Exploring Cassandra and HBase with BigTable Model

Hemanth Gokavarapu
hemagoka@indiana.edu
(Guidance of Prof. Judy Qiu)
Department of Computer Science
Indiana University Bloomington

Abstract

Cassandra is a distributed database designed to be highly scalable both in terms of storage volume and request throughput while not being subject to any single point of failure. Cassandra aim to run on top of an infrastructure of hundred nodes (Possibly spread across different data centers.). HBase is an open source, non-relational, distributed database modeled after Google's BigTable and is written in Java. It is developed as part of Apache Software Foundation's Hadoop project and runs on top of HDFS (Hadoop Distributed Filesystem), providing BigTable-like capabilities for Hadoop. This paper presents a overview of BigTable Model with Cassandra, HBase and discusses how its design is founded in fundamental principles of distributed systems. We will also discuss the usage and compare both the NoSQL databases in detailed.

Introduction to NOSQL

Over the last couple of years, there is an emerging data storage mechanism for storing large scale of data. These storage solutions differ quite significantly with the RDMS model and are also known as the NOSQL. Some of the key players include GoogleBigTable, HBase, Hypertable, AmazonDynamo, Voldemort, Cassandra, Redis, CouchDB, MongoDB.

These solutions have a number of characteristics in common such as

- Key value store
- Run on large number of commodity machines
- Data are partitioned and replicated among these machines
- Relax the data consistency requirement.

Data Model: The underlying data model can be considered as a large Hashtable (key/ value store).

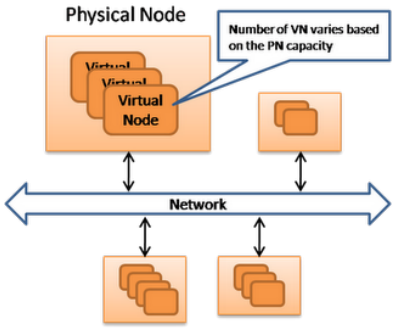
The basic form of API access is

- get(key) – Extract the value given a key
- put(key,value)—Create or update the value given its key
- delete (key) – remove the key and its associated value
- execute (key, operation, parameters) – invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map...etc).
- Mapreduce (keyList, mapFunction, reduceFunction)

– invoke a map/reduce function across a key range.

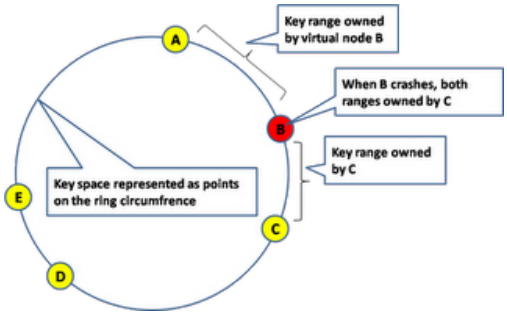
Machine Layout:

Machines connected through a network. We can each machine a physical node (PN). Each PN has the same set of software configuration but may have varying hardware capacity in terms of CPU, memory and disk storage. Within each PN, there will be a variable number of virtual node (VN) running according to the available hardware capacity of the PN.



Data partitioning:

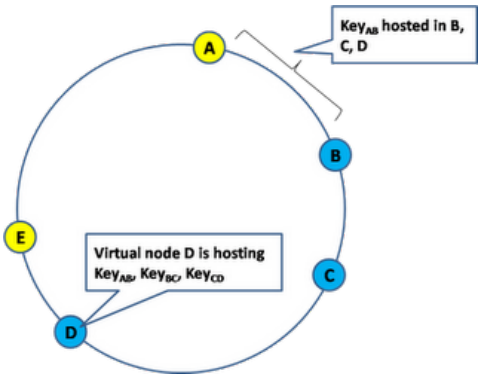
The overall hashtable is distributed across many VNs, we need a way to map each key to the corresponding VN. One way is to use partition= key mod(total_VNs). The disadvantage of this scheme is when we alter the number of VN's then the ownership of existing keys has changed dramatically, which requires full data redistribution. Most large scale store use a consistent hashing technique to minimize the amount of ownership changes.



In the consistent hashing scheme, the key space is finite and lies on the circumference of a ring. The Virtual node id is also allocated from the same key space. For any key, its owner node is defined as the first encountered virtual node if walking clockwise from that key. If the owner node crashes, all the key it owns will be adopted by its clockwise neighbor. Therefore, key redistribution happens only within the neighbor of the crashed node, all other nodes retain the same set of keys.

Data Replication:

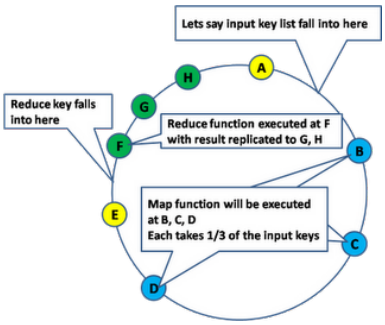
To provide high reliability from individual unreliable resources, we need to replicate the data partitions. Replication not only improves the overall reliability of data, it also helps performance by spreading the workload across multiple replicas. While read-only request can be dispatched to any replicas, update request is more challenging because we need to carefully coordinate update, which happens in these replicas.



Map Reduce Execution:

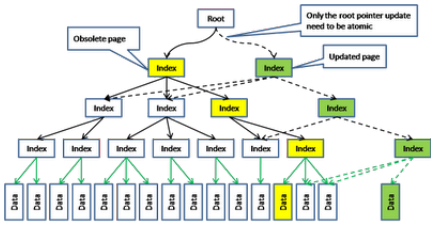
The distributed store architecture fits well into distributed processing as well. For example, to process a Map/ Reduce operation over an input key lists. The system will push the map and reduce function to all the nodes (ie: moving the processing logic towards the data). The map function of the input keys will

be distributed among the replicas of owning those input, and then forward the map output to the reduce function, where the aggregation logic will be executed.

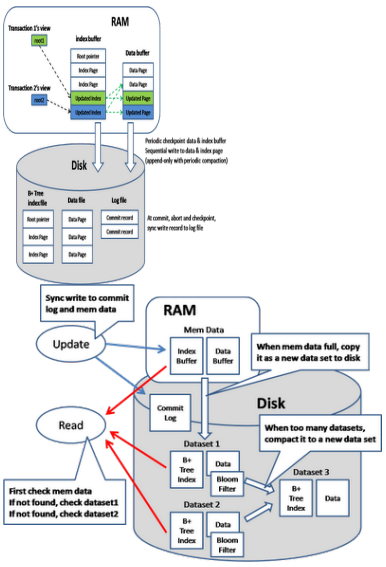


Storage Implementation:

One strategy is to use make the storage implementation pluggable. e.g. A local MySQL DB, Berkeley DB, Filesystem or even a in memory Hashtable can be used as a storage mechanism. Another strategy is to implement the storage in a highly scalable way. Here are some techniques that we can learn from CouchDB and Google BigTable. CouchDB has a MVCC model that uses a copy-on-modified approach. Any update will cause a private copy being made which in turn cause the index also need to be modified and causing the a private copy of the index as well, all the way up to the root pointer.



Copy on modify. Everyone sees his own copy of update
Finally the root pointer is swapped and everyone's view is updated
Yellow page becomes garbage over time.
File will be compacted periodically by copying to a different file.



Notice that the update happens in an append-only mode where the modified data is appended to the file and the old data becomes garbage. Periodic garbage collection is done to compact the data. Here is how the model is implemented in memory and disks. In Google BigTable model, the data is broken down into multiple generations and the memory is use to hold the newest generation. Any query will search the mem data as well as all the data sets on disks and merge all the return results. Fast detection of whether a generation contains a key can be done by checking a bloom filter. When update happens, both the mem data and the commit log will be written so that if the machine crashes before the mem data flush to disk, it can be recovered from the commit log.

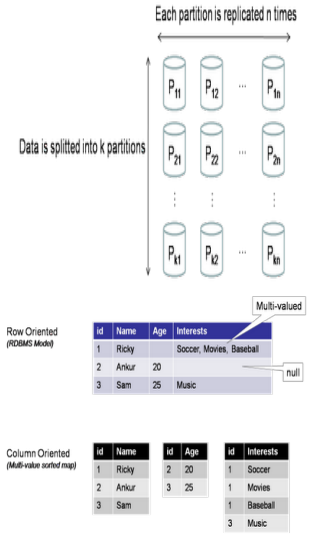
Big Table Model:

Both Hbase and Cassandra are based on Google BigTable model. Some of the key characteristics of BigTable are discussed below.

Fundamentally Distributed:

BigTable is built from the ground up on a "highly distributed", "share nothing"

architecture. Data is supposed to store in large number of unreliable, commodity server boxes by "partitioning" and "replication". Data partitioning means the data are partitioned by its key and stored in different servers. Replication means the same data element is replicated multiple times at different servers.

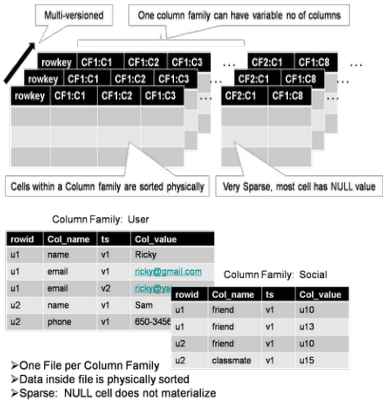


Column Oriented:

Unlike traditional RDBMS implementation where each "row" is stored contiguous on disk, BigTable, on the other hand, store each column contiguously on disk. The underlying assumption is that in most cases not all columns are needed for data access, column oriented layout allows more records sitting in a disk block and hence can reduce the disk I/O. Column oriented layout is also very effective to store very sparse data (many cells have NULL value) as well as multi-value cell. The following diagram illustrates the difference between a Row-oriented layout and a Column-oriented layout.

Variable number of Columns:

In RDBMS, each row must have a fixed set of columns defined by the table schema, and therefore it is not easy to support columns with multi-value attributes. The BigTable model introduces the "Column Family" concept such that a row has a fixed number of "column family" but within the "column family", a row can have a variable number of columns that can be different in each row.



In the Bigtable model, the basic data storage unit is a cell, (addressed by a particular row and column). Bigtable allow multiple timestamp versions of data within a cell. In other words, user can address a data element by the rowid, column name and the timestamp. At the configuration level, Bigtable allows the user to specify how many versions can be stored within each cell either by count (how many) or by freshness (how old). At the physical level, BigTable store each column family contiguously on disk (imagine one file per column family), and physically sort the order of data by rowid, column name and timestamp. After that, the sorted data will be compressed so that a disk block size can store more data. On the other hand, since data within a column family usually has a similar pattern, data compression can be very effective.

Sequential Write:

BigTable model is highly optimized for write operation (insert/update/delete) with

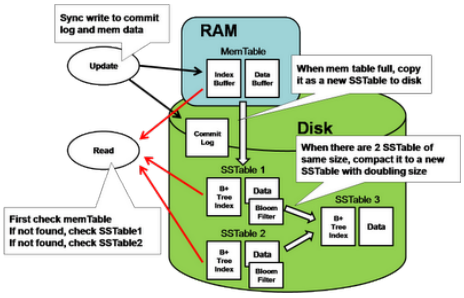
sequential write (no disk seek is needed). Basically, write happens by first appending a transaction entry to a log file (hence the disk write I/O is sequential with no disk seek), followed by writing the data into an in-memory Memtable . In case of the machine crashes and all in-memory state is lost, the recovery step will bring the Memtable up to date by replaying the updates in the log file. All the latest update therefore will be stored at the Memtable, which will grow until reaching a size threshold, then it will flushed the Memtable to the disk as an SSTable (sorted by the String key). Over a period of time there will be multiple SSTables on the disk that store the data.

Merged Read:

Whenever a read request is received, the system will first lookup the Memtable by its row key to see if it contains the data. If not, it will look at the on-disk SSTable to see if the row-key is there. We call this the "merged read" as the system need to look at multiple places for the data. To speed up the detection, SSTable has a companion Bloom filter such that it can rapidly detect the absence of the row-key. In other words, only when the bloom filter returns positive will the system be doing a detail lookup within the SSTable.

Periodic Data Compaction:

As you can imagine, it can be quite inefficient for the read operation when there are too many SSTables scattering around. Therefore, the system periodically merge the SSTable. Notice that since each of the SSTable is individually sorted by key, a simple "merge sort" is sufficient to merge multiple SSTable into one. The merge mechanism is based on a logarithm property where two SSTable of the same size will be merge into a single SSTable will doubling the size. Therefore the number of SSTable is proportion to O(logN) where N is the number of rows.



HBASE:

Based on the BigTable, HBase uses the Hadoop Filesystem (HDFS) as its data storage engine. The advantage of this approach is then HBase doesn't need to worry about data replication, data consistency and resiliency because HDFS has handled it already. Of course, the downside is that it is also constrained by the characteristics of HDFS, which is not optimized for random read access. In addition, there will be an extra network latency between the DB server to the File server (which is the data node of Hadoop). In the HBase architecture, data is stored in a farm of Region Servers. The "key-to-server" mapping is needed to locate the corresponding server and this mapping is stored as a "Table" similar to other user data table.

Before a client do any DB operation, it needs to first locate the corresponding region server.

1. The client contacts a predefined Master server who replies the endpoint of a region server that holds a "Root Region" table.
2. The client contacts the region server who replies the endpoint of a second region server who holds a "Meta Region" table, which contains a mapping from "user table" to "region server".
3. The client contacts this second region

server, passing along the user table name. This second region server will lookup its meta region and reply an endpoint of a third region server who holds a "User Region", which contains a mapping from "key range" to "region server"

4. The client contacts this third region server, passing along the row key that it wants to lookup. This third region server will lookup its user region and replies the endpoint of a fourth region server who holds the data that the client is looking for.

5. Client will cache the result along this process so subsequent request doesn't need to go through this multi-step process again to resolve the corresponding endpoint.

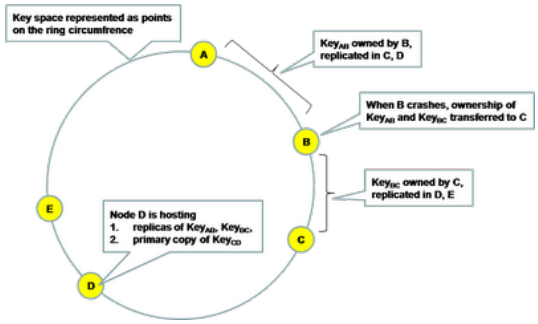
In Hbase, the in-memory data storage (what we refer to as "Memtable" in above paragraph) is implemented in Memcache. The on-disk data storage (what we refer to as "SSTable" in above paragraph) is implemented as a HDFS file residing in Hadoop data node server. The Log file is also stored as an HDFS file. (I feel storing a transaction log file remotely will hurt performance) Also in the HBase architecture, there is a special machine playing the "role of master" that monitors and coordinates the activities of all region servers (the heavy-duty worker node). The master node is the single point of failure at this moment.

CASSANDRA:

Cassandra is open sourced by Facebook in July 2008. This original version of Cassandra was written primarily by an ex-employee from Amazon and one from Microsoft. It was strongly influenced by Dynamo, Amazon's pioneering distributed key/value database. Based on the BigTable model, Cassandra uses the DHT model to partition its data.

Consistent Hashing via O(1) DHT: Each machine (node) is associated with a particular id that is distributed in a keyspace (e.g. 128 bit). All the data element is also associated with a key (in the same key space). The server owns all the data whose key lies between its id and the preceding server's id.

Data is also replicated across multiple servers. Cassandra offers multiple replication schema including storing the replicas in neighbor servers (whose id succeed the server owning the data), or a rack-aware strategy by storing the replicas in a physical location. The simple partition strategy is as follows.



Tunable Consistency Level:

Unlike Hbase, Cassandra allows you to choose the consistency level that is suitable to your application, so you can gain more scalability if willing to tradeoff some data consistency.

For example, it allows you to choose how many ACK to receive from different replicas before considering a WRITE to be successful. Similarly, you can choose how many replicas' response to be received in the case of READ before return the result to the client.

By choosing the appropriate number for W and R response, you can choose the level of

consistency you like. For example, to achieve Strict Consistency, we just need to pick W, R such that W + R > N. This including the possibility of (W = one and R = all), (R = one and W = all), (W = quorum and R = quorum). Of course, if you don't need strict consistency, you can even choose a smaller value for W and R and gain a bigger availability. Regardless of what consistency level you choose, the data will be eventual consistent by the "hinted handoff", "read repair" and "anti-entropy sync" mechanism described below.

Hinted Handoff:

The client performs a write by send the request to any Cassandra node, which will act as the proxy to the client. This proxy node will located N corresponding nodes that holds the data replicas and forward the write request to all of them. In case any node is failed, it will pick a random node as a handoff node and write the request with a hint telling it to forward the write request back to the failed node after it recovers. The handoff node will then periodically check for the recovery of the failed node and forward the write to it. Therefore, the original node will eventually receive all the write request.

Conflict Resolution:

Since write can reach different replica, the corresponding timestamp of the data is used to resolve conflict, in other words, the latest timestamp wins and push the earlier timestamps into an earlier version (they are not lost)

Read Repair:

When the client performs a "read", the proxy node will issue N reads but only wait for R copies of responses and return the one with the latest version. In case some nodes respond with an older version, the proxy node will send the latest version to them asynchronously; hence these left-behind nodes will still eventually catch up with the latest version.

Anti-Entropy data sync : To ensure the data is still in sync even there is no READ and WRITE occurs to the data, replica nodes periodically gossip with each other to figure out if anyone out of sync. For each key range of data, each member in the replica group compute a Merkel tree (a hash encoding tree where the difference can be located quickly) and send it to other neighbors. By comparing the received Merkel tree with its own tree, each member can quickly determine which data portion is out of sync. If so, it will send the diff to the left-behind members.

Anti-entropy is the "catch-all" way to guarantee eventual consistency, but is also pretty expensive and therefore is not done frequently. By combining the data sync with read repair and hinted handoff, we can keep the replicas pretty up-to-date.

Hbase Vs Cassandra:

Finally, we can compare Hbase versus Cassandra in the following way.

	Cassandra	**HBase**
Written In:	Java	JAVA
Main Point:	Best of BigTable and Dynamo	Billions of rows and X millions of columns
License:	Apache	Apache
Protocol:	Custom, Binary (Thrift)	HTTP/ REST (also Thrift)
Trade Off:	• Tunable trade-off for distribution and replication(N, R, W) • Querying by column,range of keys • BigTable-like features: columns, column families • Writes are much faster than reads • Map / reduce possible with Apache Hadoop. • I admit being a bit biased against it, because of the bloat and complexity it has partly because of Java (Configuration, seeing exceptions, etc)	• Modeled after BigTable • Map / reduce with Hadoop • Query prediction push down via server side scan and get filters. • Optimizations for real time queries. • A high performance Thrift gateway • HTTP supports XML, Protobuf and Binary • Cascading, hive, and pig source and sink modules. • Jruby-based (JIRB) shell • No single point of failure • Rolling restart for configuration changes and minor upgrades. • Random access performance is like MySQL
Best Used:	• When you write more than you read. If every component of the system must be in Java.	• If you are in love with BigTable. And when you need random, realtime read/ write access to your Big Data.
Example	• Banking, Financial industry. Writes are faster than reads, so one natural niche is real time data analysis.	• Facebook Messaging Database

CONCLUSION:

This paper has provided and introduction to NoSQL and fundamental distributed system principals on which it is based. The concept of BigTable Model has been discussed in detailed along with its usage in Cassandra and Hbase. We have also seen the

differences between Cassandra and Hbase and their usage.

REFERENCES:

[1] O'Reilly Cassandra- The Definitive Guide by Eben Hewitt.

[2] Wikipedia (2010). Social media [www]. Retrieved from <http://en.wikipedia.org/wiki/Social_media> on March 9th, 2010.

[3] The NoSQL Community (2010). NoSQL Databases [www]. Retrieved from <http://nosql-database.org> on April 19th, 2010.

[4] Vineet Gupta (2010). NoSql Databases - Landscape [www]. Retrieved from<http://www.vineetgupta.com/2010/01/nosql-databases-part-1-landscape.html> on March 9th, 2010.

[5] Jeffery Dean et. al. (2006). Bigtable: A Distributed Storage System for Structured Data. In Proceedings of OSDI 2006, Seattle, WA.

[6] Brewer, E. (2004). Towards Robust Distributed Systems [www]. Retrieved from http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf> on April 27th, 2010.

[7] Cassandra Wiki (2010). Hinted Handoff. Retrieved from <http://wiki.apache.org/cassandra/HintedHandoff> on May 2nd, 2010.

[8] Eure, I. (2009). Looking to the future with Cassandra. Retrieved from <http://about.digg.com/blog/looking-future-cassandra> on May 2nd, 2010.

[9] Chang, et al. (2006). Bigtable: A Distributed Storage System for Structured Data

[10] ^ Powered By HBase

[11] Hbase Vs Cassandra: NoSQL Battle

[12] Cassandra and HBase Compared

[13] Learning NoSQL from Twitter's Experience