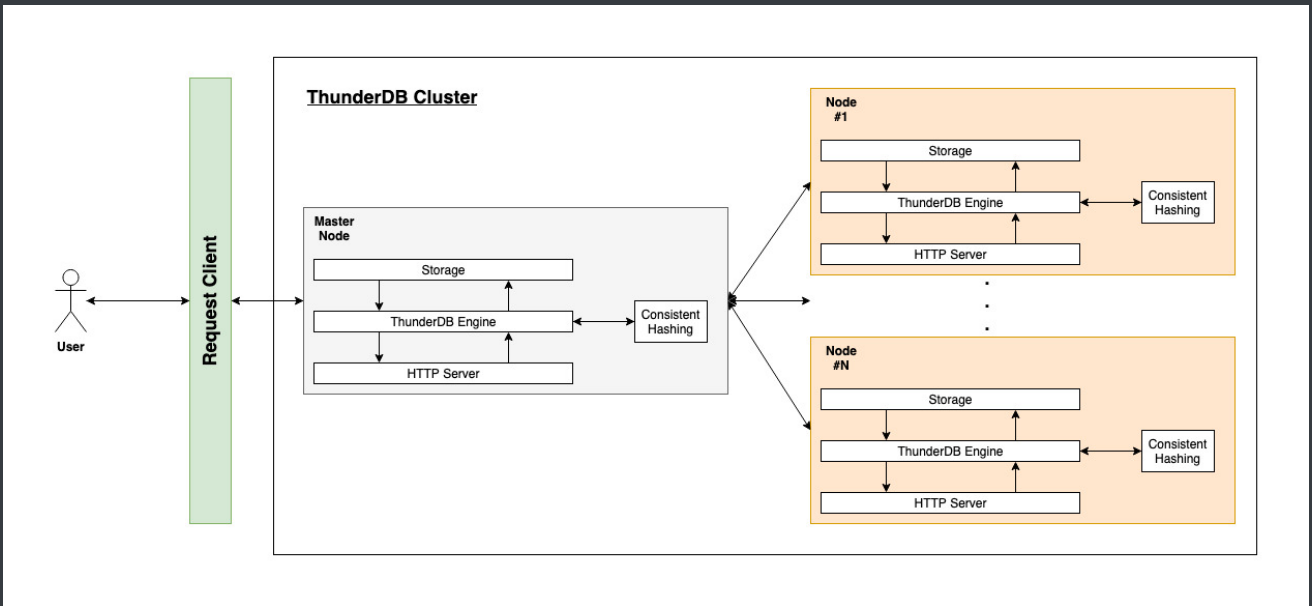
ThunderDB

ThunderDB is a scalable and maintainable key-value store that is built on Python's core dictionary data structure, making the application minimal and lightweight.

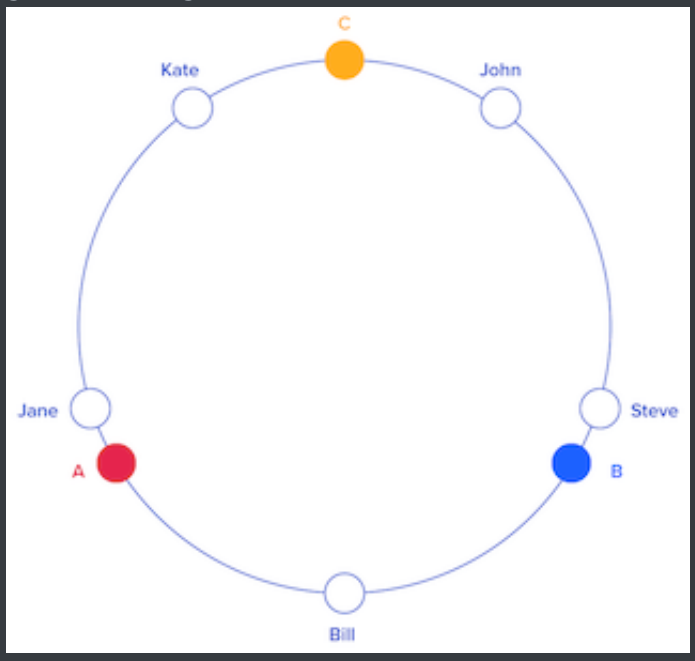
Design

The high-level design of the system is fairly straightforward. ThunderDB provides the option to run as a single node (for similicity and testing) as well as in a distributed context, with multiple nodes on your network. Let's take a look at the architecture diagram of the entire system, and then we will dive deeper into each component:



- **Request Client**: This is any client you wish to use to make your HTTP requests to ThunderDB. You can simply use Curl, or you can use something more complex like Postman.
- **ThunderDB Cluster**: A cluster is simply a collection of nodes in your network. For a single-node system, this will only consist of one node, which would be the master node. For a distributed context, this would be made up of all the nodes in your network that are running as a server for ThunderDB.
- **Master Node:** The node in your system which is accepting requests on behalf of the user. Based on the way ThunderDB was implemented, this can be any node in the network, however, for practicallity we will be denoting a single node as master.

- **Consistent Hashing**: A utility that allows us to perform consistent hashing[5] on a key and determine which node the key is/should be stored on. Based on the number of nodes, Consistent Hashing gives us a "ring" of values to let us determine which node covers which hash value ranges.



  In the example above, the hashed value of "Bill" fell in the range of Node B, therefore, Bill would be stored in Node A (first node clockwise from the hash value). Similarly, the hashed value of Jane, fell in the range of Node C, therefore, we can store/retrieve the value corresponding to the key "Jane" in Node C. As you add or remove nodes from the ring, the hash range covered by a given node will decrease or increase, respectively.

- **HTTP Server**: An HTTP server is used to serve and host requests/responses from the user. We use the minimalistic Bottle library in Python to accomplish this task. A server is started on each node, this way the master node has a mechanism to communicate with the other nodes in the cluster.
- **ThunderDB Engine**: The engine is responsible for executing the core operations of our key-value store (PUT, GET, DELETE). The engine will also use our Consistent Hashing component (in the distributed setting, i.e.: more than 1 node), to determine which node to execute the operation on.
- **Storage**: ThunderDB has built a simple abstraction for our storage system. In this case, our implementation is an in-memory store (Python core dictionary). However, you can build any storage system on-top of this abstraction and plug it into ThunderDB. For instance, you may want to use Redis or Memcached as your storage layer. With ThunderDB, this is as easy as including this Python requirement and implementing our storage abstraction with the library or data structure of your choice.

Each component in the diagram performs a specific task(s) and have been designed to be easily extensible.

Single Node

In our simplest use-case, our key-value store can be run on a single node (Master Node). In this case we can load large amounts of data very quickly (metrics in the Performance section below) because we can simply populate the Python dictionary, in O(1) time per insertion. When a user makes a request, we can perform a get operation in O(1) time.

For instructions on how to run in single-node mode: please visit the README.

Distributed Nodes

In our distributed setting , we can insert a key-value pair into our cluster by using Consistent Hashing, which ensures proper partitioning, allowing us to utilize less memory/storage space on each node. A single insertion or retreival in this setting will take O(log n) time because the time complexity of consistent hashing overwhelms the insert/get operation on a Python dictionary.

For instructions on how to run in distributed mode, which will require a couple extra dependencies to simulate multiple machines: please visit the README.

Choice of Third-Party Libraries

Bottle

Bottle[3] is a mimialistic HTTP library that gives us basic mechanisms to handle requests and responses. Bottle is similar to other frameworks like Flask, but without the unnecessary code that our application will not need. The choice of using waitress over the traditional uwsgi server in Bottle was purely for performance gains.

Pandas

Pandas[4] allows us more flexibility when reading in large datasets (> 1 million records). We use the power of Pandas to efficiently read chunks of a space-delimited file into a format that is easily processible by our ThunderDB Engine.

Performance

For measuring our performance metrics, we use the following tools/techniques:

- Simply timing the operation in Python
- ab[6] (apache HTTP server benchmarking tool): used to test concurrency of the cluster

Single Node

Operation / Metric	Value (Unit of Measurement)

GET	0.0000381 seconds
PUT	0.0000102 seconds
Number of Total GET Requests / Concurrent Requests	10000 total / 100 concurrent
Time to Execute all GET Requests	3.9 seconds
Avg time to Execute each GET Request	39 ms
Memory Usage Per Node (1 million records)	80 MB

Distributed Nodes

Operation / Metric	**Value (Unit of Measurement)**
GET	0.0000643 seconds
PUT	0.0000437 seconds
Number of Total GET Requests / Concurrent Requests	10000 total / 100 concurrent
Time to Execute all GET Requests	4.6 seconds
Avg time to Execute each GET Request	46 ms
BATCH PUT (10,000 items)	1.26 seconds
BATCH PUT (100,000 items)	5.92 seconds
BATCH PUT (1,000,000 items)	54.1 seconds
Memory Usage Per Node (1 million records over three nodes)	27 MB

Note: The numbers in the chart above will vary depending on the specs of your machine

For comparison, in the early days of DynamoDB[2] from AWS, inserting 10 million records without any optimizations would take about 18 hours[1] (approx 150 items per second). Obviously, this metrics has changed over the years and has significantly improved. None the less, the BATCH PUT operation is a very complex task in a distributed system. One way to improve our number in ThunderDB would be to create an API that allows for BATCH PUT by accepting a larger JSON object at request time, instead of having to issue one single HTTP request for each key-value pair.

Assumptions and Known Deficiencies

- In a single node setting you will always be limited by the amount of memory/storage space on the machine. Eventually you will hit a hard ceiling here.
- In a distributed setting, we utilize network calls to PUT/GET the data from another node. When

loading large amounts of data initially, this is a cause of some un-optimality.
- There is no replication, meaning that the key-value pair is stored in a single node. If this node is taken down in the network, the data will be lost. Thie can be fixed by adding replication in the distributed setting. Doing so in a trivial manner adds a considerable amount of latency, so some optimizations would be required.

Extra Information

- This solution was developed in a little under 2 hours.

References

- [1] https://www.keithrozario.com/2017/12/writing-millions-of-rows-into-dynamodb.html
- [2] https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- [3] https://github.com/bottlepy/bottle
- [4] https://github.com/pandas-dev/pandas
- [5] https://en.wikipedia.org/wiki/Consistent_hashing
- [6] https://httpd.apache.org/docs/2.4/programs/ab.html