Distributed Storage System

[1- Cap Theorem](#h.x4bcfkkzez2l)

[2- Amazon Dynamo](#h.qb09ew4y8mqg)

[Intro](#h.7yn5ns989irl)

[Main components explanation](#h.kol90hbf0tsc)

[Merkle trees](#h.4st81xik1f4v)

[Data Partitioning](#h.w21l7a72bwmm)

[Data replication](#h.vvc59xrdorfs)

[Data Versioning](#h.bqr746ld6yf4)

[Put and Get operations](#h.5c4engm0c76f)

[Hinted handoff](#h.ufy7oo3ww1bp)

[Membership management](#h.ki0p3pri0ym7)

[Failure detection](#h.fko0znhog6g3)

[3- Cassandra](#h.g7qsk0wwceui)

[Intro](#h.2xfml9n9kwt9)

[Replication](#h.w2f2pf8tm950)

[Snitches](#h.2ot18oucxktn)

[File system](#h.yk8p5jggw7z3)

[Data model: special columns](#h.3yaywsk0tdlp)

[Read and write operations](#h.5onks27ib5z)

[Write request](#h.ne4phwq0tis7)

[Flushing](#h.d6n3hmmo2lxw)

[Bloom Filters](#h.6jzxxm37c1zl)

[Read request](#h.kawej8gj1qit)

[Consistency levels](#h.i3cn06131i6a)

[Quorums](#h.jm5js0nbtmdu)

[One](#h.yqxoobdmtg2z)

[ALL, ANY](#h.126x9iprsco6)

## 1- Cap Theorem

C: **Consistency**, A: **Availability**, P: **Partition** **tolerance**

We can have just two of them at the same time.

CA in particular cannot be achieved because if we don’t allow partitions in the network and we require Consistency means that we have to wait until the network is available again, so we are relaxing Availability similarly the opposite.

CP the system is consistent and partition tolerance it just wait until all nodes are back online (ex. Google,HBase,Zookeeper)

AP the system alway replies also if some nodes are not available but data consistency cannot be guaranteed, replication can reduce inconsistencies but cannot ensure C. (ex. Cassandra, Amazon Dynamo).

## 2- Amazon Dynamo

### Intro

Amazon main goals: Security, scalability, availability, performance, cost-efficiency

AWS is in AP, sacrificing consistency. AWS follows the [BASE](http://neo4j.com/blog/acid-vs-base-consistency-models-explained/) philosophy this means that the system does not have transactions (operations execute atomically) because it requires much more computation and is it doesn’t require strong consistency. Of course this behavior is for most of services that doesn’t requires consistency for billing applications is an CP system.

Dynamo is an highly-available storage system, it favors availability over consistency under failures.

Structure:

It stores data using hash tables (API: put(key,value), get(key)), this allows easy access to data but does’t support different queries. It’s more handy with small objects (<1Mb)

Characteristics:

Scalable, low latency and available especially for writes because at read time the mechanism that tries to achieve consistency starts

### Main components explanation

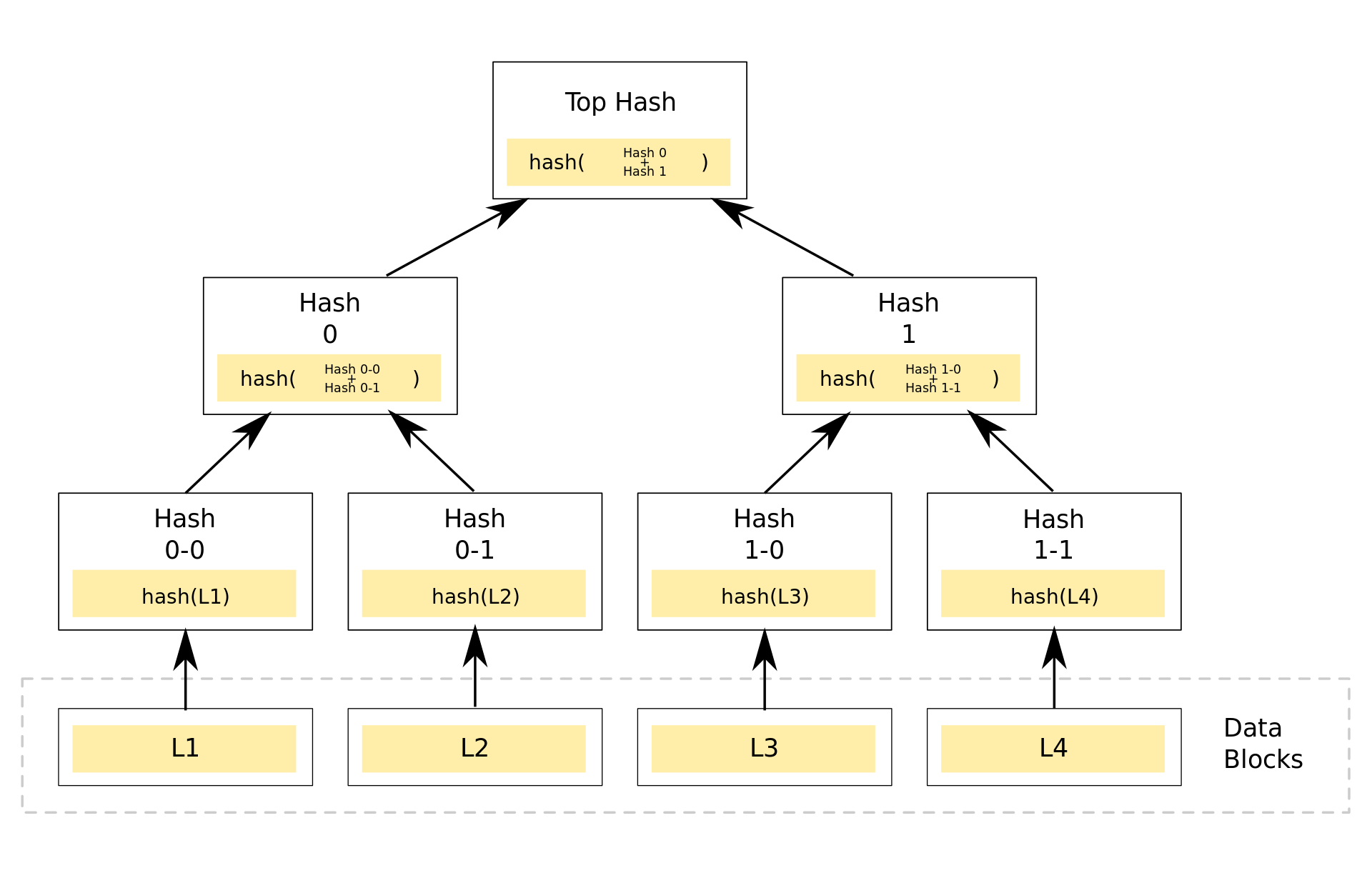
* [**Consistent hashing**](https://en.wikipedia.org/wiki/Consistent_hashing): Consistent hashing is a special kind of hashing such that when a hash table is resized and consistent hashing is used, only keys need to be remapped on average, where K is the number of keys, and n is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped.
* **Sloppy Quorums**
* **Vector Clock**: vector clock is an algorithm used to create a logical time in a distributed system every node has an id and increment the value associated to itself when an event occur. In Dynamo I guess that to each data has a value associated that is the value of the vector clock of that particular node that indicates at which logical time the last update happened
* **Gossip-based** group membership: it’s a technique used to spread the information of working nodes, each node send information about failing nodes in this way there is not a main node that takes this responsibility and so there’s no single point of failure
* anti-entropy protocol based on **Merkle trees:** at read time data is compared using Merkle trees and in case of difference (using vector clock) a “consistent” version of it is created written back and sent back to the process that sent the request.

With those element the following characteristics are achieved:

* Load balancing and data partitioning
* Membership, fault detection
* Failure recovery
* Replica synchronization
* Overload Handling
* State transfer
* Concurrency management
* Scheduling
* Request marshalling and routing
* System monitoring
* Configuration management

### Merkle trees

It’s a method to compare two files and easily find the parts that are different. It’s a binary tree where in each node there is the hash of the two hash children's, exception for the leaf nodes that are the hash of a block of the file. Looking at the figure is easy to get how the algorithm work: starting from the Top Hash we do an inorder visit of the tree two trees comparing the hash and if is different we go to the children otherwise that branch is going to be probably equal. In this way in logarithmic complexity we can find the different blocks.



### Data Partitioning

Data partitioning is implemented using consistent hashing.

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Dynamo’s partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing, the output range of a hash function is treated as a fixed circular space or “ring” (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its “position” on the ring. Each data item identified by a key is assigned to a node by hashing the data item’s key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item’s position (), If this node doesn't exist we assign it to the node with the smallest ID. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing: instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of “virtual nodes”. A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, “tokens”) in the ring.

Using virtual nodes has the following advantages:

* If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes.
* When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
* The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

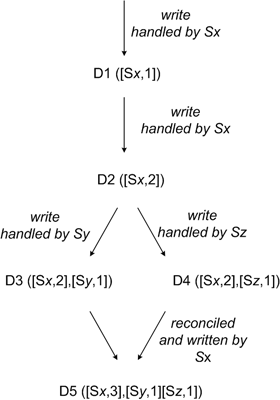
### Data replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured *“per-instance”*. Each key, *k*, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its Nth predecessor. It is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.

### Data Versioning

Each data has a verison associate to it labeled with the value of the vector clock taken at the last update version.

In this way when a fault of a node happen the data version split in different branches depending on the writes made in different node. At read time the coordinator of the key of data we are trying to read asks the other nodes which holds the replica and looking at the vector clock creates a new version of the data adding all the modification. Note that here for data we are referring to something like the shopping card and each insert or remove of an item is represented like a log of those actions.



### Put and Get operations

There are two strategies that a client can use to select a node: (1) route its request through a generic load balancer that will select a node based on load information, or (2) use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.

A node handling a read or write operation is known as the *coordinator*. Typically, this is the first among the top N nodes in the preference list. If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring. In this scenario, the node that receives the request will not coordinate it if the node is not in the top N of the requested key’s preference list. Instead, that node will forward the request to the first among the top N nodes in the preference list.

Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. When all nodes are healthy, the top N nodes in a key’s preference list are accessed. When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in *quorum systems*. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that R + W > N yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally. The coordinator then sends the new version (along with the new vector clock) to the N highest-ranked reachable nodes. If at least W-1 nodes respond then the write is considered successful.

### Hinted handoff

In case of failure amazon uses a method called sloppy quorums. In case of write if a node that was suppose to hold data fails then another node in the preference list will save it’s data and in its metadata saves the information about which was the node that was suppose to hold it; when the point recover data it’s sent to it and deleted by the node which temporary hold that.

### Membership management

Membership management it’s used to inform all nodes about each other existence and the ip position in the ring. When a change has to be issue the administrator uses a command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. The membership changes from a history because nodes can be removed and added back multiple times. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership. Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

#### Failure detection

There is no an explicit mechanism, a failure is detected when a read or write operation is sent to a node and this one doesn’t respond.

## 3- Cassandra

### Intro

Cassandra tries to combine the basic techniques of HBase and Dynamo.

* It’s a column-oriented **data model** (one key per row, columns and column family), here data is stored in order and the keys are assigned subsequently to each node this allows us to do some operation on data more easily;
* **Consistency** policy can be change with this feature we can have a AP or CP system.
* **Partitioning** can be chosen (Random or ByteOrdered partitioner) and can be changed on fly but this means the entire rewrite of all data so it has to be carefully chosen
  + **Random Partitioner:** works very similar to Dynamo, it associate to each data a key identifier and an identifier to each node. It uses consistent hashing and data monitoring in order to have load balancing between the nodes
  + **ByteOrderedPartitioner:** supports range queries, to do so row keys must be stored in order, the keys are splitted lexicographically among the nodes in the ring**.**

### Replication

Replication is asynchronous, the coordinator choose N-1 successor nodes clockwise in the ring. When the network topology is known than the replica can be located in different datacenters or racks (like HDFS)

#### Snitches

**Snitches**: a snitch determines which data centers and racks nodes belong to. Snitches inform Cassandra about the network topology so that requests are routed efficiently and allows Cassandra to distribute replicas by grouping machines into data centers and racks. Specifically, the replication strategy places the replicas based on the information provided by the new snitch. All nodes must return to the same rack and data center. Cassandra does its best not to have more than one replica on the same rack (which is not necessarily a physical location).

### File system

Cassandra by default stores it’s data on **HDFS** (but another FS can be configured). The problem with HDFS is that it doesn’t allows to delete or modify data, it just allows to append. So cassandra appends new info and at read time it just take the newer one. Then, an housekeeper is in charge to clean up data it reads it look if sono info was appended and in this case deletes the file and writes back just the newer one.

### Data model: special columns

* Counter columns: are used to store counters/timestamps associated to a row
* Expiring columns: specify a time to live after which data is deleted
* Super columns: is a multiple column but on just one lookup value (ex Address->(City,Zip code,...))
* SSTables: sorted string table is an immutable data file to which Cassandra writes memtables periodically. SSTables are stored on disk sequentially and maintained for each Cassandra table.

### Read and write operations

I/O request are proxy based: routed by a coordinator which routes the request to any replica.

The proxy node is in charge to handle the interaction between Cassandra and the client determine which are the replica nodes.

#### Write request

Is similar to HBase:

* Write commit log,
* write in memory data structure (memtable)
* write is considered SUCCESSFUL
* writes are flushed to disk in SSTable

##### Flushing

The process of turning a Memtable into a SSTable is called **flushing**. You can manually trigger flush via jmx (e.g. with bin/nodetool), which you may want to do before restarting nodes since it will reduce CommitLog replay time. Memtables are sorted by key and then written out sequentially. Thus, writes are extremely fast, costing only a commitlog append and an amortized sequential write for the flush.

Once flushed, SSTable files are immutable; no further writes may be done. So, on the read path, the server must (potentially, although it uses tricks like bloom filters to avoid doing so unnecessarily) combine row fragments from all the SSTables on disk, as well as any unflushed Memtables, to produce the requested data.

To bound the number of SSTable files that must be consulted on reads, and to reclaim space taken by unused data, Cassandra performs compactions: merging multiple old SSTable files into a single new one. Compaction strategies are pluggable; out of the box are provided SizeTieredCompactionStrategy, which combines sstables of similar sizes, and LeveledCompactionStrategy, which sorts sstables into a hierarchy of levels, each an order of magnitude larger than the previous. As a rule of thumb, SizeTiered is better for write-intensive workloads, and Leveled better for read-intensive.

(For those familiar with other LSM implementations, it's worth noting that Cassandra can remove tombstones without a "major" compaction combining all sstables into a single file.)

Since the input SSTables are all sorted by key (technically, by token), merging in a compaction can be done efficiently, again requiring no random i/o. Even so, compaction can be a fairly heavyweight operation.

##### Bloom Filters

Bloom Filters are used, in SSTable, to combine row data from multiple sources and to check if a given key exists in the SSTable before reading from the disk.

A [**Bloom filter**](https://en.wikipedia.org/wiki/Bloom_filter) is a space-efficient probabilistic data structure, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not, thus a Bloom filter has a 100% recall rate. In other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

#### Read request

The behaviour of a read it’s similar to Dynamo. The proxy checks for inconsistency and if any resolves them and write the new data back to disk. This is done in background after the read requests has been served to the client. The number of replicas used depends on the setup it can be the closest one or all replicas and the proxy waits for a quorum.

When a node receives a read request:

* Row must be combined from all SSTables on that node
* Data not yet flushed to SSTables, i.e. stored in memtables, must be considered as well

→ This produces the requested data

In order to have high performance we use row-level column index and bloom filters.

### Consistency levels

In Cassandra consistency can be reduced in order to increase the availability and read and write consistency level can be independent.

Given N replicas in the preference list. Write request:

* all N replicas are contacted
* Ends when W respond (i.e. acknowledgment) Read request: only R replicas are contacted
* This is optimistic, may need to contact all N replicas

Choices of W and R define consistency level

Dynamo: W + R > N (recall extended preference list + sloppy quorum)

Cassandra: W + R > N not mandatory

#### Quorums

QUORUM

* W = floor(N/2 + 1): A write is written to the commit log and memtable on a quorum of W replicas
* R = floor(N/2 + 1): Read returns the record with the most recent timestamp, once a quorum of size R has responded

LOCAL\_QUORUM: Restricted to a local datacenter

EACH\_QUORUM: QUORUM invariant must be satisfied across datacenters

#### One

* W=1: one replica must write to commit log e memtable
* R=1: return the response from the closest replica and the read repair runs in background to make other replica consistent

#### ALL, ANY

* W=N: all replica must acknowledge
* R=N: returns the record with the most recent timestamp across all replica

ANY: additional consistency for writes uses hinted handoff in order to complete a write also is all replica are down