

# 2

## Cassandra Architecture

This chapter aims to set you into a perspective where you can see the evolution of the NoSQL paradigm. It starts with a discussion of common problems that an average developer faces when the application starts to scale up and software components cannot keep up with it. Then, we'll see what can be assumed as a thumb rule in the NoSQL world: the CAP theorem that says to choose any two out of consistency, availability, and partition tolerance. As we discuss, we realize how important it is to serve the customers (availability) than to be correct (consistency) all the time. However, we cannot afford to be wrong (inconsistent) for a long time. The customers wouldn't like to see that the items are in stock, but the checkout is failing. Cassandra comes into the picture with its tunable consistency.

We take a quick peep into all the actions that go on when a read or mutate happens. This leaves us with lots of fancy terms. Next, we move on to see these terms in full glory with explanation as we discuss various parts of the Cassandra design. We will also see how close yet how far Cassandra is when compared with its precursors and inspiration databases, such as Google's BigTable and Amazon's Dynamo. We happen to meet with some of the modern and efficient data structures, such as Bloom filters and Merkle tree, and algorithms, such as gossip protocol, phi accrual failure detectors, and log-structured merge-trees.

### Problems in the RDBMS world

RDBMS is a great approach. It keeps data consistent, is good for OLTP ([http://en.wikipedia.org/wiki/Online\\_transaction\\_processing](http://en.wikipedia.org/wiki/Online_transaction_processing)), provides access to good grammar, and manipulates data supported by all the popular programming languages. It was tremendously successful for the last 40 years (the relational data model in its first avatar: Codd, E.F. (1970), *A Relational Model of Data for Large Shared Data Banks*). However, in the early 2000s, big companies, such as Google (BigTable, <http://research.google.com/archive/Bigtable.html>) and Amazon that have gigantic load on their databases to serve, started to feel bottlenecked with RDBMS.

If you ever used an RDBMS for a non-trivial web application, you must have faced problems, such as slow queries due to complex joins, expensive vertical scaling, and problems in horizontal scaling. Due to these problems, indexing takes a long time. At some point you choose to replicate the data, there is still some locking, and this hurts availability. That means under heavy load, locking will cause the user experience to deteriorate.

Although replication gives some relief, a busy slave may not catch up with the master (or there may be a connectivity glitch between the master and the slave). Consistency of such systems cannot be guaranteed (replication in MySQL is available at <http://www.databasejournal.com/features/mysql/article.php/3355201/Database-Replication-in-MYSQL.htm>). With growth of the application, the demand to scale the backend gets more pressing, and the developer teams decide to add a caching layer (such as Memcached) at the top of the database. This alleviates some load off the database, but now the developers need to maintain the object states at two places: in the database and the caching layer. Although some ORMs provide a built-in caching mechanism, they have their own issues: larger memory requirement, and polluted application code with mapping code. In order to achieve more from RDBMS, we start to denormalize the database to avoid joins, and keep the aggregates in the columns to avoid statistical queries.

Sharding or horizontal scaling is another way to distribute the load. Sharding in itself is a good idea, but it adds too much manual work, plus the knowledge of sharding creeps into the application code. Sharded databases make the operational tasks (backup, schema alteration, and adding index) difficult (hardships of sharding is available at <http://www.mysqlperformanceblog.com/2009/08/06/why-you-dont-want-to-shard/>).

There are ways to loosen up consistency by providing various isolation levels. But concurrency is just one part. Maintaining relational integrity, difficulties in managing data that cannot be accommodated on one machine, and difficult recovery were all making the traditional database systems hard to be accepted in the rapidly growing Big Data world. Companies needed a tool that can support hundreds of terabytes of data on the ever-failing commodity hardware reliably.

## Enter NoSQL

NoSQL is a blanket term for the databases that solve the scalability issues that are common among relational databases. This term, in its modern meaning, was first coined by Eric Evans (NoSQL naming available at [http://blog.sym-link.com/2009/10/30/nosql\\_what\\_is\\_a\\_name.html](http://blog.sym-link.com/2009/10/30/nosql_what_is_a_name.html)). It should not be confused with the database named NoSQL (NoSQL: the database available at [http://www.strozzi.it/cgi-bin/CSA/tw7/I/en\\_US/nosql/Home%20Page](http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page)).

The NoSQL solutions provide scalability and high availability, but may not guarantee ACID: atomicity, consistency, isolation, and durability in transactions. Many of the NoSQL solutions including Cassandra sit on the other extreme of ACID, named BASE (Basically Available, Soft-state, Eventual consistency).

## The CAP theorem

In 2000, Eric Brewer (Wikipedia page available at [http://en.wikipedia.org/wiki/Eric\\_Brewer\\_%28scientist%29](http://en.wikipedia.org/wiki/Eric_Brewer_%28scientist%29)), in his keynote speech at the ACM Symposium, said that one cannot guarantee consistency in a distributed system. This was his conjecture based on his experience with the distributed systems. This conjecture was later formally proved by Nancy Lynch and Seth Gilbert in 2002 (*Brewer's Conjecture and the Feasibility of Consistent, Available at Partition-tolerant Web Services*, and *ACM SIGACT News, Volume 33, Issue 2 (2002), page 51 to 59* available at <http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf>) It says, if we have a distributed system where data is replicated at two distinct locations and two conflicting requests arrive—one at each location—when the communication link between the two servers is broken. If the system (the cluster) has obligations to be highly available (a response, even when some components of the system are failing), one of the two responses will be inconsistent with what a system with no replication (no partitioning, single copy) would have returned. To understand it better, let us take an example to learn the terminologies. These terms will be used frequently throughout this book.

Let's say you are planning to read George Orwell's book titled *Nineteen Eighty-Four* (1984, *The Novel* available at [http://en.wikipedia.org/wiki/Nineteen\\_Eighty-Four](http://en.wikipedia.org/wiki/Nineteen_Eighty-Four)) over the Christmas vacation. A day before the holidays start, you logged in to your favorite online book store to find out that there is only one copy left. You add it to your cart. But then you realize that you need to buy something else to be eligible for free shipping. You start to browse the website for any other item that you might buy. To make the situation interesting, let's say there is another customer who is trying to buy *Nineteen Eighty-Four* at the same time.

## Consistency

In a distributed system, consistency will be defined as one that responds with the same output for the same request at the same time across all the replicas. Loosely, one can say a consistent system is one where each server returns the right response to each request.

In our book-buying example, we have only one copy of *Nineteen Eighty-Four*. So, only one of the two customers is going to get the book delivered from this store. In a consistent system, only one can check out the book from the payment page. As soon as one customer makes the payment, the number of *Nineteen Eighty-Four* books in stock will get decremented by one and one quantity of *Nineteen Eighty-Four* will be added to the order of that customer. When the second customer tries to check out his shopping cart, the system tells that the book is not available any more.

Relational databases are good for this task because they comply with the ACID properties. If both the customers make the request at the same time, one customer will have to wait till the other customer is done with the processing, and the database is made consistent. This may add a few milliseconds of wait to the customer who came later.

An eventual consistent (where consistency of data across the distributed servers may not be guaranteed immediately) database system may have showed both the customers availability of the book at the time of check-out. This will lead to a back order, and one of the customers will be paid back. This may or may not be a good policy. A large number of back orders may affect the shop's reputation and there may also be financial repercussions.

## ✓ Availability

Availability in simplest term is responsiveness. A system that's always available to serve. The funny thing about availability is that sometimes a system becomes unavailable exactly when you need it the most.

In our example, one day before Christmas, everyone is buying gifts. Millions of people are searching, adding items to their carts, buying, and applying for discount coupons. If one server goes down due to overload, the rest of the servers get even more loaded now, because the request from the dead server is getting redirected to the rest of the machines. Dominoes start to fall. Now the site is down. When it comes online again, it faces a storm of requests from all the people who are hurrying to place their order because the offer end time is even closer, or probably acting quickly before the site goes down again.

Availability is the key component for extremely loaded services. Bad availability leads to bad user experience, dissatisfied customers, and financial losses.

## Partition-tolerance

Partition-tolerance is a system that can operate during the network partition. The network will be allowed to lose arbitrarily many messages sent from one node to another. This means a cable between the two nodes is chopped, but the system still works.

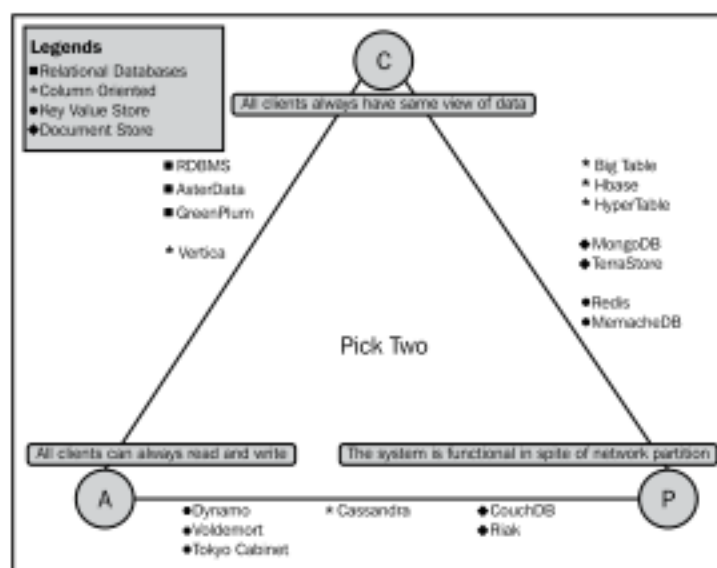


Figure 2.1. Database classification based on CAP Theorem

An example of a partition-tolerant system is a system with real-time data replication. A system where data is replicated across two datacenters, the availability will not be affected, even if a datacenter goes down.

## Significance of the CAP theorem

Once you decide to scale up, the first thing that comes to mind is vertical scaling, which means putting beefier servers with a bigger RAM, a more powerful processor, and bigger disks. For further scaling, you need to go horizontal, which means adding more servers. Once your system becomes distributed, the CAP theorem starts to play. That means, in a distributed system, you can choose only two out of consistency, availability, and partition-tolerance. So, let us see how choosing two out of the three options affect the system behavior as follows:

**CA system:** In this system, you drop partition-tolerance for consistency and availability. This happens when you put everything related to a transaction on one machine or a system that fails like an atomic unit, for example, a rack. This system will have serious problems in scaling.



**CP system:** The opposite of a CA system is a CP system. In a CP system, availability is sacrificed for consistency and partition-tolerance. What does this mean? If the system is available to serve the requests, data will be consistent. In the event of a node failure, some data will not be available. A sharded database is an example of such a system.

**AP system:** An available and partition-tolerance system is like an always-on system on risk of producing conflicting results in the event of network partition. This is good for user experience, your application stays available, and inconsistency in rare events may be alright for some use cases. In the book example, it may not be such a bad idea to back order a few unfortunate customers due to inconsistency of the system than having a lot of users to return without making any purchase because of the system's poor availability.

**Eventual consistent aka BASE system:** The AP system makes more sense when viewed from an uptime perspective – it's simple and provides good user experience. But, an inconsistent system is not good for anything, certainly not good for business. It may be acceptable that one customer for the book *Nineteen Eighty-Four* gets a back order. But if it happens more often, the users would be reluctant to use the service. It will be great if the system can fix itself (read repair) as soon as the first inconsistency is observed. Or, maybe there are processes dedicated to fix the inconsistency of a system when a partition failure is fixed or a dead node comes back to life. Such systems are called Eventual Consistent Systems.

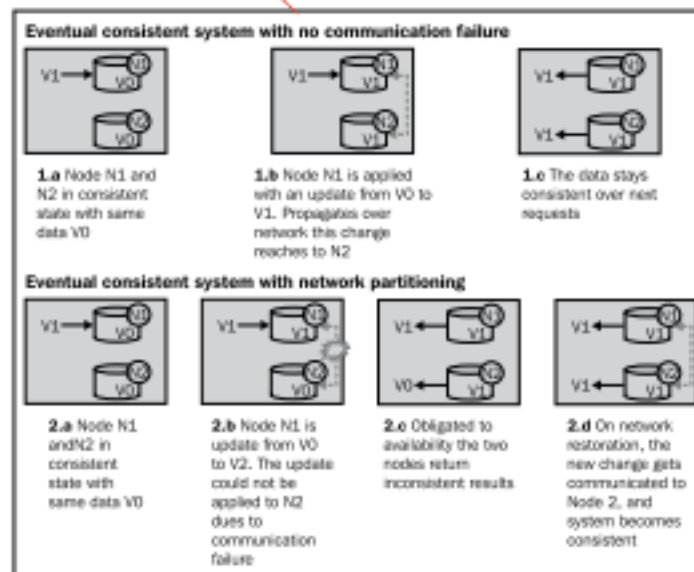


Figure 2.2: Life of an eventual consistent system

Quoting Wikipedia, "[In a distributed system] given a sufficiently long time over which no changes [in system state] are sent, all updates can be expected to propagate eventually through the system and the replicas will be consistent." (*Eventual Consistency* available at [http://en.wikipedia.org/wiki/Eventual\\_consistency](http://en.wikipedia.org/wiki/Eventual_consistency))

Eventual Consistent Systems are also called BASE, a made-up term to represent that these systems are on one end of the spectrum, which has traditional databases with the ACID properties on the opposite end.

Cassandra is one such system that provides high availability, and partition-tolerance at the cost of consistency, which is tunable. The preceding figure shows a partition-tolerant Eventual Consistent System.

## Cassandra

Cassandra is distributed, decentralized, fault tolerant, eventually consistent, linearly scalable, and a column-oriented data store. This means Cassandra is made to easily deploy over a cluster of machines located at geographically different places. There is no central master server, so no single point of failure, no bottleneck, data is replicated, and a faulty node can be replaced without any downtime. It's eventually consistent. It is linearly scalable, which means with a greater number of nodes, the requests served per second per node would not go down. Also, the total throughput of the system will increase with each node being added. And finally, it's column oriented, much like a map (or better, a map of sorted maps) or a table with flexible columns where each column is essentially a key-value pair. So, you can add columns as you go, and each row can have a different set of columns (key-value pairs). It does not provide any relational integrity. It is up to the application developer to perform relation management.

*Q* So, if Cassandra is so good at everything, why not everyone drop whatever database they are using and jump start with Cassandra? This is a natural question. We'll discuss in a later chapter what Cassandra is not good at, but there may be several obvious reasons, such as not everyone needs a super-scalable data store, and some are good with rather slow, but cozy RDBMS tools. Some applications require strong ACID compliance, such as a booking system. If you are a person who goes by statistics, you'd ask how Cassandra fares with other existing data stores. TilmannRabl et al in their paper, *Solving Big Data Challenges for Enterprise Application Performance Management* ([http://vladb.org/pvladb/vol5/p1724\\_tilmannrabl\\_vldb2012.pdf](http://vladb.org/pvladb/vol5/p1724_tilmannrabl_vldb2012.pdf)), told that, "In terms of scalability, there is a clear winner throughout our experiments. Cassandra achieves the highest throughput for the maximum number of nodes in all experiments with a linear increasing throughput from one to 12 nodes. This comes at the price of a high write and read latency. Cassandra's performance is best for high insertion rates."

If you go through the paper, Cassandra wins in almost all the criteria. Equipped with proven concepts of distributed computing, made to reliably serve from commodity servers, and simple and easy maintenance, Cassandra is one of the most scalable, fastest, and very robust NoSQL database. So, the next natural question is what makes Cassandra so blazing fast? Let us dive deeper into the Cassandra architecture.

## Cassandra architecture

Cassandra is a relative latecomer in the distributed data-store war. It takes advantage of two proven and closely similar data-store mechanisms, namely Google BigTable, a distributed storage system for structured data, 2006 ([http://static.googleusercontent.com/external\\_content/untrusted\\_dlcp/research.google.com/en//archive/bigtable-osdi06.pdf](http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//archive/bigtable-osdi06.pdf) [2006]), and Amazon Dynamo, Amazon's highly available key-value store, 2007 (<http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/decandia07dynamo.pdf> [2007]).

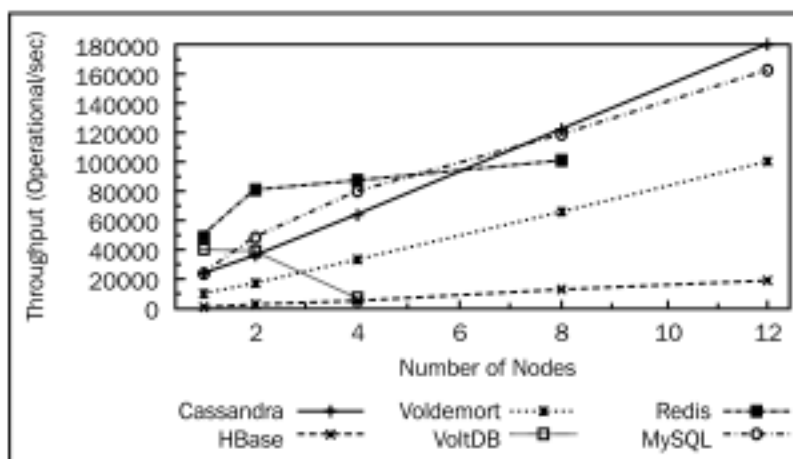


Figure 2.3: Read throughputs shows linear scaling of Cassandra

Like BigTable, it has tabular data presentation. It is not tabular in the strictest sense. It is rather a dictionary-like structure where each entry holds another sorted dictionary/map. This model is more powerful than the usual key-value store and it is named as column family. The properties such as Eventual Consistency and decentralization are taken from Dynamo.



We'll discuss column family in detail in a later chapter. For now, assume a column family as a giant spreadsheet, such as MS Excel. But unlike spreadsheets, each row is identified by a row key with a number (token), and unlike spreadsheets, each cell can have its own unique name within the row. The columns in the rows are sorted by this unique column name. Also, since the number of rows is allowed to be very large ( $1.7 \times 10^{38}$ ), we distribute the rows uniformly across all the available machines by dividing the rows in equal token groups. These rows create a **Keyspace**. Keyspace is a set of all the tokens (row IDs).

## ✓ Ring representation

Cassandra cluster is denoted as a ring. The idea behind this representation is to show token distribution. Let's take an example. Assume that there is a partitioner that generates tokens from zero to 127 and you have four Cassandra machines to create a cluster. To allocate equal load, we need to assign each of the four nodes to bear an equal number of tokens. So, the first machine will be responsible for tokens one to 32, the second will hold 33 to 64, the third, 65 to 96, and the fourth, 97 to 127 and 0. If you mark each node with the maximum token number that it can hold, the cluster looks like a ring. (Figure 2.22) Partitioner is the hash function that determines the range of possible row keys. Cassandra uses a partitioner to calculate the token equivalent to a row key (row ID).

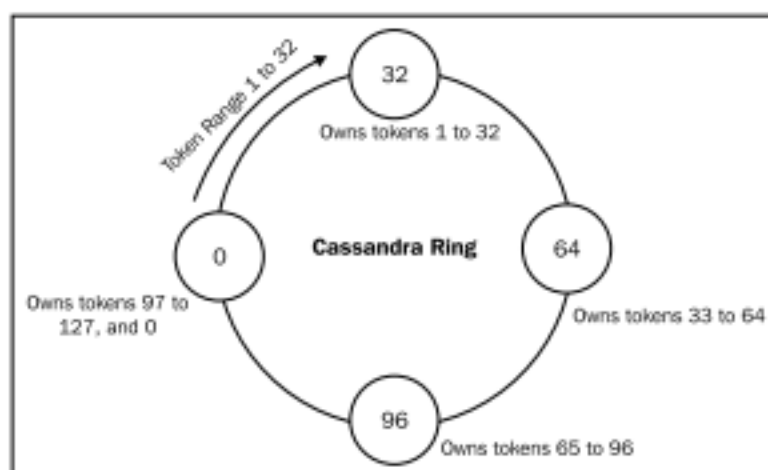


Figure 2.4: Token ownership and distribution in a balanced Cassandra ring

When you start to configure Cassandra, one thing that you may want to set is the maximum token number that a particular machine could hold. This property can be set in the `Cassandra.yaml` file as `initial_token`. One thing that may confuse a beginner is that the value of the initial token is what the last token owns. Be aware that nodes can be rebalanced and these tokens can be changed as the new nodes join or old nodes get discarded. This is the initial token because this is just the initial value, and it may be changed later.

## ✓ How Cassandra works

Diving into various components of Cassandra without having a context is really a frustrating experience. It does not make sense why you are studying **SSTable**, **MemTable**, and **Log Structured Merge (LSM)** tree without being able to see how they fit into functionality and performance guarantees that Cassandra gives. So, first, we will see Cassandra's write and read mechanism. It is possible that some of the terms that we encounter during this discussion may not be immediately understandable. The terms are explained in detail later in the chapter. A rough overview of the Cassandra components is as shown in the following figure:

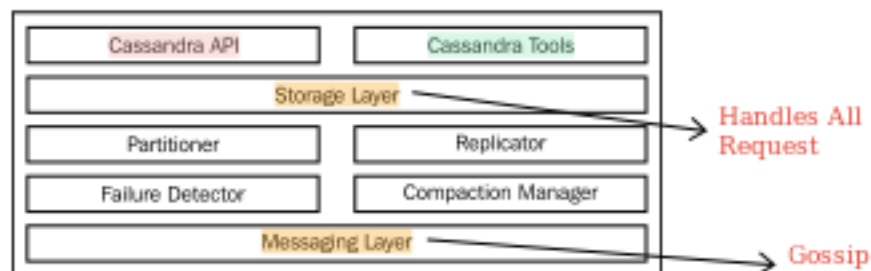


Figure 2.5: Main components of the Cassandra service

The main class of **Storage Layer** is `StorageProxy`. It handles all the requests. **Messaging Layer** is responsible for internode communications like gossip. Apart from this, process-level structures keep a rough idea about the actual data containers and where they live. There are four data buckets that you need to know. **MemTable** is a hash table-like structure that stays in memory. It contains actual column data. **SSTable** is the disk version of MemTables. When MemTables are full, SSTables are persisted to the hard disk. **Bloom filters** is a probabilistic data structure that lives in memory. It helps Cassandra to quickly detect which SSTable *does not* have the requested data. **CommitLog** is the usual commit log that contains all the mutations that are to be applied. It lives on the disk and helps to replay uncommitted changes.

With this primer, we can start looking into how write and read works in Cassandra. We will see more explanation later.

## Write in action

To write, clients need to connect to *any* of the Cassandra nodes and send a write request. This node is called as the **coordinator node**. When a node in Cassandra cluster receives a write request, it delegates it to a service called *StorageProxy*. This node may or may not be the right place to write the data to. The task of *StorageProxy* is to get the nodes (all the replicas) that are responsible to hold the data that is going to be written. It utilizes a replication strategy to do that. Once the replica nodes are identified, it sends the *RowMutation* message to them, the node waits for replies from these nodes, but it does not wait for all the replies to come. It only waits for as many responses as are enough to satisfy the client's minimum number of successful writes defined by *ConsistencyLevel*. So, the following figure and steps after that show all that can happen during a write mechanism:

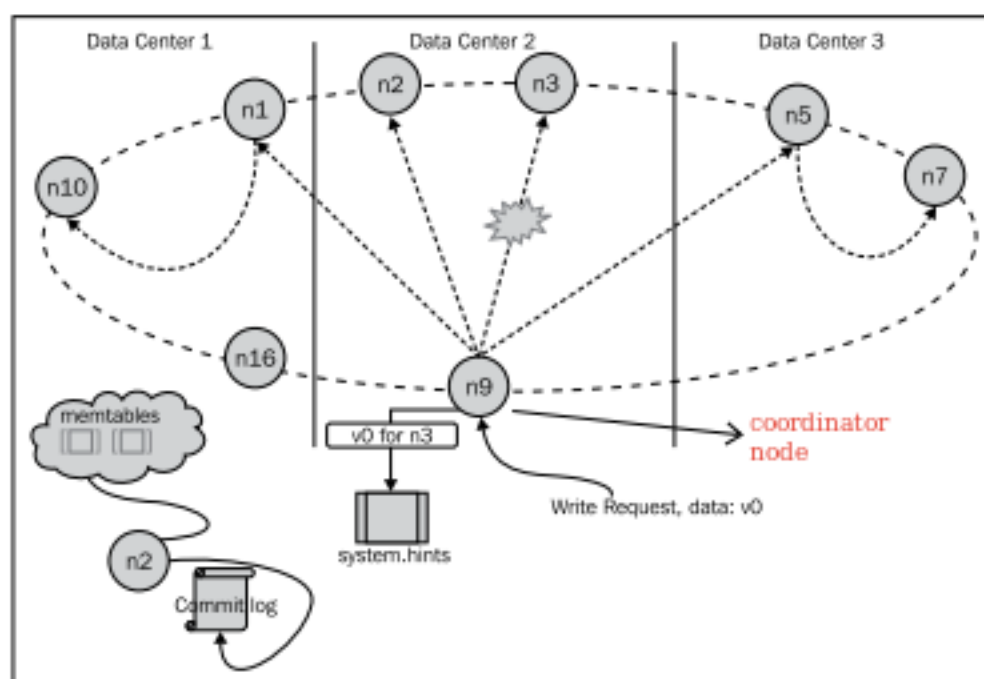


Figure 2.6: A simplistic representation of the write mechanism. The figure on the left represents the node-local activities on receipt of the write request

1. **If *FailureDetector* detects that there aren't enough live nodes to satisfy *ConsistencyLevel*, the request fails.**
2. **If *FailureDetector* gives a green signal, but writes time-out after the request is sent due to infrastructure problems or due to extreme load, *StorageProxy* writes a local *hint* to replay when the failed nodes come back to life. This is called **hinted handoff**.**



One might think that hinted handoff may be responsible for Cassandra's eventual consistency. But it's not entirely true. If the coordinator node gets shut down or dies due to hardware failure and *hints* on this machine cannot be forwarded, eventual consistency will not occur. The **Anti-entropy** mechanism is responsible for consistency rather than hinted handoff. Anti-entropy makes sure that all replicas are in sync.

3. If the replica nodes are distributed across datacenters, it will be a bad idea to send individual messages to all the replicas in other datacenters. It rather sends the message to one replica in each datacenter with a header instructing it to forward the request to other replica nodes in that datacenter.
4. Now, the data is received by the node that should actually store that data. The data first gets appended to CommitLog, and pushed to a MemTable for the appropriate column family in the memory.
5. When MemTable gets full, it gets flushed to the disk in a sorted structure named SSTable. With lots of flushes, the disk gets plenty of SSTables. To manage SSTables, a **compaction process** runs. This process merges data from smaller SSTables to one big sorted file.

## Read in action

Similar to a write case, when StorageProxy of the node that a client is connected to gets the request, it gets a list of nodes containing this key based on **Replication Strategy**. StorageProxy then sorts the nodes based on their proximity to itself. The proximity is determined by the **Snitch** function that is set up for this cluster. Basically, there are the following types of Snitch:

- **SimpleSnitch**: A closer node is the one that comes first when moving clockwise in the ring. (A ring is when all the machines in the cluster are placed in a circular fashion with each having a token number. When you walk clockwise, the token value increases. At the end, it snaps back to the first node.)

- AbstractNetworkTopologySnitch: Implementation of the Snitch function works like this: nodes on the same rack are closest. The nodes in the same datacenter but in different rack are closer than those in other datacenters, but farther than the nodes in the same rack. Nodes in different datacenters are the farthest. To a node, the nearest node will be the one on the same rack. If there is no node on the same rack, the nearest node will be the one that lives in the same datacenter, but on a different rack. If there is no node in the datacenter, any nearest neighbor will be the one in the other datacenter.
- DynamicSnitch: This Snitch determines closeness based on recent performance delivered by a node. So, a quick-responding node is perceived closer than a slower one, irrespective of their location closeness or closeness in the ring. This is done to avoid overloading a slow-performing node.

Now that we have the list of nodes that have desired row keys, it's time to pull data from them. The coordinator node (the one that the client is connected to) sends a command to the closest node to perform read (we'll discuss local read in a minute) and return the data. Now, based on `ConsistencyLevel`, other nodes will send a command to perform a read operation and send just the digest of the result. If we have **Read Repair** (discussed later) enabled, the remaining replica nodes will be sent a message to compute the digest of the command response.

Let's take an example: say you have five nodes containing a row key  $K$  (that is, **replication factor (RF)** equals 5). Your read `ConsistencyLevel` is three. Then the closest of the five nodes will be asked for the data. And the second and third closest nodes will be asked to return the digest. We still have two left to be queried. If read Repair is not enabled, they will not be touched for this request. Otherwise, these two will be asked to compute digest. The request to the last two nodes is done in the background, after returning the result. This updates all the nodes with the most recent value, making all replicas consistent. So, basically, in all scenarios, you will have a maximum one wrong response. But with correct read and write consistency levels, we can guarantee an up-to-date response all the time.



Let's see what goes within a node. Take a simple case of a read request looking for a single column within a single row. First, the attempt is made to read from MemTable, which is rapid-fast since there exists only one copy of data. This is the fastest retrieval. If the data is not found there, Cassandra looks into SSTable. Now, remember from our earlier discussion that we flush MemTables to disk as SSTables and later when compaction mechanism wakes up, it merges those SSTables. So, our data can be in multiple SSTables.

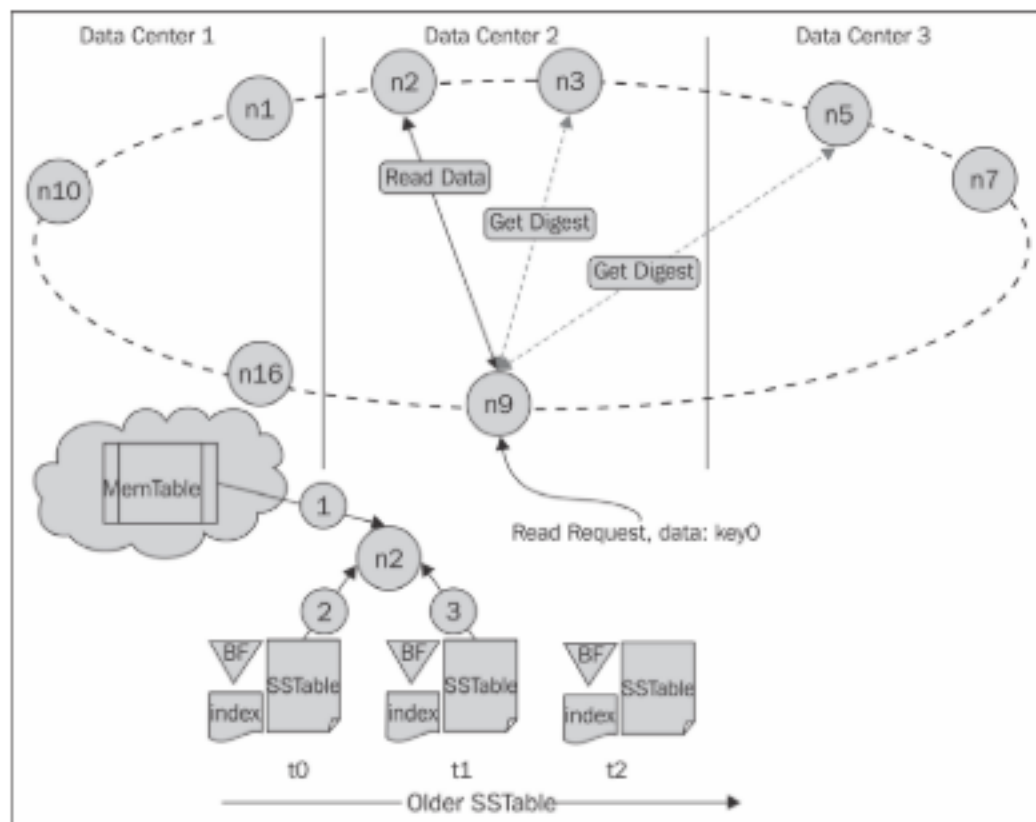


Figure 2.7: A simplified representation of the read mechanism. The bottom image shows processing on the read node. Numbers in circles shows the order of the event. BF stands for Bloom Filter

Each SSTable is associated with its Bloom Filter built on the row keys in the SSTable. Bloom Filters are kept in memory, and used to detect if an SSTable may contain (false positive) the row data. Now, we have the SSTables that may contain the row key. The SSTables get sorted in reverse chronological order (latest first).

Apart from Bloom Filter for row keys, there exists one Bloom Filter for each row in the SSTable. This secondary Bloom Filter is created to detect whether the requested column names exist in the SSTable. Now, Cassandra will take SSTables one by one from younger to older. And use the index file to locate the offset for each column value for that row key and the Bloom filter associated with the row (built on the column name). On Bloom filter being positive for the requested column, it looks into the SSTable file to read the column value. Note that we may have a column value in other yet-to-be-read SSTables, but that does not matter, because we are reading the most recent SSTables first, and any value that was written earlier to it does not matter. So, the value gets returned as soon as the first column in the most recent SSTable is allocated.

## Components of Cassandra

We have gone through how read and write takes place in highly distributed Cassandra clusters. It's time to look into individual components of it a little deeper.

### ✓ Messaging service

Messaging service is the mechanism that manages internode socket communication in a ring. Communications, for example, gossip, read, read digest, write, and so on, processed via a messaging service, can be assumed as a gateway messaging server running at each node.

To communicate, each node creates two socket connections per node. This implies that if you have 101 nodes, there will be 200 open sockets on each node to handle communication with other nodes. The messages contain a `verb` handler within them that basically tells the receiving node a couple of things: how to deserialize the payload message and what handler to execute for this particular message. The execution is done by the `verb` handlers (sort of an event handler). The singleton that orchestrates the messaging service mechanism is `org.apache.cassandra.net.MessagingService`.

## Gossip

Cassandra uses the gossip protocol for internode communication. As the name suggests, the protocol spreads information in the same way an office rumor does. It can also be compared to a virus spread. There is no central broadcaster, but the information (virus) gets transferred to the whole population. It's a way for nodes to build the global map of the system with a small number of local interactions.

Cassandra uses gossip to find out the state and location of other nodes in the ring (cluster). The gossip process runs every second and exchanges information with at the most three other nodes in the cluster. Nodes exchange information about themselves and other nodes that they come to know about via some other gossip session. This causes all the nodes to eventually know about all the other nodes. Like everything else in Cassandra, gossip messages have a version number associated with it. So, whenever two nodes gossip, the older information about a node gets overwritten with a newer one. Cassandra uses an Anti-entropy version of gossip protocol that utilizes **Merkle trees** (discussed later) to repair unread data.

Implementation-wise the gossip task is handled by the `org.apache.cassandra.gms.Gossiper` class. Gossiper maintains a list of live and dead endpoints (the unreachable endpoints). At every one-second interval, this module starts a gossip round with a randomly chosen node. A full round of gossip consists of three messages. A node **X** sends a `syn` message to a node **Y** to initiate gossip. **Y**, on receipt of this `syn` message, sends an `ack` message back to **X**. To reply to this `ack` message, **X** sends an `ack2` message to **Y** completing a full message round.

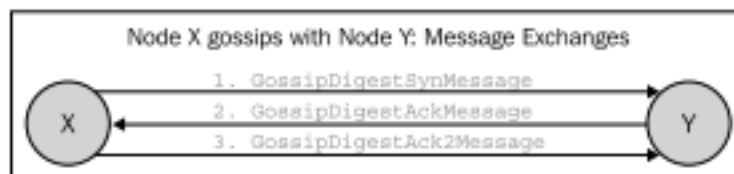


Figure 2.8: Two nodes gossiping

The Gossiper module is linked to failure detection. The module on hearing one of these messages updates `FailureDetector` with the *liveness* information that it has gained. If it hears a `GossipShutdownMessage`, the module marks the remote node as dead in `FailureDetector`.

The node to be gossiped with is chosen based on the following rules:

- Gossip to a random live endpoint
- Gossip to a random unreachable endpoint
- If the node in #1 was not a seed node or the number of live nodes is less than the number of seeds, gossip to a random seed



### Seed Node

Seed nodes are the nodes that are first contacted by a newly joining node when they first start up. Seed nodes help the newly started node to discover other nodes in the cluster. It is suggested to have more than one seed node in a cluster. Seed node is nothing like a master in a master-slave mechanism. It is just another node that helps newly joining nodes to bootstrap gossip protocol. Seeds, hence, are not a **single point of failure** (SPOF) and neither have any other purpose that makes them superior.

## Failure detection

Failure detection is one of the fundamental features of any robust and distributed system. A good failure detection mechanism implementation makes a fault-tolerant system such as Cassandra. The failure detector that Cassandra uses is a variation of *The  $\phi$  accrual failure detector (2004)* by Xavier Défago et al. (The phi accrual detector research paper is available at <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.3350>.)

The idea behind a FailureDetector is to detect a communication failure and take appropriate actions based on the state of the remote node. Unlike traditional failure detectors, phi accrual failure detector does not emit a Boolean alive or dead (true or false, trust or suspect) value. Instead, it gives a continuous value to the application and the application is left to decide the level of severity and act accordingly. This continuous suspect value is called phi ( $\phi$ ). So, how does  $\phi$  get calculated?

Let's say we are observing a heartbeat sent from a process on a remote machine. Assume that the latest heartbeat arrived at time  $T_{last}$ , current time  $t_{now}$ , and  $P_{later}(t)$  be the probability that the heartbeat will arrive  $t$  time unit later than the last heartbeat. Then  $\phi$  can be calculated as follows:

$$\phi(t_{now}) = -\log_{10}\{P_{later}(t_{now} - T_{last})\}$$

Let's observe this formula informally using common sense. On a sunny day, when everything is fine and heartbeat is at a constant interval  $\Delta t$ . The probability of the next heartbeat will keep increasing towards 1 as  $(t_{now} - T_{last}) \Delta t$ . So, the value of  $\phi$  will go up. If a heartbeat is not received in  $\Delta t$ , the more we depart away from  $\Delta t$ , the chances are that something has gone wrong.

The lower the value of  $P_{\text{later}}$  becomes, and the value of  $\phi$  keeps on increasing as shown in the following figure:

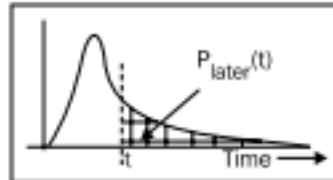


Figure 2.9: The curve shows a heartbeat arrival distribution estimate based on past samples. It is used to calculate the value of  $\phi$  based on the last arrival,  $T_{\text{last}}$  and  $t_{\text{now}}$ .

One may question where we send a heartbeat in Cassandra. Gossip has it.

**Gossip and Failure Detection:** During gossip sessions, each node maintains a list of the arrival timestamps of gossip messages from other nodes. This list is basically a sliding window, which, in turn, is used to calculate  $P_{\text{later}}$ . One may set the sensitivity of  $\phi_{\text{thres}}$  threshold.

The best explanation of failure detector can be found in Cassandra research paper (<http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf>):

*Given some threshold  $\phi$ , and assuming that we decide to suspect a node A when  $\phi = 1$ , then the likelihood that we will make a mistake (that is, the decision will be contradicted in the future by the reception of a late heartbeat) is about 10%. The likelihood is about 1% with  $\phi = 2$ , 0.1% with  $\phi = 3$ , and so on. Every node in the system maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster. The distribution of these inter-arrival times is determined and  $\phi$  is calculated.*

## Partitioner

Cassandra is a distributed database management system. This means it takes a single logical database and distributes it over one or more machines in the database cluster. So, when you insert some data in Cassandra, it assigns each row to a row key; and based on that row key, Cassandra assigns that row to one of the nodes that's responsible for managing it.

Let's try to understand this. Cassandra inherits the data model from Google's BigTable (BigTable research paper can be found at <http://research.google.com/archive/bigtable.html>). This means we can roughly assume that the data is stored in some sort of a table that has an unlimited number of columns (not unlimited, Cassandra limits the maximum number of columns to be two billion) with rows binded with a *unique* key, namely, row key.



Now, your terabytes of data on one machine will be restrictive from multiple points of views. One is disk space, another being limited parallel processing, and if not duplicated, a source of single point of failure. What Cassandra does is, it defines some rules to slice data across rows and assigns which node in the cluster is responsible for holding which slice. This task is done by a partitioner. There are several types of partitioners to choose from. We'll discuss them in detail in *Chapter 4, Deploying a Cluster*, under the Partitioners section. In short, Cassandra (as of Version 1.2) offers three partitioners as follows:

- **RandomPartitioner:** It uses MD5 hashing to distribute data across the cluster. Cassandra 1.1.x and precursors have this as the default partitioner.
- **Murmur3Partitioner:** It uses Murmur hash to distribute the data. It performs better than **RandomPartitioner**. It is the default partitioner from Cassandra Version 1.2 onwards.
- **ByteOrderPartitioner:** Keeps keys distributed across the cluster by key bytes. This is an ordered distribution, so the rows are stored in lexical order. This distribution is commonly discouraged because it may cause a hotspot.

One of the key benefits of partitioning data is that it allows the cluster to grow incrementally. What any partitioning algorithm does is it gives a consistent divisibility of data across all available nodes. The *key* that a node is assigned to by the partitioner also determines the node's *position* in the ring. Since partitioning is a global setting, any node in the cluster can calculate which nodes to look for in a given row key. This ability to calculate data-holding nodes without knowing anything other than the row key, enables any node to calculate what node to forward requests to. This makes the node selection process a single-hop mechanism.

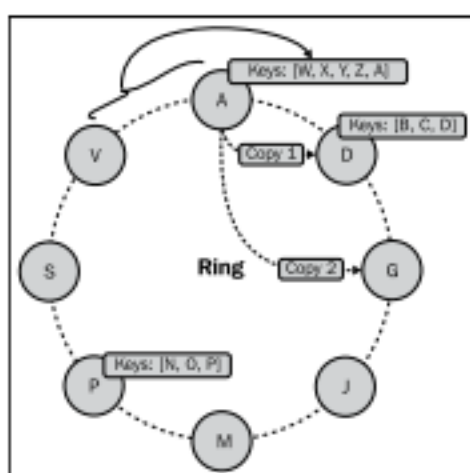


Figure 2.10: A Cassandra ring with alphabetical partitioner shows keys owned by the nodes and data replication

Another benefit of good partitioning is that the addition or removal of a node only affects the neighbors of the arriving or departing node. How so? *Figure 2.10* shows what a Dynamo (<http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/deandia07dynamo.pdf>) or a Cassandra cluster looks like; it looks like a ring. In this particular figure, each node is assigned with a letter as its token ID. So, the partitioner here is something that slices row keys based on their alphabetical ordering. When a data arrives to a node, the row key tells its position in the cluster. Then, while walking clockwise in the cluster, the first node with a position (token ID) larger than or equal to the row key's position (row key converted to a token ID) becomes responsible for that data. This way, each node in the cluster becomes responsible for the region in the cluster between it (inclusive) and its previous node (exclusive). So, node **D** will keep keys starting with **B**, **C**, and **D**.

Another frequently used terminology for a Cassandra cluster is **ring**. This is because the last node wraps around the first one, thus making the system look like a ring. In the preceding figure, the first node **A** is responsible for wrapping up from the last node **V**. So, it holds the data from **W** to **Z** to **A**. If a node **D** disappears from the system, the only node that gets affected is the node **G**, which now has to carry data from **B** to **G**. If we add a node **X** (between **V** and **A**), node **A** will offload some of its rows to the new node **X**. Particularly, **A** will stream out rows starting from **W** to **X** to node **X** and will become responsible for data from **Y** to **A**.

Now that we have observed that partitioning has such a drastic effect on the data movement and distribution, one may think that a bad partitioner can lead to uneven data distribution. In fact, our example ring in the previous paragraph might be a bad partitioner. For a data set, where terms with a specific starting letter has very high population than the terms with other letters, the ring will be lopsided. A good partitioner is one that is quick to calculate the position from the row key and distributes the row keys evenly; something like a partitioner based on consistent hashing algorithm.

## Replication

Cassandra runs on commodity hardware, and works reliably in network partitions. However, this comes with a cost: replication. To avoid data inaccessibility in case a node goes down or becomes unavailable, one must replicate data to more than one node. Replication brings features such as fault tolerance and no single point of failure to the system. Cassandra provides more than one strategy to replicate the data, and one can configure the replication factor while creating keyspace. It will be discussed in detail in *Chapter xx*.

Replication is tightly binded to **consistency level (CL)**. Consistency level can be thought of as an answer to the question: how many replicas must respond positively to declare a successful operation? If you have read consistency level three, that means a client will be returned a successful read as soon as three replicas respond with the data. Same goes for write. For write consistency three, at least three replicas must respond that the write to them was successful. Obviously, replication factor *must* be greater than any consistency level, else there will never be enough replicas to write to or read from successfully.



Do not confuse replication factor with the number of nodes in the system. Replication factor is the number of copies of a data. The number of nodes just affects how much data a node will hold based on the configured partitioner.

Replication should be thought of as an added redundancy. One should never have a replication factor of 1 in their production environment. If you think having multiple writes to different replicas will slow down the writes, you can set up a favorable consistency level. Cassandra offers a set of consistency levels including CL ZERO for fire and forget, and CL ALL for all replica success. This is where so-called **tunable** consistency of Cassandra is. The following table shows all the consistency levels:

WRITE		READ	
Consistency level	Meaning	Consistency level	Meaning
ZERO	Fire and forget		
ANY	Success on hinted handoff write		
ONE	First replica returned successfully	ONE	First replica returned successfully
QUORUM	$N/2 + 1$ replica success	QUORUM	$N/2 + 1$ replica success
ALL	All replica success	ALL	All replica success

**The notorious  $R+W > N$  inequality:** Imagine you have the replication factor as 3. That means your data will be stored in three nodes. They may or may not be consistent. You have write consistency level as one, and read consistency level as one. A write happens to the first replica node (N1) and is returned to the user. This means that while a read happens, unfortunately that read lands on a second replica (N2). Since consistency level is equal to one, it will just read this node and return the value. This will be inconsistent as the write hasn't propagated yet. This will be made consistent after the read happens as a part of read repair. But the first read is wrong.

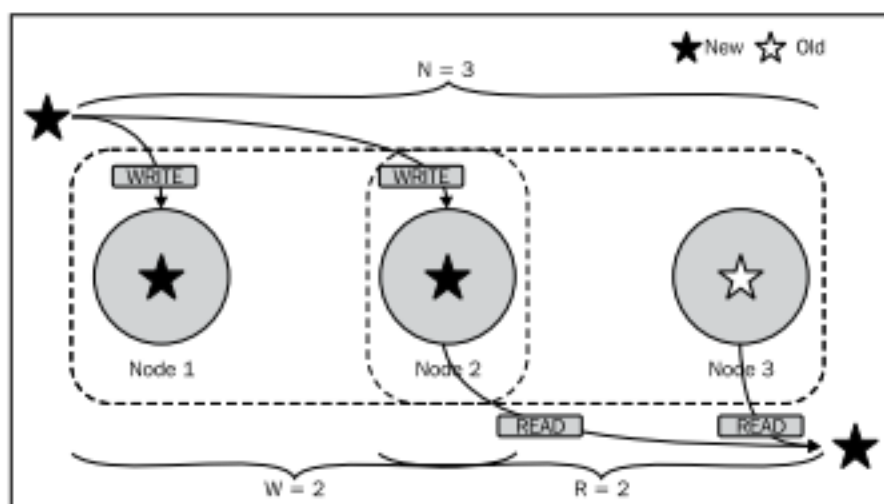


Figure 2.11: Read and write on a  $R + W > N$  system

The concept of weak and strong consistency comes here. Weak consistency is the one where read repair takes place after returning the result to the client. And strong consistency is the one where repair happens before returning the result. Basically, a weak consistency is the one that may return inconsistent results. If you have an  $N$  replica, to ensure that your reads always result in the latest value, you must write and read from as many nodes that ensure at least one node overlaps. So, if you write to  $W$  nodes and read from  $R$  nodes such that  $R+W > N$ , there must be at least one node that is common in both read and write. And that will ensure that you have the latest data. See Figure 2.11. So, `ZERO` and `ANY` consistency levels are weak consistency. `ALL` is strong. `ONE` for read and `ALL` for write or vice versa will make a strongly consistent system. A system with `QUORUM` for both read and write is a strongly consistent system. Again, the idea is to make sure that between the reads and the writes at least one node overlaps. While we are on this topic, it may be worth noticing that the higher the consistency level, the slower the operation. So, if you want a superfast write and not-so-fast read and you also want the system to be strongly consistent, you may opt for a consistency level `ONE` for the writes and `ALL` for the reads.

## Log Structured Merge tree

Cassandra (also, HBase) is heavily influenced by Log Structured Merge (LSM). It uses an LSM tree-like mechanism to store data on a disk. The writes are sequential (in append fashion) and the data storage is contiguous. This makes writes in Cassandra superfast, because there is no seek involved. Contrast this with an RDBMS system that is based on the B+ Tree ([http://en.wikipedia.org/wiki/B%2B\\_tree](http://en.wikipedia.org/wiki/B%2B_tree)) implementation.

LSM tree advocates the following mechanism to store data: note down the arriving modification into a log file (CommitLog), push the modification/new data into memory (MemTable) for faster lookup, and when the system has gathered enough updates in memory or after a certain threshold time, flush this data to a disk in a structured store file (SSTable). The logs corresponding to the updates that are flushed can now be discarded.

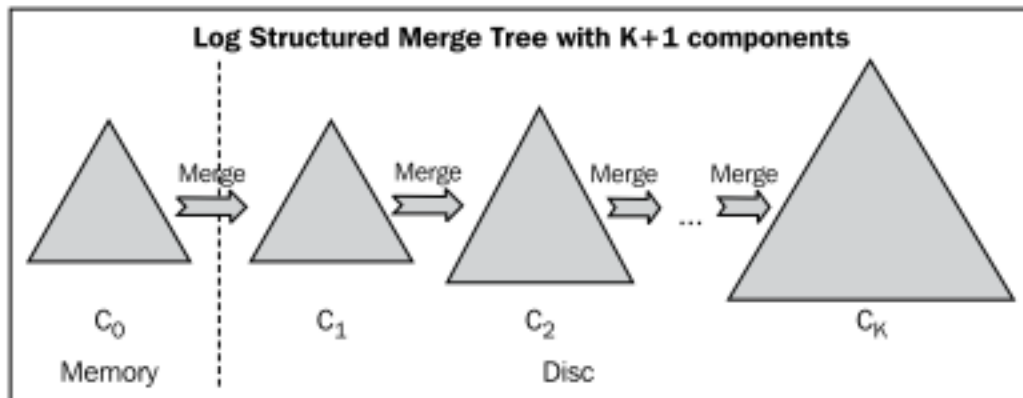



Figure 2.12: Log Structured Merge (LSM) Trees


 The Log-Structured Merge-Tree (LSM-Tree)(1996) by Patrick O'Neil et al is available at <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.2782>.



The preceding paper suggests multicomponent LSM trees, where data from memory is flushed into a smaller tree on disk for a quicker merge. When this tree fills up, it rolls them into a bigger tree. So, if you have K trees with the first tree being the smallest and the Kth being the largest, the memory gets flushed into the first tree, which when full, performs a rolling merge to the second tree, and so on. The change eventually lands up onto the Kth tree. This is a background process (similar to the compaction process in Cassandra). Cassandra differs a little bit where memory-resident data is flushed into immutable SSTables, which are eventually merged into one big SSTable by a background process. Like any other disk-resident access tree, popular pages are buffered into memory for faster access. Cassandra has a similar concept with key cache and row cache (optional) mechanisms.

We'll see the LSM tree in action in the context of Cassandra in the next three sections.

## CommitLog

One of the promises that Cassandra makes to the end users is durability. In conventional terms (or in ACID terminology), durability guarantees that a successful transaction (write, update) will survive permanently. This means once Cassandra says write successful that means the data is persisted and will survive system failures. It is done the same way as in any DBMS that guarantees durability: by writing the replayable information to a file before responding to a successful write. This log is called the CommitLog in the Cassandra realm.

This is what happens. Any write to a node gets tracked by `org.apache.cassandra.db.commitlog.CommitLog`, which writes the data with certain metadata into the CommitLog file in such a manner that replaying this will recreate the data. The purpose of this exercise is to ensure there is no data loss. If due to some reason the data could not make it into MemTable or SSTable, the system can replay the CommitLog to recreate the data.

CommitLog, MemTable, and SSTable in a node are tightly coupled. Any write operation gets written to the CommitLog first and then MemTable gets updated. MemTable, based on certain criteria, gets flushed to a disk in immutable files called SSTable. The data in CommitLogs gets purged after its corresponding data in MemTable gets flushed to SSTable. See *Figure 2.13*.

Also, there exists one single CommitLog per node server. Like any other logging mechanism, CommitLog is set to rolling after a certain size. (Why a single CommitLog? Why not one CommitLog per column family?)

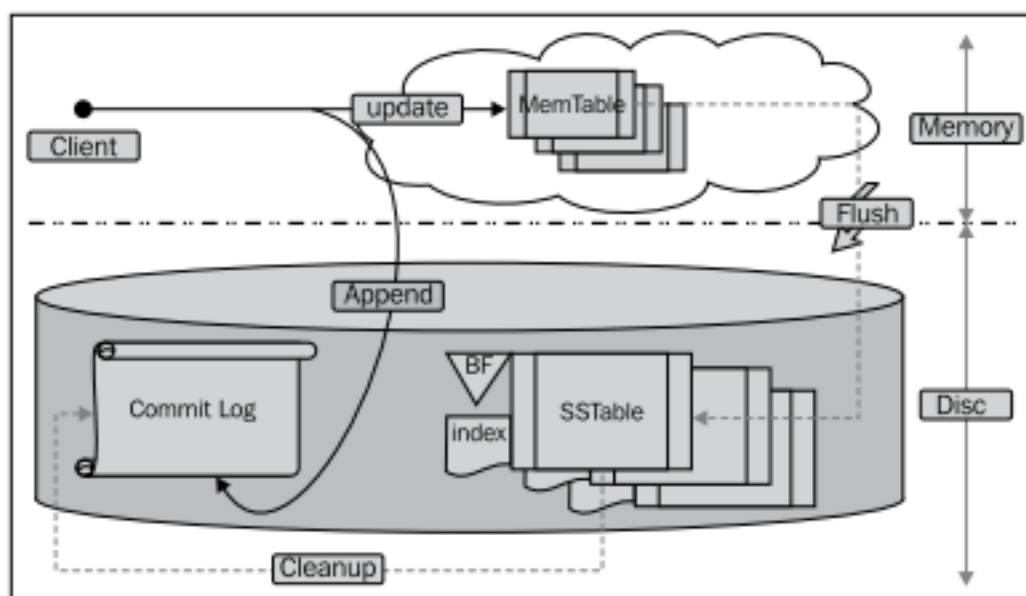
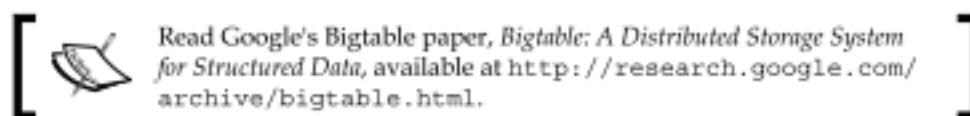


Figure 2.13: CommitLog, MemTable, and SSTable in action

Let's quickly go a bit deeper into implementation. All the classes that deal with the CommitLog management reside under `org.apache.cassandra.db.commitlog` package. The CommitLog singleton is a facade for all the operations. The implementations of `ICommitLogExecutorService` are responsible for write commands to the CommitLog file. Then there is a `CommitLogSegment` class. It manages a single CommitLog file, writes serialized write (mutation) to CommitLog, and it holds a very interesting property, `cfLastWrite`. `cfLastWrite` is a map with key as the column family name and value as an integer that represents the position (offset) in the CommitLog file where the last mutation for that column family is written. It can be thought of as a cursor one cursor per column family. When MemTable of a column family is flushed, the segments containing those mutations are marked as *clean* (for that particular column family). And when a new write arrives, it is marked *dirty* with offset at the latest mutation.

In the event of failure (hardware crash, abrupt shutdown), this is how CommitLog helps the system to recover:

1. Each CommitLog segment is iterated in ascending timestamp.
2. Lowest `ReplayPosition` (offset till which the data is persisted in SSTable) is chosen from SSTable metadata.
3. The log entry is replayed for a column family if the position of the log entry is greater than the replay position in the latest SSTable metadata.
4. After the log replay is done, all the MemTables are force flushed to a disk, and all the CommitLog segments are recycled.

## MemTable

MemTable is an in-memory representation of column family. It can be thought of as a cached data. MemTable is sorted by key. Data in MemTable is sorted by row key. Unlike CommitLog, which is append-only, MemTable does not contain duplicates. A new write with a key that already exists in the MemTable overwrites the older record. This being in memory is both fast and efficient. The following is an example:

Write 1: {k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

In CommitLog (new entry, append):  
{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

In MemTable (new entry, append):  
{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

Write 2: {k2: [{c4, v4}]}

In CommitLog (new entry, append):  
{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}  
{k2: [{c4, v4}]}

In MemTable (new entry, append):  
{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}  
{k2: [{c4, v4}]}

Write 3: {k1: [{c1, v5}, {c6, v6}]}

In CommitLog (old entry, append):  
{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}  
{k2: [{c4, v4}]}

```
{k1: [{c1, v5}, {c6, v6}]}
```

```
In MemTable (old entry, update):
```

```
{k1: [{c1, v5}, {c2, v2}, {c3, v3}, {c6, v6}]}
```

```
{k2: [{c4, v4}]}
```

Cassandra Version 1.1.1 uses SnapTree (<https://github.com/nbrnson/snaptree>) for MemTable representation, which claims it to be "... a drop-in replacement for ConcurrentSkipListMap, with the additional guarantee that clone() is atomic and iteration has snapshot isolation." See also copy-on-write and compare-and-swap (<http://en.wikipedia.org/wiki/Copy-on-write>, <http://en.wikipedia.org/wiki/Compare-and-swap>).

Any write gets written first to CommitLog and then to MemTable.

## SSTable

SSTable is a disk representation of the data. MemTables gets flushed to disk to immutable SSTables. All the writes are sequential, which makes this process fast. So, the faster the disk speed, the quicker the flush operation.

The SSTables eventually get merged in the compaction process and the data gets organized properly into one file. This extra work in compaction pays off during reads.

SSTables have three components: Bloom filter, index files, and datafiles.

## Bloom filter

Bloom filter is a litmus test for the availability of certain data in storage (collection). But unlike a litmus test, a Bloom filter may result in false positives: that is, it says that a data exists in the collection associated with the Bloom filter, when it actually does not. A Bloom filter never results in a false negative. That is, it never states that a data is not there while it is. The reason to use Bloom filter, even with its false-positive defect, is because it is superfast and its implementation is really simple.

Cassandra uses Bloom filters to determine whether an SSTable has the data for a particular row key. Bloom filters are unused for range scans, but they are good candidates for index scans. This saves a lot of disk I/O that might take in a full SSTable scan, which is a slow process. That's why it is used in Cassandra, to avoid reading many, many SSTables, which can become a bottleneck.

**How Bloom filter works:** Bloom filter in its simplest form can be assumed as a bit array of length  $l$ , with all elements set to zero. It also has  $k$  predefined hash functions associated with it. See Figure 2.14 as shown:



Figure 2.14: Bloom filter in action. It uses three hash functions and sets the corresponding bit in the array to 1 (it might already be 1)

To add a key to a Bloom filter (at the time of entering data in the associated collection),  $k$  hashes are calculated using  $k$  predefined hash functions. A modulus of each hash value is taken using array length  $l$ , and the value at this array position is set to 1.

The following pseudo code shows what happens when a value  $v$  is inserted in Bloom filter:

```
//calculate hash, mod it to get location in bit array
arrayIndex1 = md5(v) % arrayLength
arrayIndex2 = sha1(v) % arrayLength
arrayIndex3 = murmur(v) % arrayLength

//set all those indexes to 1
bitArray[arrayIndex1] = 1
bitArray[arrayIndex2] = 1
bitArray[arrayIndex3] = 1
```

To query the existence of a key in the Bloom filter, the process is similar. Take the key and calculate the predefined hash values. Take mod with bit array length. Look into those locations. If it turns out that at least one of those array locations have zero value in it, it is sure that this value was never inserted in this Bloom filter and hence does not exist in the associated collection. On the other hand, if all values are 1s, this means the value may exist in the collection associated with this Bloom filter. We cannot guarantee its presence in the collection because it is possible that there exists other  $k$  keys whose  $i$ th hash function filled the same spot in the array as the  $j$ th hash of the key that we are looking for.



Removal of a key from Bloom filter as in its original avatar is not possible. One may break multiple keys because multiple keys may have the same index bit set to 1 in the array for different hashes. Counting Bloom filter solves these issues by changing the bit array into an integer array where each element works as a counter; insertion increments the counter and deletion decrements.

Effectiveness of Bloom filter depends on the size of the collection it is applied to. The bigger the collection associated with the Bloom filter, the higher the frequency of false positives (because the array will be more densely packed with 1s). Another thing that governs Bloom filter is the quality of a good hash function. A good hash function will distribute hash values evenly in the array, and it will be fast. One does not look at the cryptic strength of the hash function here, so Murmur3 hash will be preferred over SHA1 hash.

## Index files

Index files are companion files of SSTables. The same as Bloom filter, there exists one index file per SSTable. It contains all the row keys in the SSTable and its offset at which the row starts in the datafile.

At startup, Cassandra reads every 128th key (configurable) into the memory (sampled index). When the index is looked for a row key (after Bloom filter hinted that the row key might be in this SSTable), Cassandra performs a binary search on the sampled index in memory. Followed by a positive result from the binary search, Cassandra will have to read a block in the index file from the disk starting from the nearest value lower than the value that we are looking for.

Let's take an example; see *Figure 2.16*. Cassandra is looking for a row key 404. It is not in MemTable. On querying the Bloom filter of a certain SSTable, Cassandra gets a positive nod that this SSTable may contain the row. Next is to look into the SSTable. But before we start scanning the SSTable or the index file, we can get some help from the sampled index in memory. Looking through the sampled index, Cassandra finds out that there exists a row key 400 and another 624. So, the row fragments may be in this SSTable. But more importantly, the sampled index tells the offset about the 400 entry in the index file. Cassandra now scans the SSTable from 400 and gets to the entry for 404. This tells Cassandra the offset of the entry for the 404 key in SSTable and it reads from there.

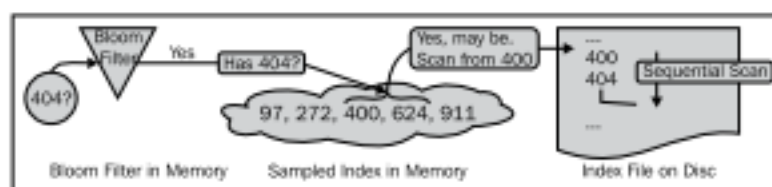


Figure 2.15: Cassandra SSTable index in action

If you followed the example, you must have observed that the smaller the sampling size, the more the number of keys in the memory; the smaller the size of the block to read on disk, the faster the results. This is a trade-off between memory usage and performance.

## Datafiles

Datafiles are the actual data. They contain row keys, metadata, and columns (partial or full). Reading data from datafiles is just one disk seek followed by a sequential read, as offset to a row key is already obtained from the associated index file.

## Compaction

As we discussed earlier in the section (See *Figure 2.8*), a read require may require Cassandra to read across multiple SSTables to get a result. This is wasteful, costs multiple (disk) seeks, may require a conflict resolution, and if there are too many, SSTables were created. To handle this problem, Cassandra has a process in place, namely, compaction. Compaction merges multiple SSTable files into one. Off the shelf, Cassandra offers two types of compaction mechanism: **Size Tiered Compaction Strategy** and **Level Compaction Strategy** (*Chapter 5, Performance Tuning*, under the Choosing the right compaction strategy section). This section stays focused on the Size Tiered Compaction mechanism for better understanding.

The compaction process starts when the number of SSTables on disk reaches a certain threshold (N: configurable). Although the merge process is a little I/O intensive, it benefits in the long term with a lower number of disk seeks during reads. Apart from this, there are a few other benefits of compaction as follows:

- Removal of expired tombstones (Cassandra v0.8+)
- Merging row fragments
- Rebuilds primary and secondary indexes

Merge is not as painful as it may seem, because SSTables are already sorted. (Remember merge-sort?) Merge results into larger files, but old files are not deleted immediately. For example, let's say you have compaction threshold set to four. Cassandra initially creates SSTables of the same size as MemTable. When the number of SSTables surpasses the threshold, the compaction thread triggers. This compacts the four equal-sized SSTables into one. Temporarily, you will have two times the total SSTable data on your disk. Another thing to note is that SSTables that get merged have the same size. So, when the four SSTables get merged to give a larger SSTable of size, say G, the buckets for the rest to-be-filled SSTables will be G each. So, the next compaction is going to take an even larger space while merging.

The SSTables, after merging, are marked as *deleteable*. They get deleted at a garbage collection cycle of the JVM or when Cassandra restarts.

The compaction process happens on each node and does not affect others; this is called **minor compaction**. This is automatically triggered, system controlled, and regular. There is more than one type of compaction setting that exists in Cassandra. We'll see them again in detail in *Chapter 5, Performance Tuning*, under the Choosing the right compaction strategy section. Another league of compaction is called, obviously, **major compaction**.

What's a major compaction? Major compaction takes all the SSTables, and merges it to one single SSTable. It is somewhat confusing when you see that a minor compaction merges SSTables and a major one does it too. There is a slight difference; for example, if we take Size Tiered Compaction Strategy, it merges the tables of the same size. So, if your threshold is four, Cassandra will start to merge when it finds four same-sized SSTables. If your system starts with four SSTables of size X, after the compaction you will end up with one SSTable of size 4X. Next time when you have four X-sized SSTables you will end up with two 4X tables, and so on. (These larger SSTables will get merged after 16 X-sized SSTables gets merged into four 4X tables.) After a really long time you will end up with a couple of really big SSTables, a handful of large SSTables, and many smaller SSTables. This is a result of continuous minor compaction. So, you may need to hop a couple of SSTables to get data for a query. Then, you run a major compaction and all the big and small SSTables get merged into one. This is the only benefit of major compaction.



Major compaction may not be the best idea after Cassandra v0.8+. There are a couple of reasons for this. One reason is automated minor compaction no longer runs after a major compaction is executed. So, this adds up manual intervention or doing extra work (such as setting a cron job) to perform regular major compaction. The performance gain after major compaction may deteriorate with time. Major compaction creates large SSTable. The larger the SSTable, the higher the false positive rate from the Bloom filter. Large SSTable will have large index, it will take longer to perform binary search for them.

## Tombstones

Cassandra is a complex system with its data distributed among CommitLogs, MemTables, and SSTables on a node. The same data is then replicated over replica nodes. So, like everything else in Cassandra, deletion is going to be eventful. Deletion, to an extent, follows an update pattern except Cassandra tags the deleted data with a special value, and marks it as a **tombstone**. This marker helps future queries, compaction, and conflict resolution. Let's step further down and see what happens when a column from a column family is deleted.

A client connected to a node (coordinator node, but it may not be the one holding the data that we are going to mutate), issues a delete command for a column *C*, in a column family *CF*. If the consistency level is satisfied, the delete command gets processed. When a node, containing the row key, receives a delete request, it updates or inserts the column in *MemTable* with a special value, namely, tombstone. The tombstone basically has the same column name as the previous one; the value is set to UNIX epoch. The timestamp is set to what the client has passed. When a *MemTable* is flushed to *SSTable*, all tombstones go into it as any regular column.

On the read side, when the data is read locally on the node and it happens to have multiple versions of it in different *SSTables*, they are compared and the latest value is taken as the result of reconciliation. If a tombstone turns out to be a result of reconciliation, it is made a part of the result that this node returns. So, at this level, if a query has a deleted column, this exists in the result. But the tombstones will eventually be filtered out of the result before returning it back to the client. So, a client can never see a value that is a tombstone.

For consistency levels more than one, the query is executed on as many replicas as the consistency level. The same as a regular read process, data from the closest node and a digest from the remaining nodes is obtained (to satisfy the consistency level). If there is a mismatch such as the tombstone is not yet propagated to all the replicas, a partial read repair is triggered, where the final view of the data is sent to all the nodes that were involved in this read to satisfy the consistency level.

One thing where delete differs from update is a compaction. A compaction removes a tombstone only if its (the tombstone's) garbage collection's grace seconds ( $\tau$ ) are over. This  $\tau$  is called *GCGraceSeconds* (configurable). So, do not expect that a major deletion will free up a lot of space immediately.

What happens to a node that was holding a data that was deleted (in other live replicas) when this node was down? If a tombstone still exists in any of the replica nodes, the delete information will eventually be available to the previously dead node. But a compaction occurs at *GCGraceSeconds*, after the deletion will kick the old tombstones out. This is a problem, because no information about the deleted column is left. Now, if a node, that was dead all the time during *GCGraceSeconds*, wakes up and sees that it has some data that no other node has, it will treat this data as a fresh data and assuming a write failure, it will replicate the data over all the other replica nodes. The old data will resurrect and replicate, and may reappear in client results.

Although *GCGraceSeconds* is 10 days by default, before which any sane system admin will bring the node back in, or discard the node completely. But it is something to watch out for and repair nodes occasionally.

## Hinted handoff

When we last talked about durability, we observed Cassandra provides CommitLogs to provide write durability. This is good. But what if the node, where the writes are going to be, is itself dead? No communication will keep anything new to be written to the node. Cassandra, inspired by Dynamo, has a feature called hinted handoff. In short, it's the same as taking a quick note locally that X cannot be contacted. Here is the mutation (operation that requires modification of the data such as insert, delete, and update) M that will be required to be replayed when it comes back.

The coordinator node (the node which the client is connected to) on receipt of a mutation/write request, forwards it to appropriate replicas that are alive. If this fulfills the expected consistency level, write is assumed successful. The write requests to a node that does not respond to a write request or is known to be dead (via gossip) and is stored locally in the `system.hints` table. This hint contains the mutation. When a node comes to know via gossip that a node is recovered, it replays all the hints it has in store for that node. Also, every 10 minutes, it keeps checking any pending hinted handoffs to be written.

Why worry about hinted handoff when you have written to satisfy consistency level? Wouldn't it eventually get repaired? Yes, that's right. Also, hinted handoff may not be the most reliable way to repair a missed write. What if the node that has hinted handoff dies? This is a reason why we do not count on hinted handoff as a mechanism to provide consistency (except for the case of the consistency level, `ANY`) guarantee; it's a single point of failure. The purpose of hinted handoff is, one, to make restored nodes quickly consistent with the other live ones; and two, to provide extreme write availability when consistency is not required.

The way extreme write availability is obtained is at the cost of consistency. One can set consistency level for writes to `ANY`. What happens next is, if all the replicas that are meant to hold this value are down, Cassandra will just write a local hinted handoff and return write success to the client. There is one caveat; the handoff can be on any node. So, a read for the data that we have written as a hint will not be available as long as the replicas are dead plus until the hinted handoff is replayed. But it is a nice feature.



There is a slight difference where hinted handoff is stored in Cassandra's different versions. Prior to Cassandra 1.0, hinted handoff is stored on one of the replica nodes that can be communicated with. From Version 1.0+ (including 1.0), handoff can be written on the coordinator node (the node which the client is connected to).

Removing a node from a cluster causes deletion of hinted handoff stored for that node. All hints for deleted records are dropped.



## Read repair and Anti-entropy

Cassandra promises eventual consistency and read repair is the process which does that part. Read repair, as the name suggests, is the process of fixing inconsistencies among the replicas at the time of read. What does that mean? Let's say we have three replica nodes A, B, and C that contain a data  $x$ . During an update,  $x$  is updated to  $x_1$  in replicas A and B. But it is failed in replica C for some reason. On a read request for data  $x$ , the coordinator node asks for a full read from the nearest node (based on the configured Snitch) and *digest* of data  $x$  from other nodes to satisfy consistency level. The coordinator node compares these values (something like `digest(full_x) == digest_from_node_c`). If it turns out that the digests are the same as the digests of full read, the system is consistent and the value is returned to the client. On the other hand, if there is a mismatch, full data is retrieved and reconciliation is done and the client is sent the reconciled value. After this, in background, all the replicas are updated with the reconciled value to have a consistent view of data on each node. See Figure 2.1 as shown:

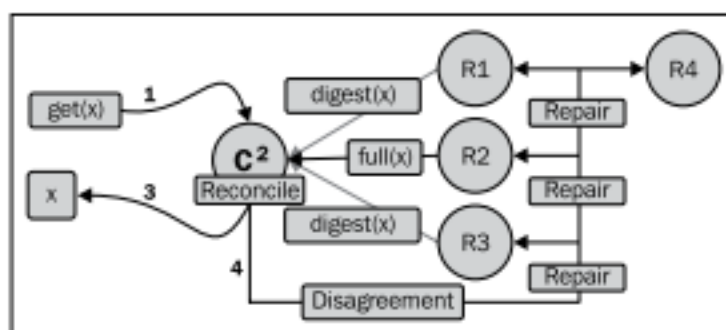


Figure 2.16: Image showing read repair dynamics. 1. Client queries for data  $x$ , from a node C (coordinator). 2. C gets data from replicas R1, R2, and R3; reconciles. 3. Sends reconciled data to client. 4. If there is a mismatch across replicas, repair is invoked.

So, we have got a consistent view on read. What about the data that is inserted, but never read? Hinted handoff is there, but we do not rely on hinted handoff for consistency. What if the node containing hinted handoff data dies, and the data that contains the hint is never read? Is there a way to fix them without read? This brings us to the Anti-entropy architecture of Cassandra (borrowed from Dynamo).

Anti-entropy compares all the replicas of a column family and updates the replica to the latest. This happens during major compaction. It uses Merkle tree to determine discrepancy among the replicas and fixes it.

## Merkle tree

Merkle tree (*A digital signature Based On A Conventional Encryption Function* by Merkle, R. (1988), available at <http://www.cse.msstate.edu/~ramkumar/merkle2.pdf>.) is a hash tree where leaves of the tree hashes hold actual data in a column family and non-leaf nodes hold hashes of their children. The unique advantage of Merkle tree is a whole subtree can be validated just by looking at the value of the parent node. So, if nodes on two replica servers have the same hash values, then the underlying data is consistent and there is no need to synchronize. If one node passes the whole Merkle tree of a column family to another node, it can determine all the inconsistencies.

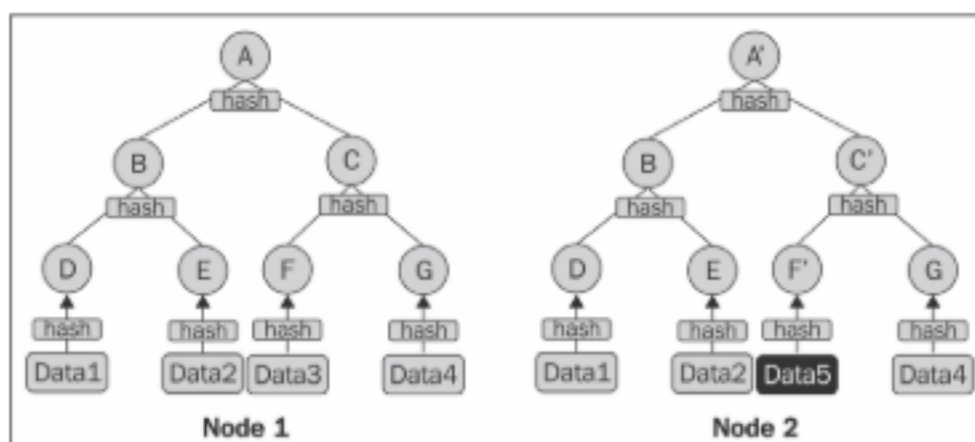


Figure 2.17: Merkle tree to determine mismatch in hash values at parent nodes due to the difference in underlying data

To exemplify, Figure 2.17 shows the Merkle tree from two nodes with inconsistent data. A process comparing these two trees would know that there is something inconsistent, because the hash value stored in the top node does not match. It can descend down and know that the right subtree is likely to have an inconsistency. And then the same process is repeated until it finds out all the data that mismatches.

## Summary

By now, you are familiar with all the nuts and bolts of Cassandra. We have discussed how the pressure to make data stores to web scale inspired a rather not-so-common database mechanism to come to the mainstream; how CAP theorem governs the behavior of such databases. We have seen that Cassandra shines out among its peers. Then, we dipped our toes into the big picture of Cassandra read and write mechanisms. This left us with lots of fancy terms. Further, we looked into the definition of these words, components that drive Cassandra and their influence on its behavior.

It is understandable that it may be a lot to take in for someone new to NoSQL systems. It is okay if you do not have complete clarity at this point. As you start working with Cassandra, tweaking it, experimenting with it, and going through the Cassandra mailing list discussions or talks, you will start to come across stuff that you have read in this chapter and it will start to make sense, and perhaps you may want to come back and refer to this chapter to improve clarity.

It is not required to understand this chapter fully to be able to write queries, set up clusters, maintain clusters, or do anything else related to Cassandra. A general sense of this chapter will take you far enough to work extremely well with Cassandra-based projects.

How does this knowledge help us in building an application? Isn't it just about learning Thrift or CQL API and get going? You might be wondering why you need to know about compaction and storage mechanism when all you need to do is to deliver an application that has a superfast backend. It is not obvious at this point why we are learning this, but as we move ahead with developing an application, we will come to realize that knowledge about underlying storage mechanism helps. In later chapters, when we will learn about deploying a cluster, performance tuning, maintenance, and integrating with other tools such as Apache Hadoop, you may find this chapter useful. At this point, we are ready to learn some of the common use cases, and how they utilize various features of Cassandra. The next chapter is about knowing how to use Cassandra.