Introduction

An Introduction To NoSQL & Apache Cassandra , introduced us to various types of NoSQL database and Apache Cassandra. In this article I am going to delve into Cassandra's Architecture. Cassandra is a peer-to-peer distributed database that runs on a cluster of homogeneous nodes. Cassandra has been architected from the ground up to handle large volumes of data while providing high availability. Cassandra provides high write and read throughput. A Cassandra cluster has no special nodes i.e. the cluster has no masters, no slaves or elected leaders. This enables Cassandra to be highly available while having no single point of failure.

Key Concepts, Data Structures and Algorithms

In order to understand Cassandra's architecture it is important to understand some key concepts, data structures and algorithms frequently used by Cassandra.

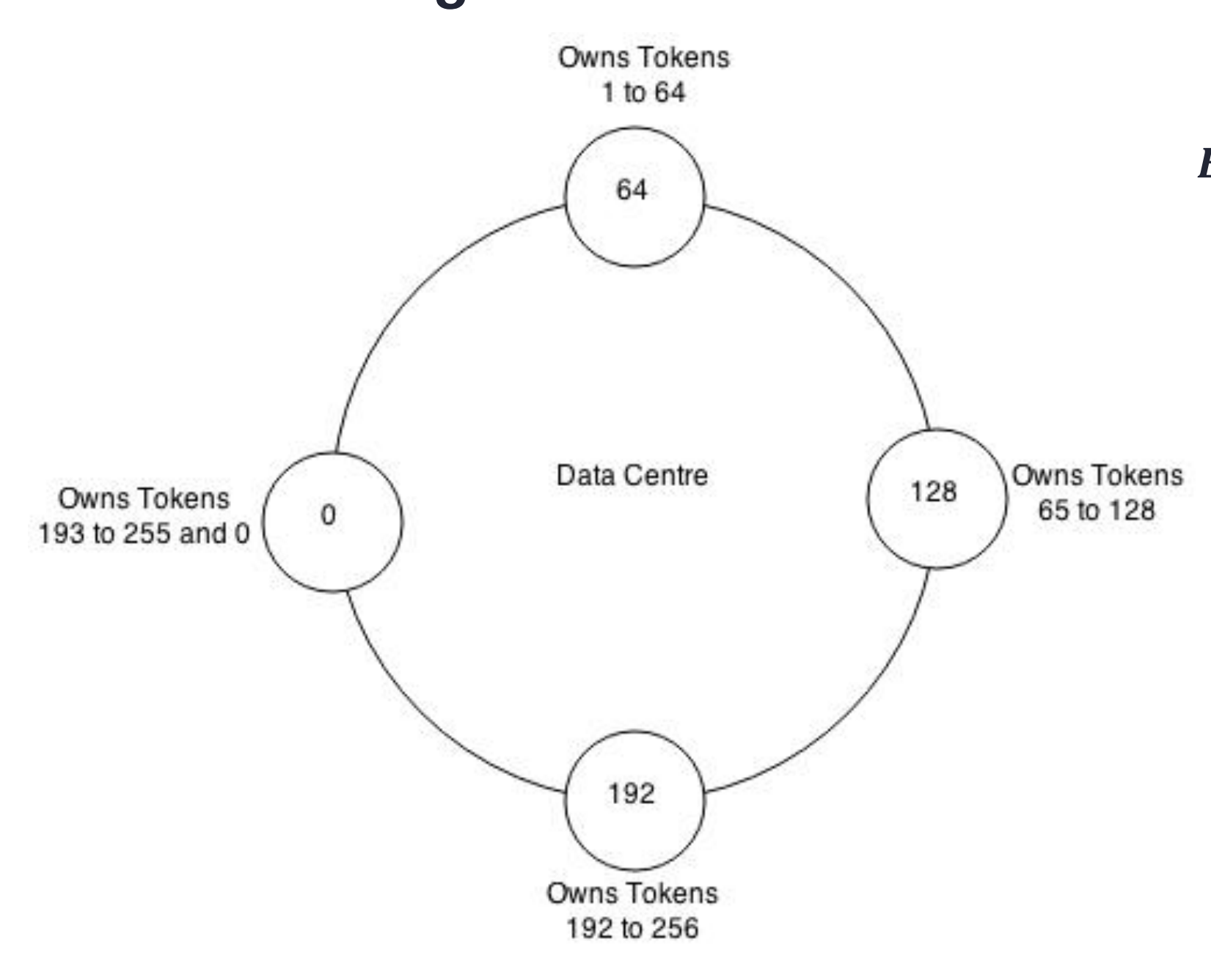
- **Data Partitioning** - Apache Cassandra is a distributed database system using a shared nothing architecture. A single logical database is spread across a cluster of nodes and thus the need to spread data evenly amongst all participating nodes. At a 10000 foot level Cassandra stores data by dividing data evenly around its cluster of nodes. Each node is responsible for part of the data. The act of distributing data across nodes is referred to as data partitioning.
- **Consistent Hashing** - Two main problems crop up when trying to distribute data efficiently. One, determining a node on which a specific piece of data should reside on. Two, minimising data movement when adding or removing nodes. Consistent hashing enables us to achieve these goals. A consistent hashing algorithm enables us to map Cassandra row keys to physical nodes. The range of values from a consistent hashing algorithm is a fixed circular space which can be visualised as a ring. Consistent hashing also minimises the key movements when nodes join or leave the cluster. On average only k/n keys need to be remapped where k is the number of keys and n is the number of slots (nodes). This is in stark contrast to most hashing algorithms where a change in the number of slots results in the need to remap a large number of keys.
- **Data Replication** - Partitioning of data on a shared nothing system results in a single point of failure i.e. if one of the nodes goes down part of your data is unavailable. This limitation is overcome by creating copies of the data, know as replicas, thus avoiding a single point of failure. Storing copies of data on multiple nodes is referred to as replication. Replication of data ensures fault tolerance and reliability.
- **Eventual Consistency** - Since data is replicated across nodes we need to ensure that data is synchronized across replicas. This is referred to as data consistency. Eventual consistency is a consistency model used in distributed computing. It theoretically guarantees that, provided there are no new updates, all nodes/replicas will eventually return the last updated value. Domain Name System (DNS) are a good example of an eventually consistent system.
- **Tunable Consistency -** Cassandra provides tunable consistency i.e. users can determine the consistency level by tuning it via read and write operations. Eventual consistency often conjures up fear and doubt in the minds of application developers. The key thing to keep in mind is that reaching a consistent state often takes microseconds.
- **Consistency Level** - Cassandra enables users to configure the number of replicas in a cluster that must acknowledge a read or write operation before considering the operation successful. The consistency level is a required parameter in any read and write operation and determines the exact number of nodes that must successfully complete the operation before considering the operation successful.
- **Data Centre, Racks, Nodes** - A Data Centre (DC) is a centralised place to house computer and networking systems to help meet an organisation's information technology needs. A rack is a unit that contains multiple servers all stacked one on top of another. A rack enables data centres to conserve floor space and consolidates networked resources. A node is a single server in a rack. Why do we care? Often Cassandra is deployed in a DC environment and one must replicate data intelligently to ensure no single point of failure. Data must be replicated to servers in different racks to ensure continued availability in the case of rack failure. Cassandra can be easily configured to work in a multi DC environment to facilitate fail over and disaster recovery.
- **Snitches and Replication Strategies** - As mentioned above it is important to intelligently distribute data across DC's and racks. In Cassandra the distribution of data across nodes is configurable. Cassandra uses snitches and replication strategies to determine how data is replicated across DC's, racks and nodes. Snitches determine proximity of nodes within a ring. Replication strategies use proximity information provided by snitches to determine locality of a particular copy.
- **Gossip Protocol** - Cassandra uses a gossip protocol to discover node state for all nodes in a cluster. Nodes discover information about other nodes by exchanging state information about themselves and other nodes they know about. This is done with a maximum of 3 other nodes. Nodes do not exchange information with every other node in the cluster in order to reduce network load. They just exchange information with a few nodes and over a period of time state information about every node propagates throughout the cluster. The gossip protocol facilitates failure detection.
- **Bloom Filters** - A bloom filter is an extremely fast way to test the existence of a data structure in a set. A bloom filter can tell if an item might exist in a set or definitely does not exist in the set. False positives are possible but false negatives are not. Bloom filters are a good way of avoiding expensive I/O operation.
- **Merkle Tree** - Merkle tree is a hash tree which provides an efficient way to find differences in data blocks. Leaves contain hashes of individual data blocks and parent nodes contain hashes of their respective children. This enables efficient way of finding differences between nodes.
- **SSTable** - A Sorted String Table (SSTable) ordered immutable key value map. It is basically an efficient way of storing large sorted data segments in a file.
- **Write Back Cache** - A write back cache is where the write operation is only directed to the cache and completion is immediately confirmed. This is different from Write-through cache where the write operation is directed at the cache but is only confirmed once the data is written to both the cache and the underlying storage structure.
- **Memtable** - A memtable is a write back cache residing in memory which has not been flushed to disk yet.
- **Cassandra Keyspace** - Keyspace is similar to a schema in the RDBMS world. A keyspace is a container for all your application data. When defining a keyspace, you need to specify a replication strategy and a replication factor i.e. the number of nodes that the data must be replicate too.
- **Column Family** - A column family is analogous to the concept of a table in an RDBMS. But that is where the similarity ends. Instead of thinking of a column family as RDBMS table think of a column family as a map of sorted map. A row in the map provides access to a set of columns which is represented by a sorted map. Map<RowKey, SortedMap<ColumnKey, ColumnValue>> Please note in CQL (Cassandra Query Language) lingo a Column Family is referred to as a table.
- **Row Key** - A row key is also known as the partition key and has a number of columns associated with it i.e. a sorted map as shown above. The row key is responsible for determining data distribution across a cluster.

Cassandra Cluster/Ring

Cluster Bootstrapping

Every Cassandra cluster must be 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



Example Cassandra ring distributing 255 tokens evenly across four nodes.

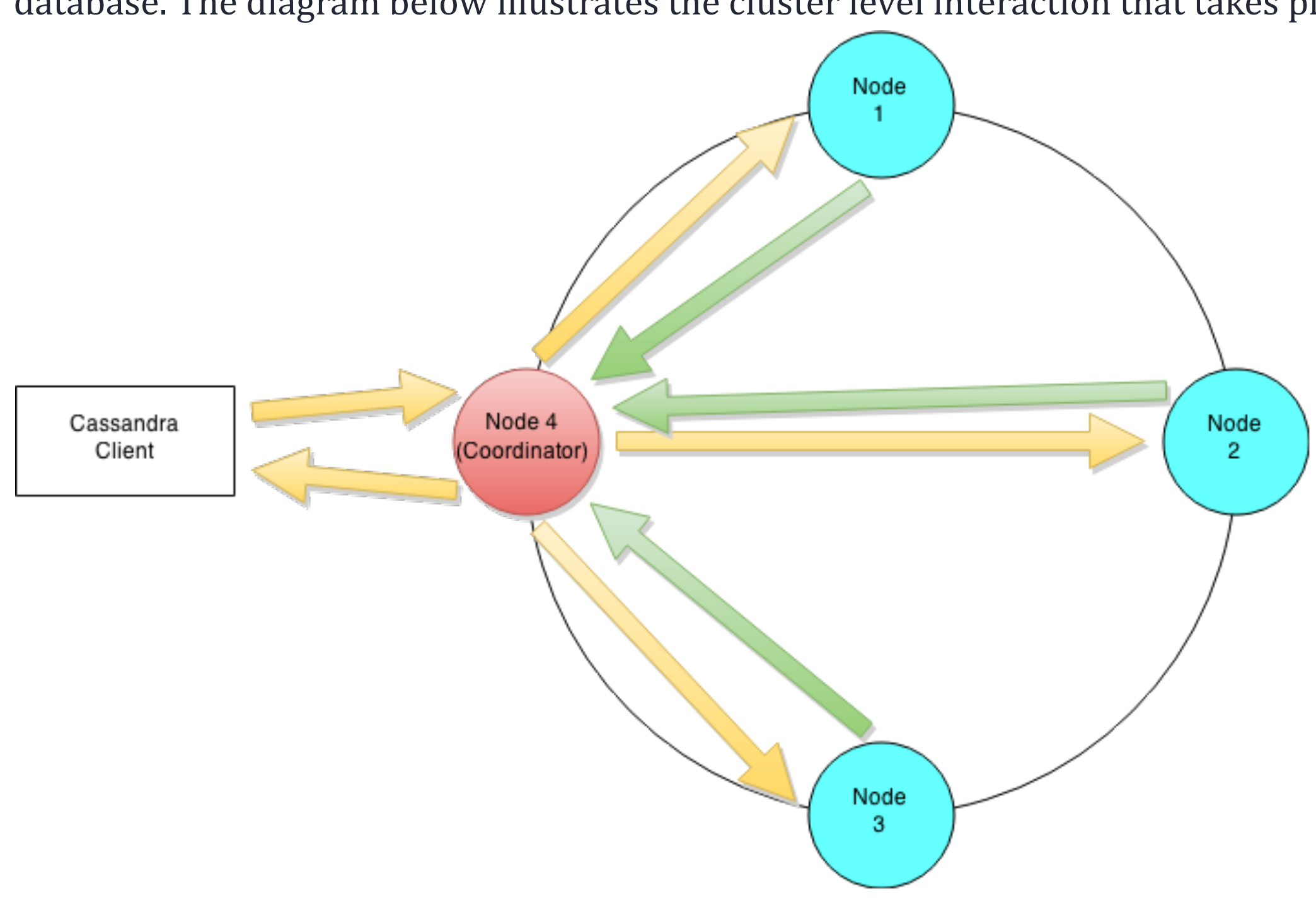
A Cassandra cluster is visualised as a ring because it uses a consistent hashing algorithm to distribute data. At start up each node is assigned a token range which determines its position in the cluster and the rage of data stored by the node. Each node receives a proportionate range of the token ranges 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.

Replication

Cassandra also replicates data according to the chosen replication strategy. The replication strategy determines placement of the replicated data. There are two main replication strategies used by Cassandra, Simple Strategy and the Network Topology Strategy. The first replica for the data is determined by the partitioner. The placement of the subsequent replicas is determined by the replication strategy. The simple strategy places the subsequent replicas on the next node in a clockwise manner. The network topology strategy works well when Cassandra is deployed across data centres. The network topology strategy is data centre aware and makes sure that replicas are not stored on the same rack. Cassandra uses snitches to discover the overall network overall topology. This information is used to efficiently route inter-node requests within the bounds of the replica placement strategy.

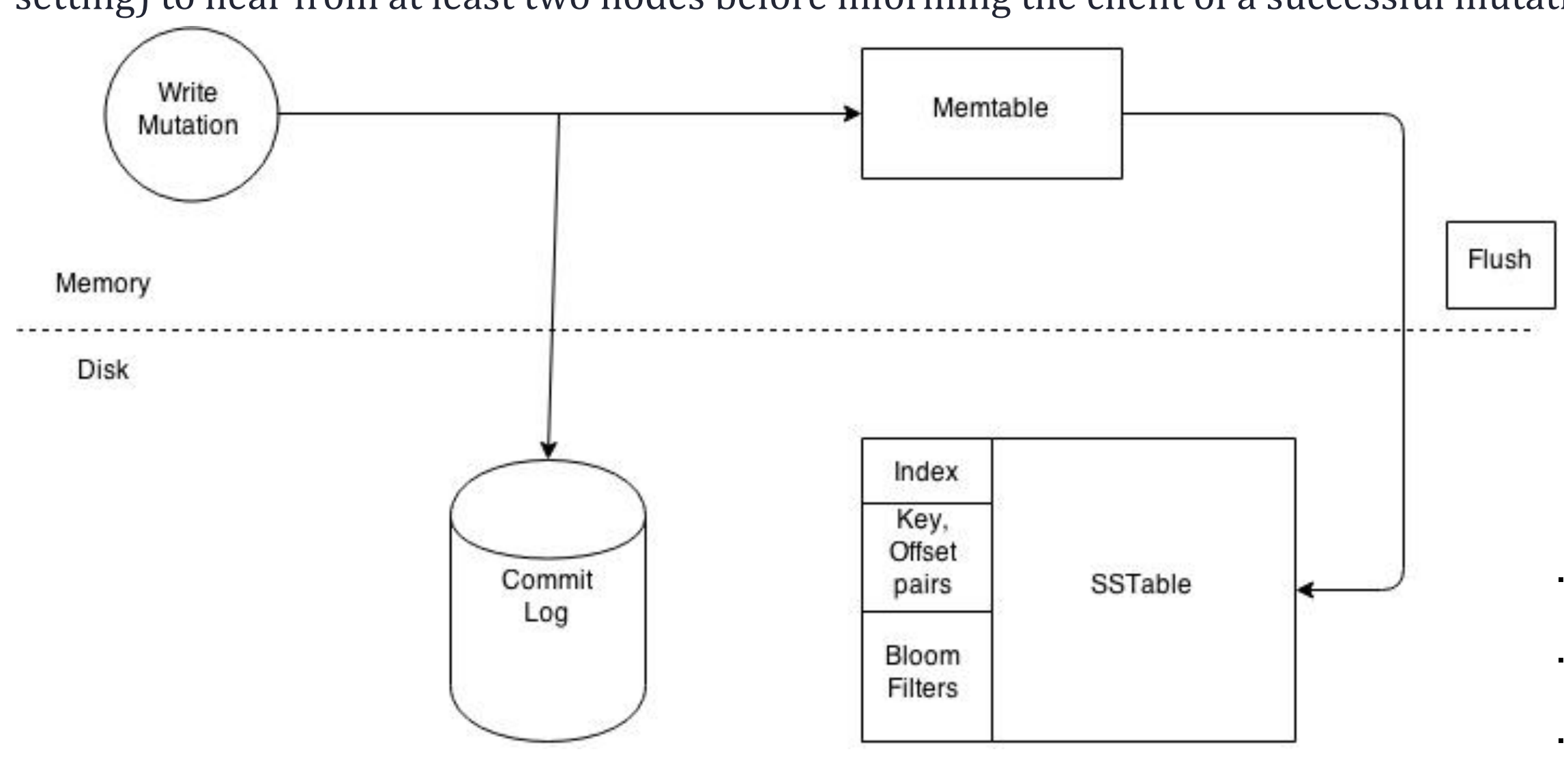
Cassandra Write Path

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.



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.



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:

. It reaches its maximum allocated size in memory

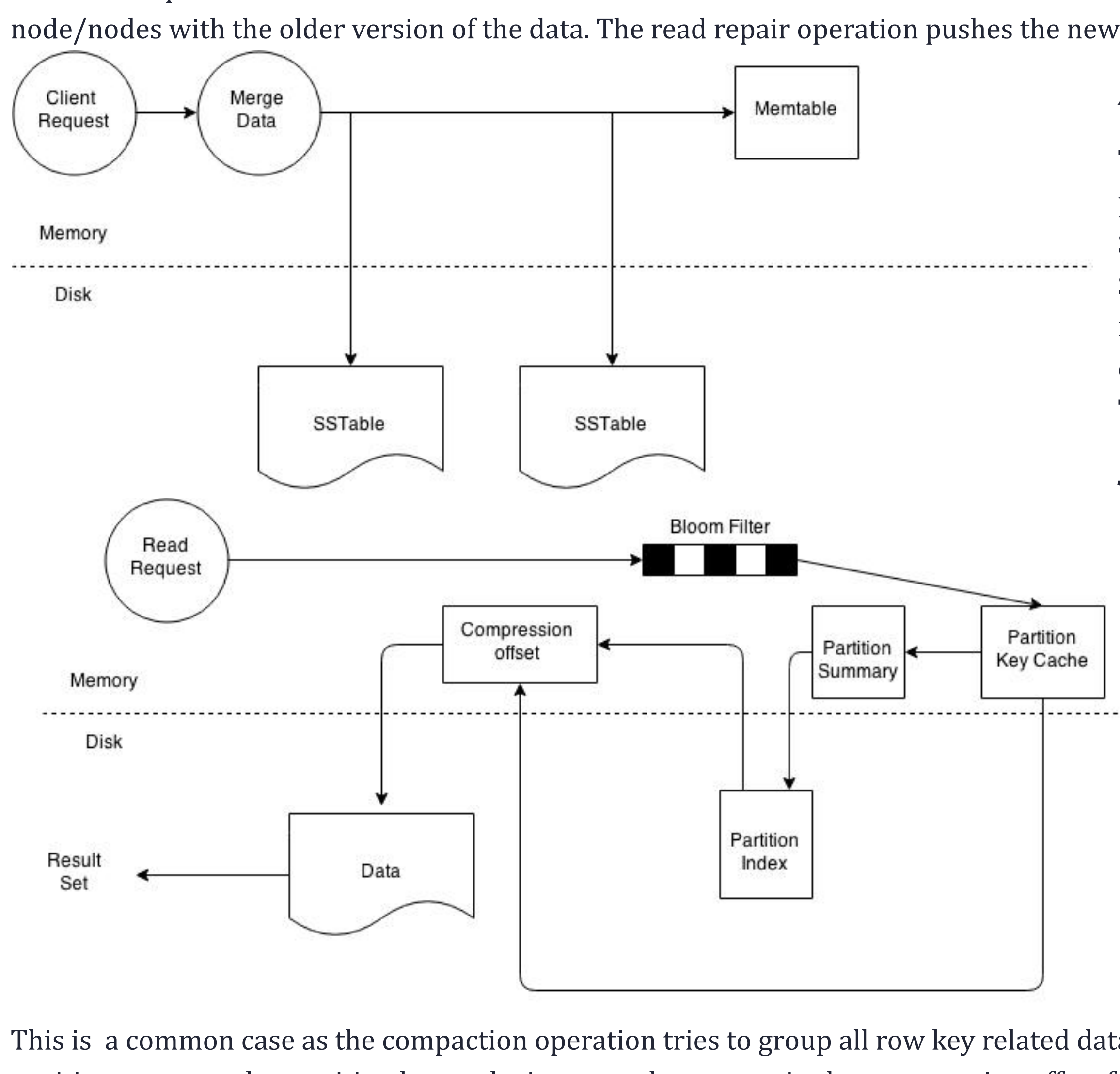
. The number of minutes a memtable can stay in memory elapses.

. 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.



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.

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.