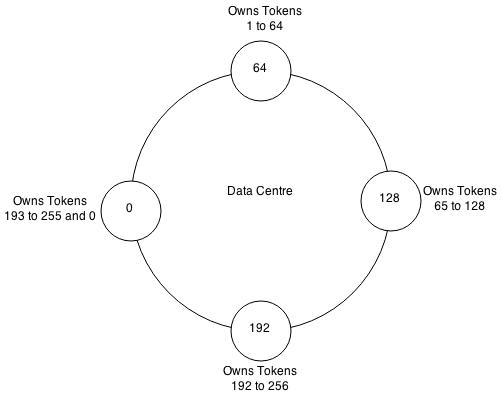
**Cassandra Cluster/Ring**

* **Cluster Bootstrapping** - assigned a name. All nodes participating in a cluster have the same name. Seed nodes are used during start up to help discover all participating nodes. Seeds nodes have no special purpose other than helping bootstrap the cluster using the gossip protocol. When a node starts up it looks to its seed list to obtain information about the other nodes in the cluster. Cassandra uses the gossip protocol for intra cluster communication and failure detection. A node exchanges state information with a maximum of three other nodes. State information is exchanged every second and contains information about itself and all other known nodes.  This enables each node to learn about every other node in the cluster even though it is communicating with a small subset of nodes.

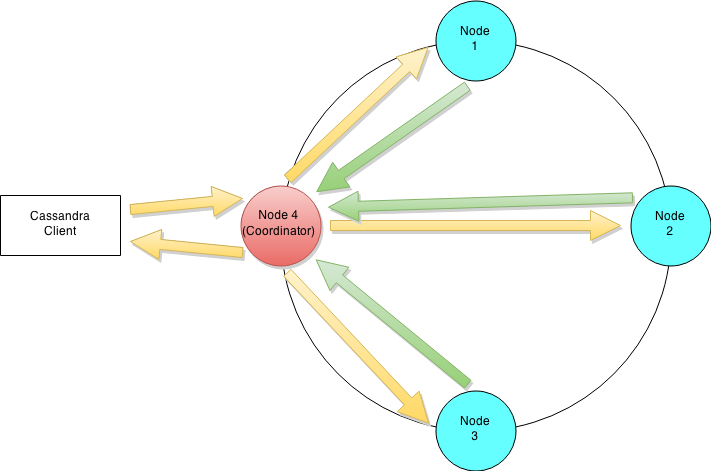
**Cassandra Ring**

[](http://abiasforaction.net/wp-content/uploads/2015/01/Cassandra-Ring.jpg)

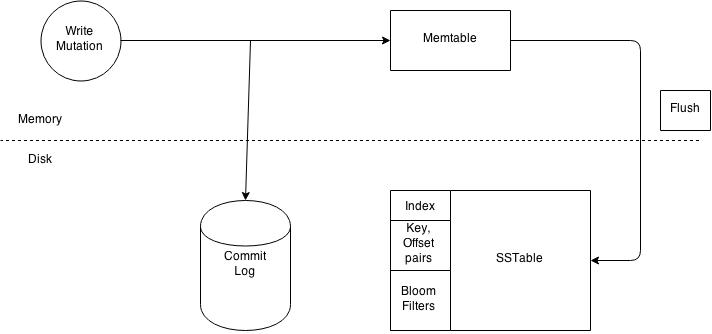
***Example Cassandra ring distributing 255 tokens evenly across four nodes.***

A Cassandra cluster is visualised as a ring because it uses a consistent hashing algorithm to distribute data. At start up each node is assigned a token range which determines its position in the cluster and the rage of data stored by the node. Each node receives a proportionate range of the token ranges to ensure that data is spread evenly across the ring. The figure above illustrates dividing a 0 to 255 token range evenly amongst a four node cluster. Each node is assigned a token and is responsible for token values from the previous token (exclusive) to the node's token (inclusive). Each node in a Cassandra cluster is responsible for a certain set of data which is determined by the partitioner. A partitioner is a hash function for computing the resultant token for a particular row key. This token is then used to determine the node which will store the first replica.  Currently Cassandra offers a Murmur3Partitioner (default), RandomPartitioner and a ByteOrderedPartitioner.

**Cassandra Write Path**

Lets try and understand Cassandra's architecture by walking through an example write mutation. Let's assume that a client wishes to write a piece of data to the database. The diagram below illustrates the cluster level interaction that takes place.[](http://abiasforaction.net/wp-content/uploads/2015/01/Cassandra-Write-Path-Ring.png)

***Cluster level interaction for a write and read operation.***

Since Cassandra is masterless a client can connect with any node in a cluster. Clients can interface with a Cassandra node using either a thrift protocol or using CQL. In the picture above the client has connected to Node 4. The node that a client connects to is designated as the coordinator, also illustrated in the diagram. The coordinators is responsible for satisfying the clients request. The consistency level determines the number of nodes that the coordinator needs to hear from in order to notify the client of a successful mutation.  All inter-node requests are sent through a messaging service and in an asynchronous manner. Based on the partition key and the replication strategy used the coordinator forwards the mutation to all applicable nodes. In our example it is assumed that nodes 1,2 and 3 are the applicable nodes where node 1 is the first replica and nodes two and three are subsequent replicas. The coordinator will wait for a response from the appropriate number of nodes required to satisfy the consistency level.  QUORUM is a commonly used consistency level which refers to a majority of the nodes.QUORUM can be calculated using the formula (n/2 +1) where n is the replication factor. In our example let's assume that we have a consistency level of QUORUM and a replication factor of three. Thus the coordinator will wait for at most 10 seconds (default setting) to hear from at least two nodes before informing the client of a successful mutation.[](http://abiasforaction.net/wp-content/uploads/2015/01/Cassandra-Write-Path.jpg)

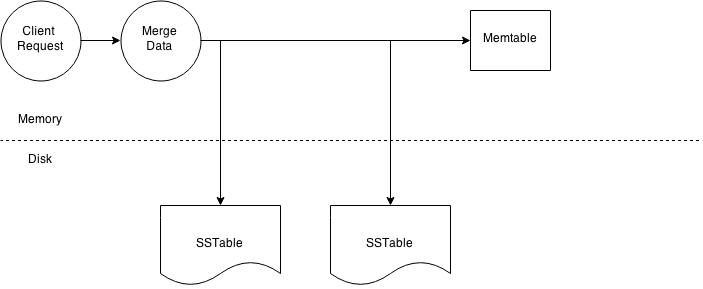
***Write operations at a node level.***

Each node processes the request individually. Every node first writes the mutation to the commit log and then writes the mutation to the memtable. Writing to the commit log ensures durability of the write as the memtable is an in-memory structure and is only written to disk when the memtable is flushed to disk. A memtable is flushed to disk when:

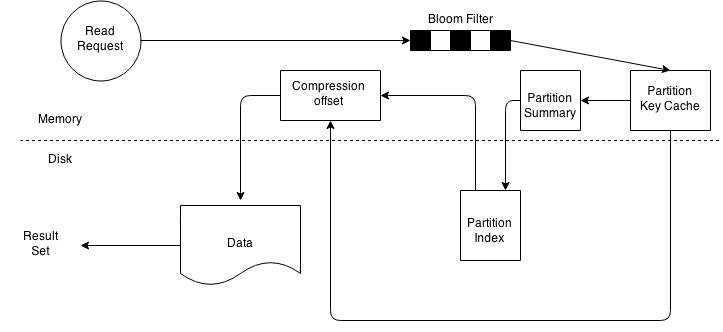
1. It reaches its maximum allocated size in memory
2. The number of minutes a memtable can stay in memory elapses.
3. Manually flushed by a user

A memtable is flushed to an immutable structure called and SSTable (Sorted String Table). The commit log is used for playback purposes in case data from the memtable is lost due to node failure. For example the machine has a power outage before the memtable could get flushed. Every SSTable creates three files on disk which include a bloom filter, a key index and a data file. Over a period of time a number of SSTables are created. This results in the need to read multiple SSTables to satisfy a read request. Compaction is the process of combining SSTables so that related data can be found in a single SSTable. This helps with making reads much faster.

**Cassandra Read Path**

At the cluster level a read operation is similar to a write operation. As with the write path the client can connect with any node in the cluster. The chosen node is called the coordinator and is responsible for returning the requested data.  A row key must be supplied for every read operation. The coordinator uses the row key to determine the first replica. The replication strategy in conjunction with the replication factor is used to determine all other applicable replicas. As with the write path the consistency level determines the number of replica's that must respond before successfully returning data. Let's assume that the request has a consistency level of QUORUM and a replication factor of three, thus requiring the coordinator to wait for successful replies from at least two nodes. If the contacted replicas has a different version of the data the coordinator returns the latest version to the client and issues a read repair command to the node/nodes with the older version of the data. The read repair operation pushes the newer version of the data to nodes with the older version.[](http://abiasforaction.net/wp-content/uploads/2015/01/Cassandra-Read-Path-Overview.jpg)

***Node level read operation.***

The illustration above outlines key steps when reading data on a particular node. Every Column Family stores data in a number of SSTables. Thus Data for a particular row can be located in a number of SSTables and the memtable. Thus for every read request Cassandra needs to read data from all applicable SSTables ( all SSTables for a column family) and scan the memtable for applicable data fragments. This data is then merged and returned to the coordinator.[](http://abiasforaction.net/wp-content/uploads/2015/01/Cassandra-Read-Path.jpg)

***SSTable read path.***

On a per SSTable basis the operation becomes a bit more complicated. The illustration above outlines key steps that take place when reading data from an SSTable. Every SSTable has an associated bloom filter which enables it to quickly ascertain if data for the requested row key exists on the corresponding SSTable. This reduces IO when performing an row key lookup. A bloom filter is always held in memory since the whole purpose is to save disk IO. Cassandra also keeps a copy of the bloom filter on disk which enables it to recreate the bloom filter in memory quickly .  Cassandra does not store the bloom filter Java Heap instead makes a separate allocation for it in memory.  If the bloom filter returns a negative response no data is returned from the particular SSTable. This is  a common case as the compaction operation tries to group all row key related data into as few SSTables as possible. If the bloom filter provides a positive response the partition key cache is scanned to ascertain the compression offset for the requested row key. It then proceeds to fetch the compressed data on disk and returns the result set. If the partition cache does not contain a corresponding entry the partition key summary is scanned. The partition summary is a subset to the partition index and helps determine the approximate location of the index entry in the partition index. The partition index is then scanned to locate the compression offset which is then used to find the appropriate data on disk. If you reached the end of this long post then well done. In this post I have provided an introduction to Cassandra architecture. In my upcoming posts I will try and explain Cassandra architecture using a more practical approach.