Jorge Acetozi Follow

Nov 16, 2017 · 11 min read ·

Save

Cassandra Architecture and Write Path Anatomy

Cassandra is a NoSQL database that belongs to the Column Family NoSQL database category. It's an Apache project and it has an Enterprise version maintained by DataStax. Cassandra is written in Java and it's mainly used for time-series data such as metrics, IoT (Internet of Things), logs, chat messages, and so on. It is able to handle a massive amount of writes and reads and scale to thousands of nodes. Let's list out here some important Cassandra characteristics. Basically, Cassandra…

- mix ideas from Google's Big Table and Amazon's Dynamo.
- is based on peer-to-peer architecture. Every node is equal (can perform both reads and writes), therefore there is no master or slave node, that is, there is no master

single point of failure.

- does automatic partitioning and replication.
- has tunable write and read consistency for both read and write operations.
- is able to horizontally scale keeping linear scalability for both reads and writes.
- handles inter-node communication through the Gossip protocol.
- handles client communication through the CQL (Cassandra Query Language), which is very similar to SQL.

Coordinator

When a request is sent to any Cassandra node, this node acts as a proxy for the application (actually, the Cassandra driver) and the nodes involved in the request flow. This proxy node is called as the coordinator. The coordinator is responsible for managing the entire request path and to respond back to the client.

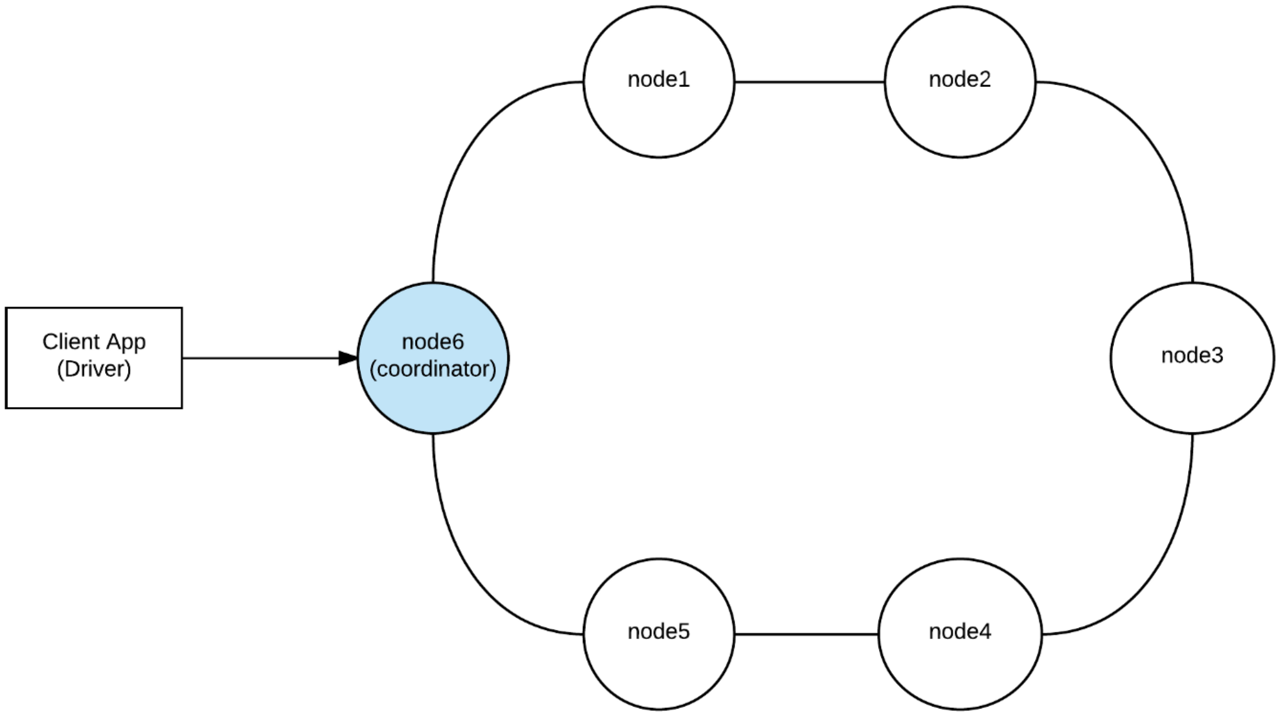

Figure 1 — Coordinator

Besides, sometimes when the coordinator forwards a write request to the replica nodes, they may happen to be unavailable at this very moment. In this case, the coordinator plays an important role implementing a mechanism called Hinted Handoff, which will be described in details later.

Partitioner

Basically, for each node in the Cassandra cluster (Cassandra ring) is assigned a range of tokens as shown in Figure 2 for a 6-node cluster (with imaginary tokens, of course).

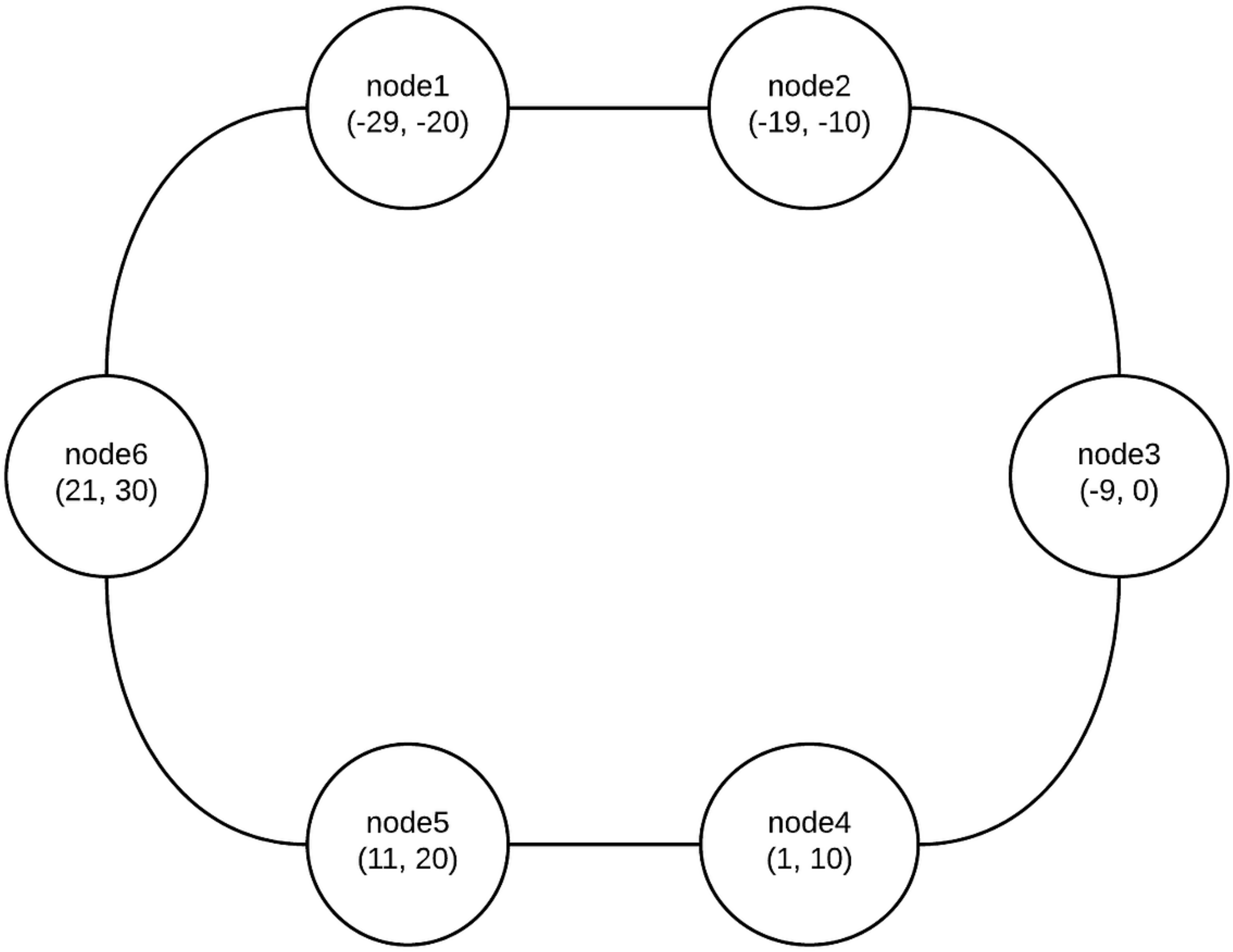

Figure — Token Ranges

Cassandra distributes data across the cluster using a Consistent Hashing algorithm and, starting from version 1.2, it also implements the concept of virtual nodes (vnodes), where each node owns a large number of small token ranges in order to improve token reorganization and avoid hotspots in the cluster, that is, some nodes storing much more data than the others. Virtual nodes also allow to add and remove nodes in the cluster more easily and manages the token assignment automatically for you so that you can enjoy a nice coffee when adding or removing a node instead of calculating and assigning new token ranges for each node (which is a very error-prone operation, by the way).

Well, that said, the partitioner is the component responsible for determining how to distribute the data across the nodes in the cluster given the partition key of a row.

Basically, it is a hash function for computing a token given the partition key.

Once the partitioner applies the hash function to the partition key and gets the token, it knows exactly which node is going to handle the request.

Let's consider a simple example: suppose a request is issued to node6 (that is, node6 is the coordinator for this request) with a row containing the partition key "jorge_acetozi". Suppose the partitioner applies the hash function to the partition key "jorge_acetozi" and gets the token -17. As figure 3 shows, node2 token ranges include -17, so this node will be the one handling the request.

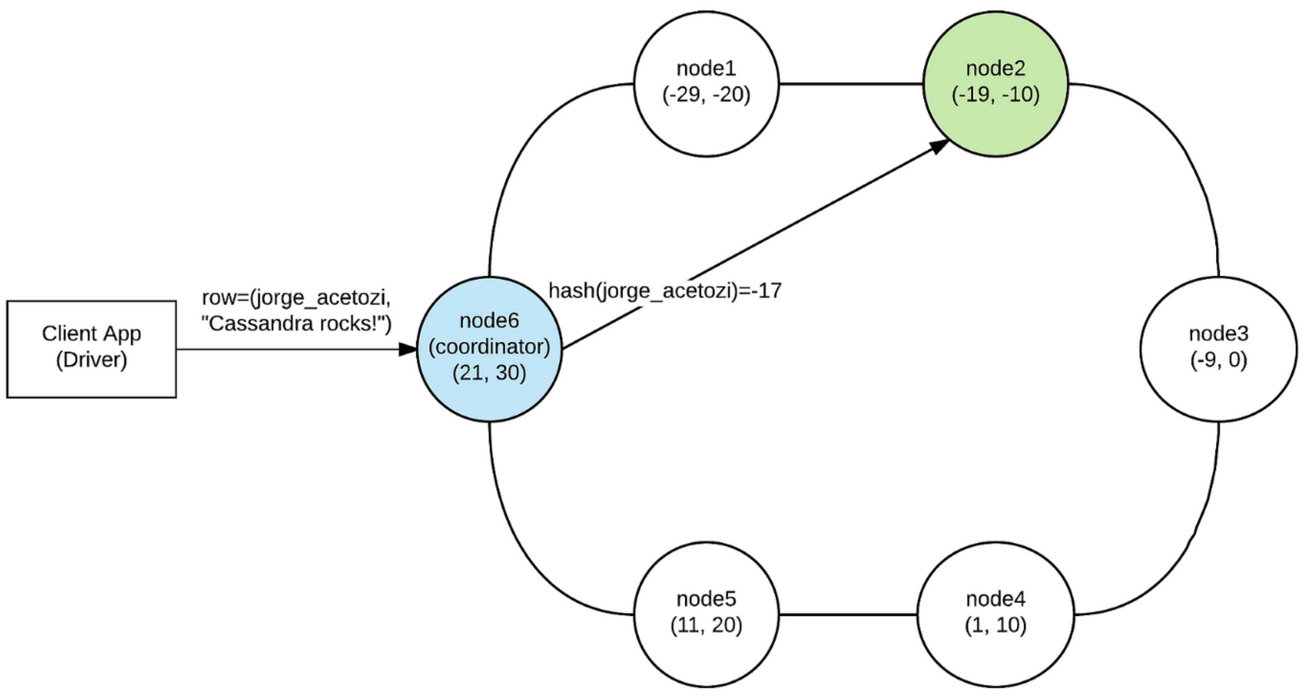

Figure 3 — Partitioner

Cassandra offers three types of partitioners: Murmur3Partitioner (which is the default), RandomPartitioner, and ByteOrderedPartitioner.

Replication

Life would be much easier if…

- Nodes never fail
- Networks had no latency
- People did not stumble on cables
- Amazon did not restart your instances

- Full GC meant “Full Guitar Concert”

And so on. Unfortunately, these things happen all the time and you already chose a software engineer career (your mother used to advise you to study hard and to become a doctor, but you chose to keep playing Counter-Strike instead. Now you are a software engineer, know what AK-47 means and have to care about stuff like that).

Fortunately, Cassandra offers automatic data replication and keeps your data redundant throughout different nodes in the cluster. This means that (in certain levels) you can even resist to node failure scenarios and your data would still be safe and available. But everything comes at a price, and the price of replication is consistency.

Replication Strategy

Basically, the coordinator uses the Replication Strategy to find out which nodes will be the replica nodes for a given request.

There are two replication strategies available:

SimpleStrategy: used for a single data center deployment (not recommended for production environment). It doesn’t consider the network topology. Basically, it just takes the partitioner’s decision (that is, the node that will handle the request first based on the token range) and places the remaining replicas clockwise in relation to this node. For example, in Figure 3, if the table replication factor was 3, which nodes would have been chosen by the SimpleStrategy to act as replicas (besides node2, which was already chosen by the partitioner)? That’s correct, node3 and node4! What if the replication factor was 4? Well, then node5 would also be included.

NetworkTopologyStrategy: used for multiple data centers deployment (recommended for production environment). It also takes the partitioner’s decision and places the remaining replicas clockwise, but it also takes into consideration the rack and data centers configuration.

Replication Factor

When you create a table (Column Family) in Cassandra, you specify the replication factor. The replication factor is the number of replicas that Cassandra will hold for this table in different nodes. If you specify REPLICATION_FACTOR=3, then your data will be replicated to 3 different nodes throughout the cluster. That provides fault tolerance and resilience because even if some nodes fail your data would still be safe and available.

Write Consistency Level

Do you still remember that when the client sends a request to a Cassandra node, this node is called a coordinator and acts as a proxy between the client and the replica nodes?

Well, when you write to a table in Cassandra (inserting data, for example), you can specify the write consistency level. The write consistency level is the number of replica nodes that have to acknowledge the coordinator that its local insert was successful (success here means that the data was appended to the commit log and written to the memtable). As soon as the coordinator gets WRITE_CONSISTENCY_LEVEL success acknowledgments from the replica nodes, it returns success back to the client and doesn't wait for the remaining replicas to acknowledge success.

For example, if an application issue an insert request with WRITE_CONSISTENCY_LEVEL=TWO to a table that is configured with REPLICATION_FACTOR=3, the coordinator will only return success to the application when two of the three replicas acknowledge success. Of course, this doesn't mean that the third replica will not write the data too; it will, but at this point, the coordinator would already have sent success back to the client.

There are many different types of write consistency levels you can specify in your write request. From the less consistent to full consistency: ANY, ONE, TWO, THREE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL.

Write Flow Example

For simplicity, suppose a write request is issued to a 6-node Cassandra cluster with the following characteristics:

- WRITE_CONSISTENCY_LEVEL=TWO
- TABLE_REPLICATION_FACTOR=3
- REPLICATION_STRATEGY=SimpleStrategy

First, the client sends the write request to the Cassandra cluster using the driver. We haven't discussed the role of the driver in this post (maybe in another post), but it plays a very important role as well. The driver is responsible for a lot of features such as asynchronous IO, parallel execution, request pipelining, connection pooling, auto node discovery, automatic reconnection, token awareness, and so on. For example, by using a driver that implements a token-aware policy, the driver reduces network hops by sending requests directly to the node that owns the data instead of sending it to a "random" coordinator.

As soon as the coordinator gets the write request, it applies the partitioner hash function to the partition key and uses the configured Replication Strategy in order to determine the TABLE_REPLICATION_FACTOR replica nodes that will actually write the data (in this sentence, replace TABLE_REPLICATION_FACTOR with the number 3). Figure 4 shows the replica nodes (in green) that will handle the write request.

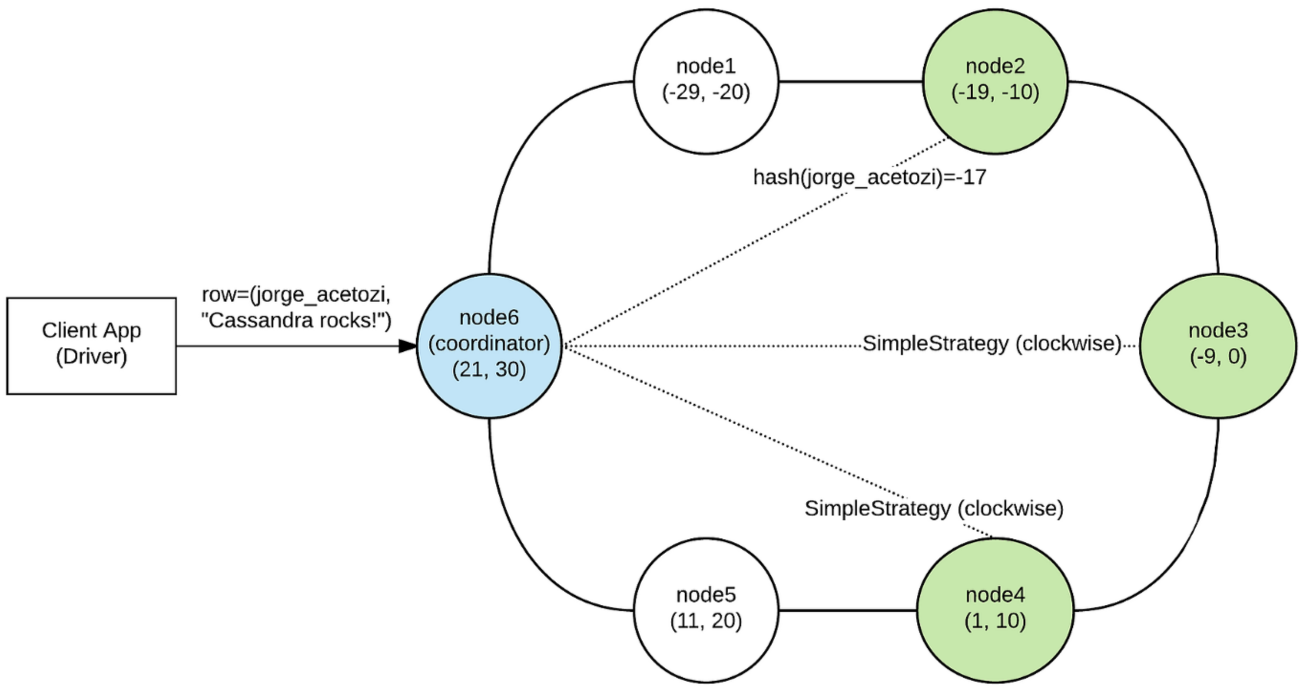

Figure 4 — Replica Nodes

Now, before the coordinator forwards the write request to all the 3 replica nodes, it will ask to the Failure Detector component how many of these replica nodes are actually available and compare it to the WRITE_CONSISTENCY_LEVEL provided in the request. If the number of replica nodes available is less than the WRITE_CONSISTENCY_LEVEL provided, the Failure Detector will immediately throw an Exception.

For our example, suppose the 3 replica nodes are available (that is, the Failure Detector will allow the request to continue) such as shown in Figure 5. Now, the coordinator will asynchronously forward the write request to all the replica nodes (in these case, the 3 replica nodes that were figured in the first step). As soon as WRITE_CONSISTENCY_LEVEL replica nodes acknowledge success (node2 and node4), the coordinator returns success back to the driver.

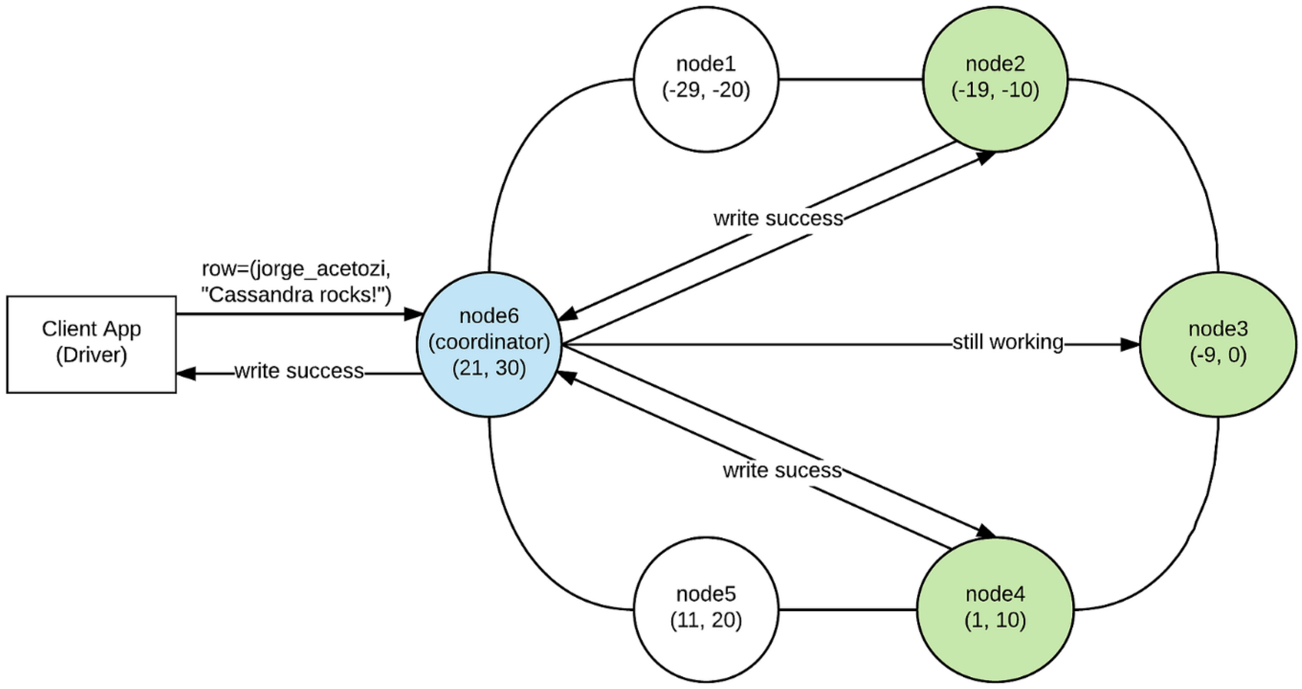

Figure 5 — Write Success

If the WRITE_CONSISTENCY_LEVEL for this request was THREE (or ALL), the coordinator would have to wait until node3 acknowledges success too, and of course that this write request would be slower.

So, basically…

- Do you need fault tolerance and high availability? Use replication.
- Just bear in mind that using replication means you will pay with consistency (for most of the cases, this is not a problem. Availability is often more important than consistency).
- If consistency is not an issue for your domain, perfect. If it is, just increase the consistency level, but then you will pay with higher latency.
- If you want fault tolerance and high availability, strong consistency and low latency, then you should be the client, not the software engineer (Lol).

Hinted Handoff

Suppose in the last example that only 2 of 3 replica nodes were available. In this case, the Failure Detector would still allow the request to continue as the number of available replica nodes is not less than the WRITE_CONSISTENCY_LEVEL provided. In this case, the coordinator would behave exactly as described before but there would be one additional step. The coordinator would write locally the hint (the write request blob along with some metadata) in the disk (hints directory) and would keep the hint there for 3 hours (by default) waiting for the replica node to become available again. If the replica node recovers within this period, the coordinator will send the hint to the replica node so that it can update itself and become consistent with the other replicas. If the replica node is offline for more than 3 hours, then a read repair is needed. This process is referred as Hinted Handoff.

Write Internals

In short, when a write request reaches a node, mainly two things happen:

1. The write request is appended to the commit log in the disk. This ensures data durability (the write request data would permanently survive even in a node failure scenario)

2. The write request is sent to the memtable (a structure stored in the memory). When the memtable is full, the data is flushed to a SSTable on disk using sequential I/O and the data in the commit log is purged.

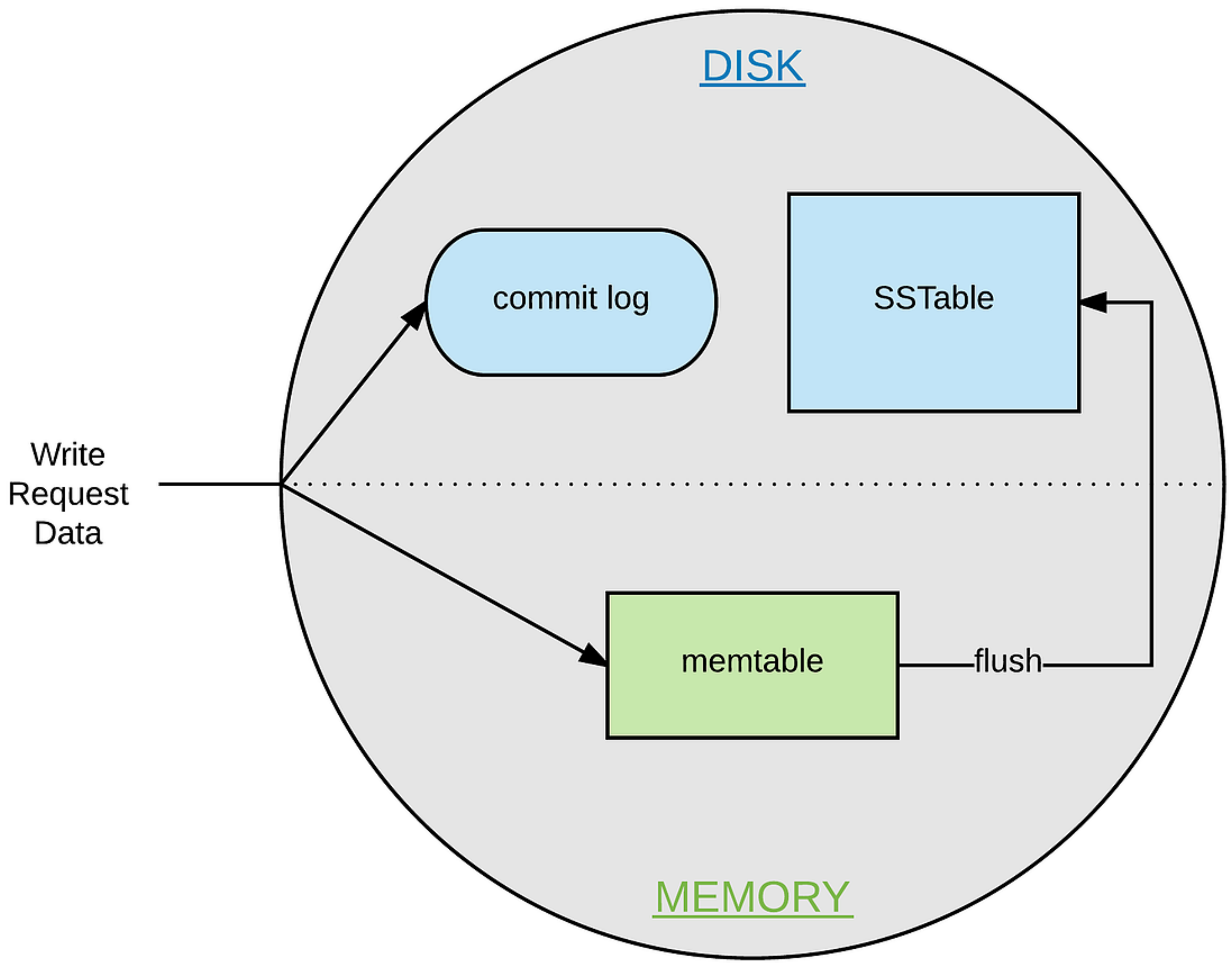

Figure 6 — Cassandra Node Internals

I really hope this article has been useful to you. If you enjoyed reading it, let me know if you would like to read another article diving into the Read Request Path. The Read Request Path is a little bit more complicated as it involves Snitches, Bloom Filter, Indexes, and so on, but it's pretty interesting as well.

If you are a software developer interested in how to use Cassandra in a realistic scenario coding a real-time chat application from the scratch, please take a look at my book: Pro Java Clustering and Scalability: Building Real-Time Apps with Spring, Cassandra, Redis, WebSocket and RabbitMQ

Thank you very much!