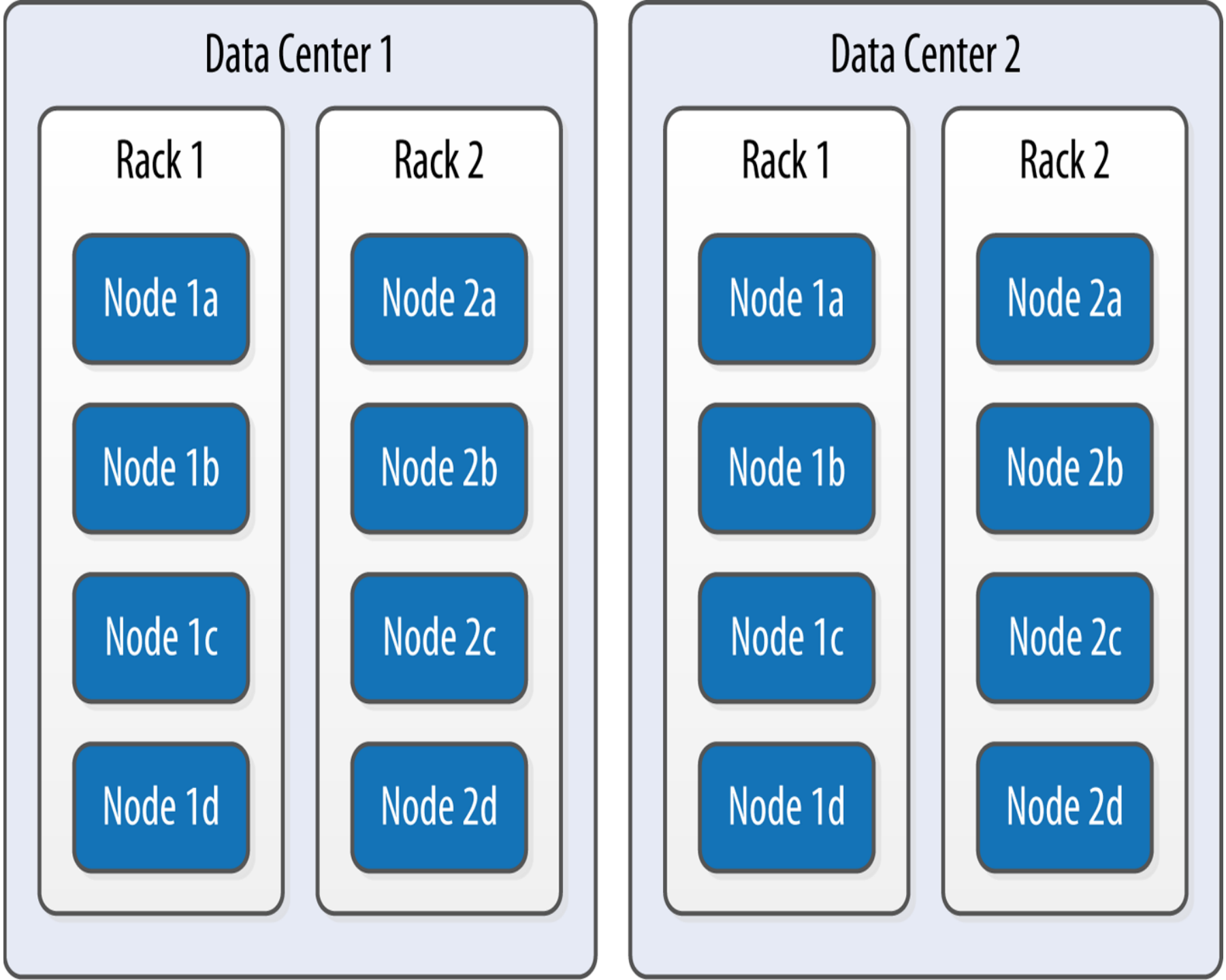
Unit 4 Syllabus

- **Data Modeling:** Conceptual Data modeling, RDBMS design, Logical Data Modeling, physical data Modeling, Evaluating and Refining, Defining database Schema.
- **Cassandra Architecture**: Data Centres and Racks, Gossip and Failure Detection, Snitches, Rings and Tokens, Virtual Nodes , Partitioners, Replication Strategies, Consistency Levels, Anti-Entropy, Repair, and Merkle Trees, Lightweight Transactions and Paxos, Memtables, SSTables, and Commit Logs, Caching.

The Cassandra Architecture

- **Data Centers and Racks**
- Cassandra is frequently used in systems spanning physically separate locations.
- Cassandra provides two levels of grouping that are used to describe the topology of a cluster: data center and rack.
- A *rack* is a logical set of nodes in close proximity to each other, perhaps on physical machines in a single rack of equipment.
- A *data center* is a logical set of racks, perhaps located in the same building and connected by reliable network.


Data Center 1
Rack 1
Node 1a
Node 1b
Node 1c
Node 1d
Rack 2
Node 2a
Node 2b
Node 2c
Node 2d
Data Center 2
Rack 1
Node 1a
Node 1b
Node 1c
Node 1d
Rack 2
Node 2a
Node 2b
Node 2c
Node 2d

Gossip and Failure Detection

- To support decentralization and partition tolerance, Cassandra uses a gossip protocol that allows each node to keep track of state information about the other nodes in the cluster.
- The gossiper runs every second on a timer.
- Gossip protocols (sometimes called *epidemic protocols*) generally assume a faulty network, are commonly employed in very large, decentralized network systems, and are often used as an automatic mechanism for replication in distributed databases.

Gossip and Failure Detection

- They take their name from the concept of human gossip, a form of communication in which peers can choose with whom they want to exchange information.
- The gossip protocol in Cassandra is primarily implemented by the org.apache.cassandra.gms.Gossiper class, which is responsible for managing gossip for the local node.
- When a server node is started, it registers itself with the gossiper to receive endpoint state information.
- Cassandra gossip is used for failure detection, the Gossiper class maintains a list of nodes that are alive and dead.

How the gossiper works

- 1. Once per second, the gossiper will choose a random node in the cluster and initialize a gossip session with it. Each round of gossip requires three messages.
- 2. The gossip initiator sends its chosen friend a GossipDigestSyn message.
- 3. When the friend receives this message, it returns a GossipDigestAck message.
- 4. When the initiator receives the ack message from the friend, it sends the friend a GossipDigestAck2 message to complete the round of gossip.
- When the gossiper determines that another endpoint is dead, it "convicts" that endpoint by marking it as dead in its local list and logging that fact

Failure Detection

- Cassandra has robust support for failure detection, as specified by a popular algorithm for distributed computing called Phi Accrual Failure Detector.
- This manner of failure detection originated at the Advanced Institute of Science and Technology in Japan in 2004.
- Accrual failure detection is based on two primary ideas.
- The first general idea is that failure detection should be flexible, which is achieved by decoupling it from the application being monitored.

Failure Detection

- The second and more novel idea challenges the notion of traditional failure detectors, which are implemented by simple “heartbeats” and decide whether a node is dead or not dead based on whether a heartbeat is received or not.
- But accrual failure detection decides that this approach is naive, and finds a place in between the extremes of dead and alive—a *suspicion level*.

Failure Detection

- Therefore, the failure monitoring system outputs a continuous level of “suspicion” regarding how confident it is that a node has failed.
- This is desirable because it can take into account fluctuations in the network environment.
- For example, just because one connection gets caught up doesn’t necessarily mean that the whole node is dead.
- So suspicion offers a more fluid and proactive indication of the weaker or stronger possibility of failure based on interpretation (the sampling of heartbeats), as opposed to a simple binary assessment

PHI THRESHOLD AND ACCRUAL FAILURE DETECTORS

- Accrual failure detectors output a value associated with each process (or node) called Phi.
- The Phi value represents the level of *suspicion* that a server might be down. The computation of this value is designed to be adaptive in the face of volatile network conditions, so it's not a binary condition that simply checks whether a server is up or down.
- The Phi convict threshold in the configuration adjusts the sensitivity of the failure detector.
- Lower values increase the sensitivity and higher values decrease it, but not in a linear fashion. With default settings, Cassandra can generally detect a failed node in about 10 seconds using this mechanism

Failure Detection

- Failure detection is implemented in Cassandra by the org.apache.cassandra.gms.FailureDetector class, which implements the org.apache.cassandra.gms.IFailureDetector interface.
- Together, they allow operations including:
- *isAlive(InetAddressAndPort)*
- What the detector will report about a given node's alive-ness.
- *interpret(InetAddressAndPort)*
- *report(InetAddressAndPort)*
- When a node receives a heartbeat, it invokes this method.

Snitches

- The job of a snitch is to provide information about your network topology so that Cassandra can efficiently route requests.
- The snitch will figure out where nodes are in relation to other nodes.
- The snitch will determine relative host proximity for each node in a cluster, which is used to determine which nodes to read and write from.
- When Cassandra performs a read, it must contact a number of replicas determined by the consistency level.

Snitches

- In order to support the maximum speed for reads, Cassandra selects a single replica to query for the full object, and asks additional replicas for hash values in order to ensure the latest version of the requested data is returned.
- The snitch helps to help identify the replica that will return the fastest, and this is the replica that is queried for the full data.
- The default snitch (the SimpleSnitch) is topology unaware; that is, it does not know about the racks and data centers in a cluster, which makes it unsuitable for multiple data center deployments.
- For this reason, Cassandra comes with several snitches for different network topologies and cloud environments, including Amazon EC2, Google Cloud, and Apache Cloudstack.

Snitches

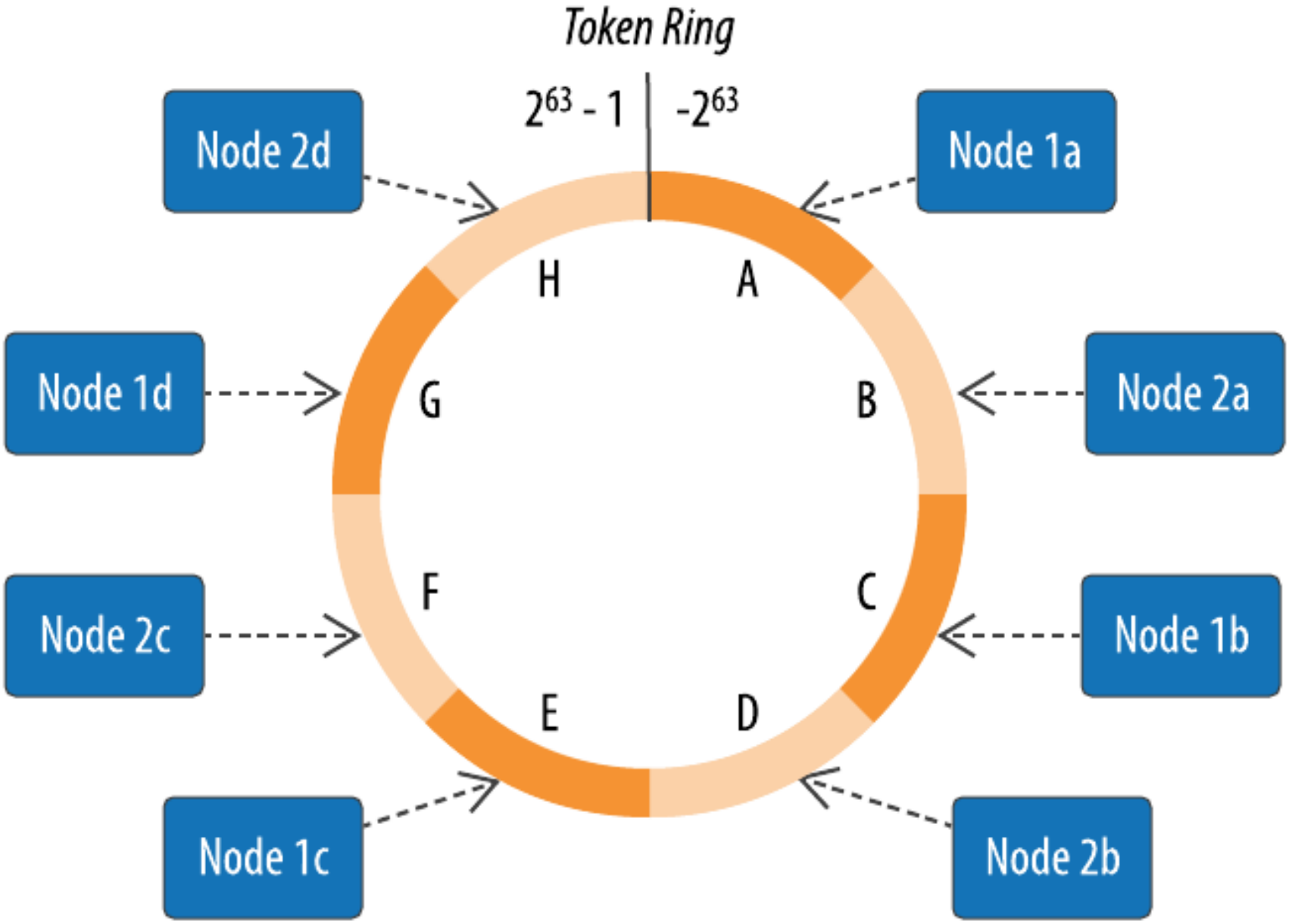
- The snitches can be found in the package org.apache.cassandra.locator.
- Each snitch implements the IEndpointSnitch interface.
- Cassandra provides a pluggable way to statically describe your cluster's topology, it also provides a feature called *dynamic snitching* that helps optimize the routing of reads and writes over time.
- Your selected snitch is wrapped with another snitch called the DynamicEndpointSnitch.
- The dynamic snitch gets its basic understanding of the topology from the selected snitch. It then monitors the performance of requests to the other nodes, even keeping track of things like which nodes are performing compaction

Snitches

- The dynamic snitching implementation uses a modified version of the Phi failure detection mechanism used by gossip.
- The *badness threshold* is a configurable parameter that determines how much worse a preferred node must perform than the best-performing node in order to lose its preferential status.
- The scores of each node are reset periodically in order to allow a poorly performing node to demonstrate that it has recovered and reclaim its preferred status

Rings and Tokens

- Cassandra represents the data managed by a cluster as a *ring*. Each node in the ring is assigned one or more ranges of data described by a *token*, which determines its position in the ring.
- For example, in the default configuration, a token is a 64-bit integer ID used to identify each partition.
- This gives a possible range for tokens from −2 to 2 −1.
- A node claims ownership of the range of values less than or equal to each token and greater than the last token of the previous node, known as a *token range*.
- The node with the lowest token owns the range less than or equal to its token and the range greater than the highest token, which is also known as the *wrapping range*


Token Ring
$2^{63} - 1$
-2^{63}
Node 2d
Node 1a
H
A
Node 1d
G
B
Node 2a
F
C
Node 2c
Node 1b
E
D
Node 1c
Node 2b

Rings and Tokens

- Data is assigned to nodes by using a hash function to calculate a token for the partition key.
- This partition key token is compared to the token values for the various nodes to identify the range, and therefore the node, that owns the data.
- Token ranges are represented by the org.apache.cassandra.dht.Range class.

Rings and Tokens

- The CQL language provides a token() function that we can use to request the value of the token corresponding to a partition key, in this case the last_name:
- cqlsh:my_keyspace> SELECT last_name, first_name, token(last_name) FROM user;
- last_name | first_name | system.token(last_name)
- ----------+-----------+----------------------
- Rodriguez | Mary | -7199267019458681669
- Scott | Isaiah | 1807799317863611380
- Nguyen | Bill | 60007101983668045 98
- Nguyen | Wanda | 60007101983668045 98
- (5 rows)

Virtual Nodes

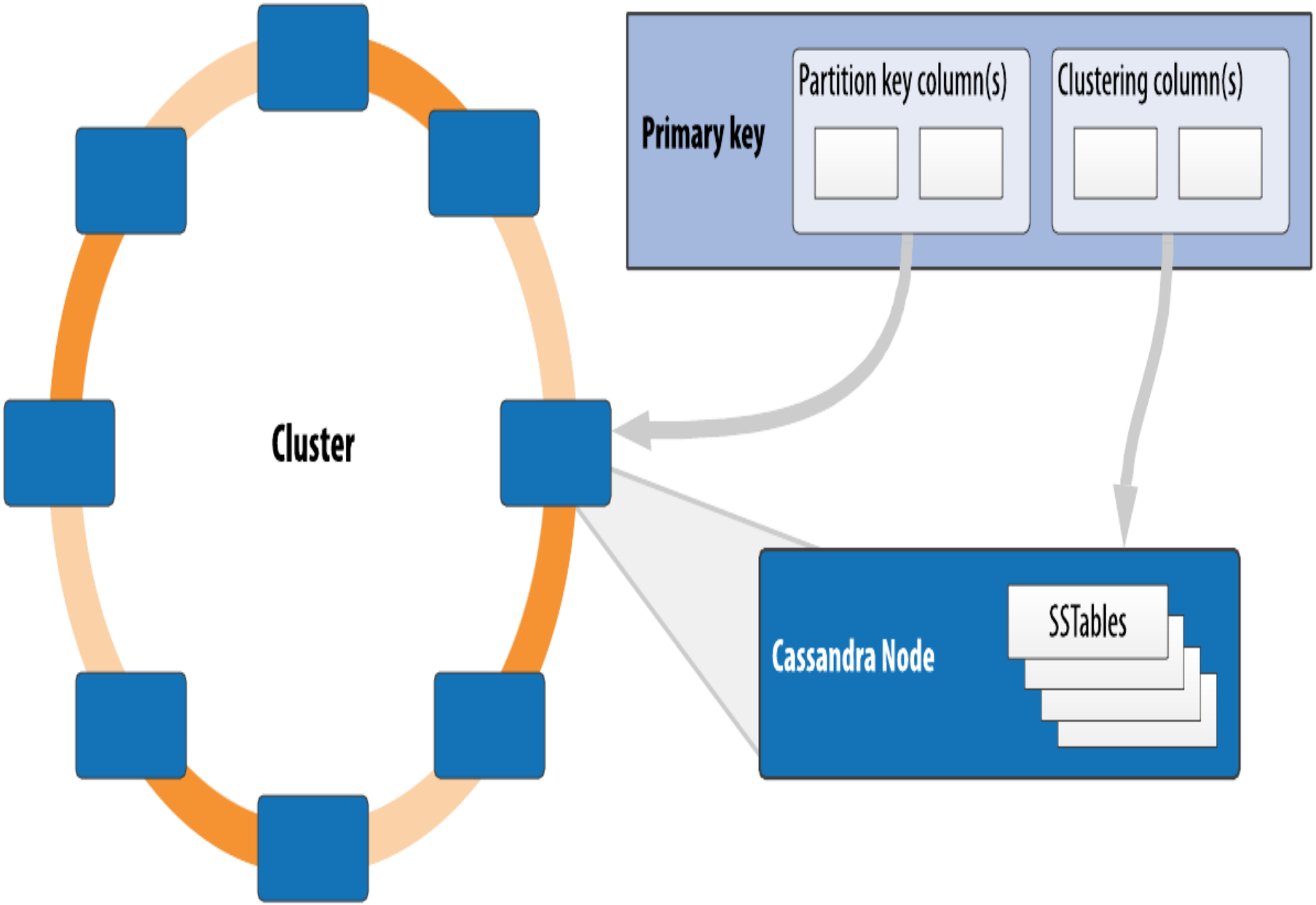
- Early versions of Cassandra assigned a single token (and therefore by implication, a single token range) to each node, in a fairly static manner, requiring you to calculate tokens for each node.
- Although there are tools available to calculate tokens based on a given number of nodes, it was still a manual process to configure the initial_token property for each node in the *cassandra.yaml* file.
- Instead of assigning a single token to a node, the token range is broken up into multiple smaller ranges.
- Each physical node is then assigned multiple tokens. Historically, each node has been assigned 256 of these tokens, meaning that it represents 256 virtual nodes

Virtual Nodes

- Vnodes make it easier to maintain a cluster containing heterogeneous machines.
- For nodes in your cluster that have more computing resources available to them, you can increase the number of vnodes by setting the num_tokens property in the *cassandra.yaml* file.
- Conversely, you might set num_tokens lower to decrease the number of vnodes for less capable machines.
- Cassandra automatically handles the calculation of token ranges for each node in the cluster in proportion to their num_tokens value.
- Token assignments for vnodes are calculated by the
- org.apache.cassandra.dht.tokenallocator.ReplicationAwareToken Allocator class

Partitioners

- A *partitioner* determines how data is distributed across the nodes in the cluster.
- Cassandra organizes rows in partitions.
- Each row has a partition key that is used to identify the partition to which it belongs.
- A partitioner, then, is a hash function for computing the token of a partition key.
- Each row of data is distributed within the ring according to the value of the partition key token.
- Any clustering columns that may be present in the primary key are used to determine the ordering of rows within a given node that owns the token representing that partition.


Cluster
Primary key
Partition key column(s)
Clustering column(s)
Cassandra Node
SSTables

Partitioners

- A client may connect to any node in the cluster to initiate a read or write query.
- This node is known as the *coordinator node*. The coordinator identifies which nodes are replicas for the data that is being written or read and forwards the queries to them.
- For a write, the coordinator node contacts all replicas, as determined by the consistency level and replication factor, and considers the write successful
- when a number of replicas commensurate with the consistency level acknowledge the write.
- For a read, the coordinator contacts enough replicas to ensure the required consistency level is met, and returns the data to the client

Replication Strategies

- A node serves as a *replica* for different ranges of data.
- If one node goes down, other replicas can respond to queries for that range of data.
- Cassandra replicates data across nodes in a manner transparent to the user, and the *replication factor* is the number of nodes in your cluster that will receive copies (replicas) of the same data. If your replication factor is 3, then three nodes in the ring will have copies of each row.

Replication Strategies

- The first replica will always be the node that claims the range in which the token falls, but the remainder of the replicas are placed according to the *replication strategy* (sometimes also referred to as the *replica placement strategy*).
- For determining replica placement, Cassandra implements the Gang of Four strategy pattern, which is outlined in the common abstract class org.apache.cassandra.locator.AbstractReplicationStrategy, allowing different implementations of an algorithm (different strategies for accomplishing the same work).
- Each algorithm implementation is encapsulated inside a single class that extends the AbstractReplicationStrategy.

Replication Strategies

- The SimpleStrategy places replicas at consecutive nodes around the ring, starting with the node indicated by the partitioner.
- The NetworkTopologyStrategy allows you to specify a different replication factor for each data center. Within a data center, it allocates replicas to different racks in order to maximize availability.
- The NetworkTopologyStrategy is recommended for keyspaces in production deployments, even those that are initially created with a single data center, since it is more straightforward to add an additional data center should the need arise.

Consistency Levels

- Cassandra provides tuneable consistency levels that allow you to make these trade-offs at a fine-grained level.
- You specify a consistency level on each read or write query that indicates how much consistency you require.
- A higher consistency level means that more nodes need to respond to a read or write query, giving you more assurance that the values present on each replica are the same

Consistency Levels

- For read queries, the consistency level specifies how many replica nodes must respond to a read request before returning the data.
- For write operations, the consistency level specifies how many replica nodes must respond for the write to be reported as successful to the client.
- Because Cassandra is eventually consistent, updates to other replica nodes may continue in the background.

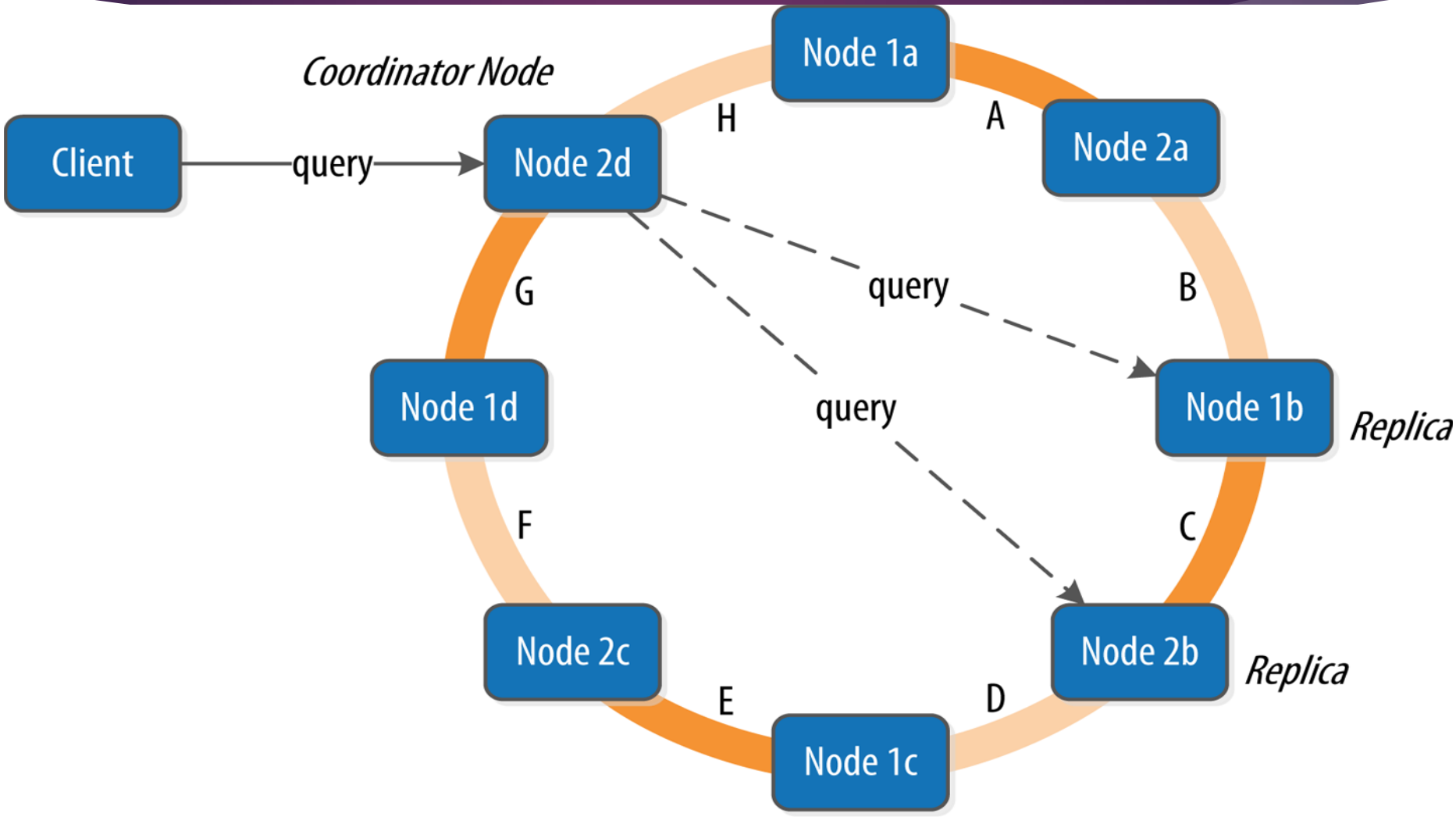
Consistency Levels

- The available consistency levels include ONE, TWO, and THREE, each of which specify an absolute number of replica nodes that must respond to a request.
- The QUORUM consistency level requires a response from a majority of the replica nodes.
- This is sometimes expressed as
- Q = floor(RF/2 + 1)
- In this equation, *Q* represents the number of nodes needed to achieve quorum for a replication factor *RF*.
- It may be simpler to illustrate this with a couple of examples: if *RF* is 3, *Q* is 2; if *RF* is 4, *Q* is 3; if *RF* is 5, *Q* is 3, and so on.

Consistency Levels

- Consistency is tuneable in Cassandra because clients can specify the desired consistency level on both reads and writes.
- There is an equation that is popularly used to represent the way to achieve *strong consistency* in Cassandra: $R + W > RF$ = *strong consistency*.
- In this equation, R, W, and RF are the read replica count, the write replica count, and the replication factor, respectively; all client reads will see the most recent write in this scenario, and you will have strong consistency

Queries and Coordinator Nodes



Queries and Coordinator Nodes

- A client may connect to any node in the cluster to initiate a read or write query.
- This node is known as the *coordinator node*.
- The coordinator identifies which nodes are replicas for the data that is being written or read and forwards the queries to them.
- For a write, the coordinator node contacts all replicas, as determined by the consistency level and replication factor, and considers the write successful when a number of replicas commensurate with the consistency level acknowledge the write.

Queries and Coordinator Nodes

- For a read, the coordinator contacts enough replicas to ensure the required consistency level is met, and returns the data to the client.

Hinted Handoff

- Consider the following scenario: a write request is sent to Cassandra, but a replica node where the write properly belongs is not available due to network partition, hardware failure, or some other reason.
- In order to ensure general availability of the ring in such a situation, Cassandra implements a feature called *hinted handoff*. You might think of a *hint* as a little Post-it
- Note that contains the information from the write request

Hinted Handoff

- If the replica node where the write belongs has failed, the coordinator will create a hint, which is a small reminder that says, “I have the write information that is intended for node B. I’m going to hang on to this write, and I’ll notice when node B comes back online; when it does, I’ll send it the write request.”
- That is, once it detects via gossip that node B is back online, node A will “hand off” to node B the “hint” regarding the write. Cassandra holds a separate hint for each partition that is to be written

Anti-Entropy, Repair, and Merkle Trees

- Cassandra uses an *anti-entropy* protocol as an additional safeguard to ensure consistency.
- Anti-entropy protocols are a type of gossip protocol for repairing replicated data.
- They work by comparing replicas of data and reconciling differences observed between the replicas.

Anti-Entropy, Repair, and Merkle Trees

- Replica synchronization is supported via two different modes known as *read repair* and *anti-entropy repair*. Read repair refers to the synchronization of replicas as data is read.
- Cassandra reads data from multiple replicas in order to achieve the requested consistency level, and detects if any replicas have out-of-date values

Anti-Entropy, Repair, and Merkle Trees

- Anti-entropy repair (sometimes called *manual repair*) is a manually initiated
- operation performed on nodes as part of a regular maintenance process. This
- type of repair is executed by using a tool called nodetool

Anti-Entropy, Repair, and Merkle Trees

- Running nodetool repair causes Cassandra to execute a *validation compaction*
- During a validation compaction, the server initiates a TreeRequest/TreeReponse conversation to exchange Merkle trees with neighboring replicas.
- The Merkle tree is a hash representing the data in that table. If the trees from the different nodes don't match, they have to be reconciled (or "repaired") to determine the latest data values they should all be set to.
- This tree comparison validation is the responsibility of the
- org.apache.cassandra.service.reads.AbstractReadExecutor class

Lightweight Transactions and Paxos

- Cassandra provides the ability to achieve strong consistency by specifying sufficiently high consistency levels on writes and reads.
- However, strong consistency is not enough to prevent race conditions in cases where clients need to read, then write data.

Lightweight Transactions and Paxos

- Imagine we are building a client that wants to manage user records as part of an account management application. In creating a new user account, we'd like to make sure that the user record doesn't already exist, lest we unintentionally overwrite existing user data.
- So first we do a read to see if the record exists, and then only perform the create if the record doesn't exist.
- The behavior we're looking for is called *linearizable consistency*, meaning that we'd like to guarantee that no other client can come in between our read and write queries with their own modification.

Lightweight Transactions and Paxos

- Cassandra supports a *lightweight transaction* (LWT) mechanism that provides linearizable consistency.
- Cassandra's LWT implementation is based on Paxos. Paxos is a consensus algorithm that allows distributed peer nodes to agree on a proposal, without requiring a leader to coordinate a transaction.
- Paxos and other consensus algorithms emerged as alternatives to traditional two-phase commit-based approaches to distributed transactions

Lightweight Transactions and Paxos

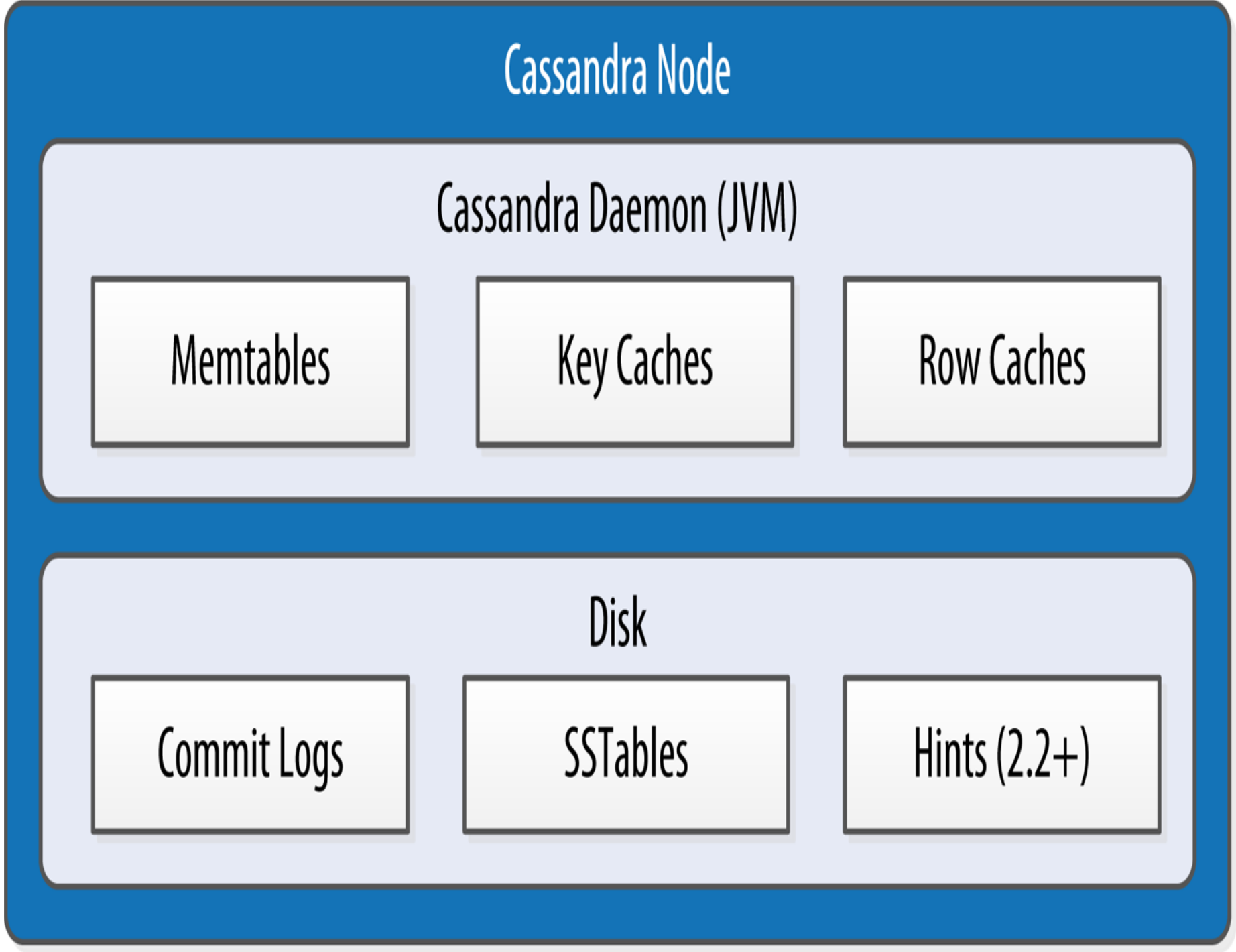
- The basic Paxos algorithm consists of two stages: prepare/promise and propose/accept.
- To modify data, a coordinator node can propose a new value to the replica nodes, taking on the role of leader. Other nodes may act as leaders simultaneously for other modifications.
- Each replica node checks the proposal, and if the proposal is the latest it has seen, it promises to not accept proposals associated with any prior proposals.
- Each replica node also returns the last proposal it received that is still in progress. If the proposal is approved by a majority of replicas, the leader commits the proposal, but with the caveat that it must first commit any in-progress proposals that preceded its own proposal

Lightweight Transactions and Paxos

- The Cassandra implementation extends the basic Paxos algorithm to support the desired read-before-write semantics (also known as *check-and-set*), and to allow the state to be reset between transactions. It does this by inserting two additional phases into the algorithm, so that it works as follows:
- 1. Prepare/Promise
- 2. Read/Results
- 3. Propose/Accept
- 4. Commit/Ack

Memtables, SSTables, and Commit Logs

- Cassandra stores data both in memory and on disk to provide both high performance and durability.
- In this section, we'll focus on Cassandra's *storage engine* and its use of constructs called *memtables*, *SSTables*, and *commit logs* to support the writing and reading of data from tables.


Cassandra Node
Cassandra Daemon (JVM)
Memtables
Key Caches
Row Caches
Disk
Commit Logs
SSTables
Hints (2.2+)

Memtables, SSTables, and Commit Logs

- When a node receives a write operation, it immediately writes the data to a *commit log*.
- The commit log is a crash-recovery mechanism that supports Cassandra's durability goals.
- A write will not count as successful on the node until it's written to the commit log, to ensure that if a write operation does not make it to the in-memory store it will still be possible to recover the data

Memtables, SSTables, and Commit Logs

- If you shut down the node or it crashes unexpectedly, the commit log can ensure that data is not lost.
- That's because the next time you start the node, the commit log gets replayed. In fact, that's the only time the commit log is read; clients never read from it.
- cqlsh> DESCRIBE KEYSPACE my_keyspace ;
- CREATE KEYSPACE my_keyspace WITH replication =
- {'class': 'SimpleStrategy',
- 'replication_factor': '1'} AND durable_writes = true;

Memtables, SSTables, and Commit Logs

- After it's written to the commit log, the value is written to a memory-resident data structure called the *memtable*.
- Each memtable contains data for a specific table When the number of objects stored in the memtable reaches a threshold, the contents of the memtable are flushed to disk in a file called an *SSTable*.
- A new memtable is then created. This flushing is a nonblocking operation; multiple memtables may exist for a single table, one current and the rest waiting to be flushed. They typically should not have to wait very long, as the node should flush them very quickly unless it is overloaded

Memtables, SSTables, and Commit Logs

- Each commit log maintains an internal bit flag to indicate whether it needs flushing.
- When a write operation is first received, it is written to the commit log and its bit flag is set to 1.
- There is only one bit flag per table, because only one commit log is ever being written to across the entire server.
- All writes to all tables will go into the same commit log, so the bit flag indicates whether a particular commit log contains anything that hasn't been flushed for a particular table.
- Once the memtable has been properly flushed to disk, the corresponding commit log's bit flag is set to 0, indicating that the commit log no longer has to maintain that data for durability purposes

Memtables, SSTables, and Commit Logs

- Once a memtable is flushed to disk as an SSTable, it is immutable and cannot be changed by the application.
- Despite the fact that SSTables are compacted, this compaction changes only their on-disk representation; it essentially performs the “merge” step of a mergesort into new files and removes the old files on success

Bloom Filters

- Bloom filters are used to boost the performance of reads. They are named for their inventor, Burton Bloom.
- Bloom filters are very fast, nondeterministic algorithms for testing whether an element is a member of a set.
- They are nondeterministic because it is possible to get a false-positive read from a Bloom filter, but not a false-negative. Bloom filters work by mapping the values in a data set into a bit array and condensing a larger data set into a digest string using a hash function

Caching

- The *key cache* stores a map of partition keys to row index entries, facilitating faster read access into SSTables stored on disk.
- The key cache is stored on the JVM heap.
- The *row cache* caches entire rows and can greatly speed up read access for frequently accessed rows, at the cost of more memory usage. The row cache is stored in off-heap memory.

Caching

- The *chunk cache* was added in the 3.6 release to store uncompressed chunks of data read from SSTable files that are accessed frequently. The chunk cache is stored in off-heap memory

Caching

- The *counter cache* was added in the 2.1 release to improve counter performance by reducing lock contention for the most frequently accessed counters.
- By default, key and counter caching are enabled, while row caching is disabled, as it requires more memory. Cassandra saves its caches to disk periodically in order to warm them up more quickly on a node restart