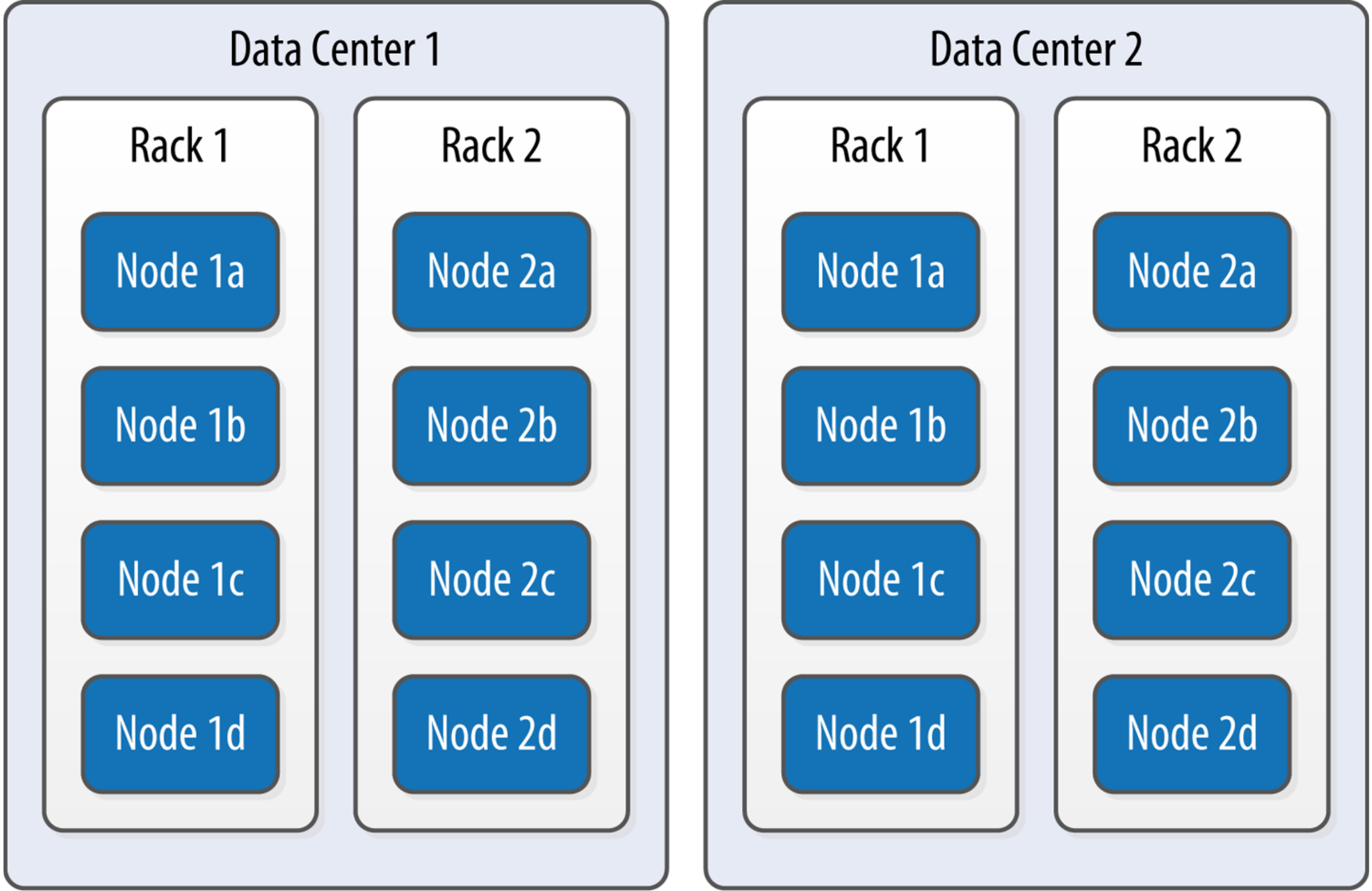
Cassandra Architecture

ANJU MUNOTH

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

Data Centers and Racks

- Out of the box, Cassandra comes with a simple default configuration of a single data center ("datacenter1") containing a single rack ("rack1").
- Cassandra leverages the information you provide about your cluster's topology to determine where to store data, and how to route queries efficiently.
- Cassandra stores copies of your data in the data centers you request to maximize availability and partition tolerance, while preferring to route queries to nodes in the local data center to maximize performance.

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.
- Gossiper runs every second on a timer.
- Gossip protocols (also called epidemic protocols) -Employed in very large, decentralized network systems,
- Often used as an automatic mechanism for replication in distributed databases.
- 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.

Gossip and Failure Detection

- Gossip protocol in Cassandra - 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.
- Because Cassandra gossip is used for failure detection, the Gossiper class maintains a list of nodes that are alive and dead.

How the gossiper works?

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.

The gossip initiator sends its chosen friend a GossipDigestSyn message.

When the friend receives this message, it returns a GossipDigestAck message.

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.

Snitches

- Job of a snitch is to provide information about your network topology so that Cassandra can efficiently route requests.
- Snitch will figure out where nodes are in relation to other nodes.
- Determine relative host proximity for each node in a cluster, which is used to determine which nodes to read and write from.
- As an example, let's examine how the snitch participates in a read operation.
- When Cassandra performs a read, it must contact a number of replicas determined by the consistency level.
- 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 which is queried for the full data.

Snitches

- 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.
- The snitches can be found in the package org.apache.cassandra.locator.
- Each snitch implements the IEndpointSnitch interface.

Dynamic Snitching

Selected snitch is wrapped with another snitch called the DynamicEndpointSnitch.

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.

Performance data is used to select the best replica for each query.

This enables Cassandra to avoid routing requests to replicas that are busy or performing poorly.

Rings and Tokens

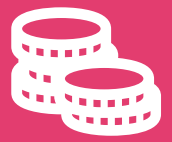
Cassandra represents the data managed by a cluster as a ring.

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 −2^63 to 2^63−1.

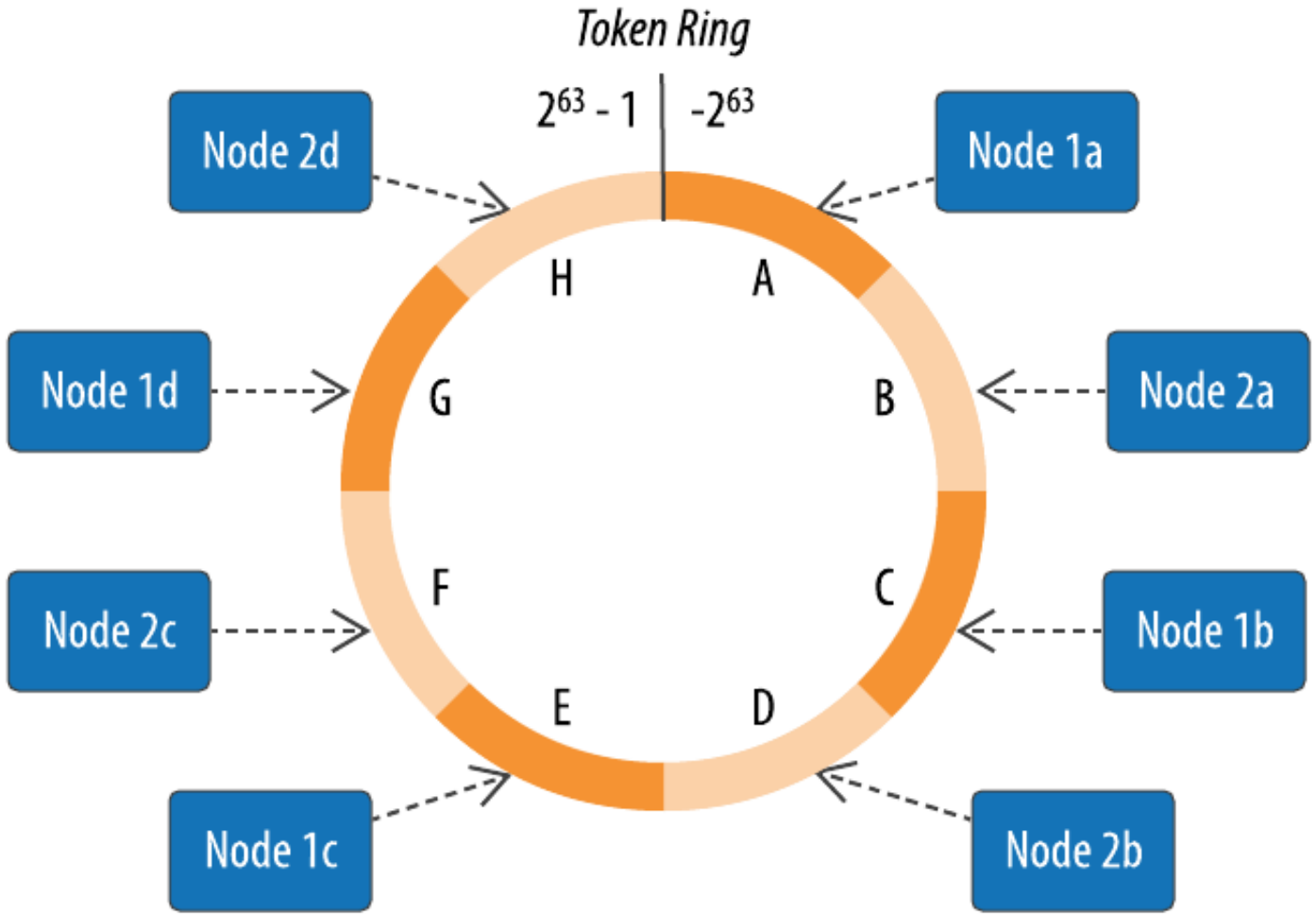
Rings and Tokens



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.

In this way, the tokens specify a complete ring.


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

Consistent hashing

Consistent hashing allows distribution of data across a cluster to minimize reorganization when nodes are added or removed.

Consistent hashing partitions data based on the partition key.

For example, if you have the following data:

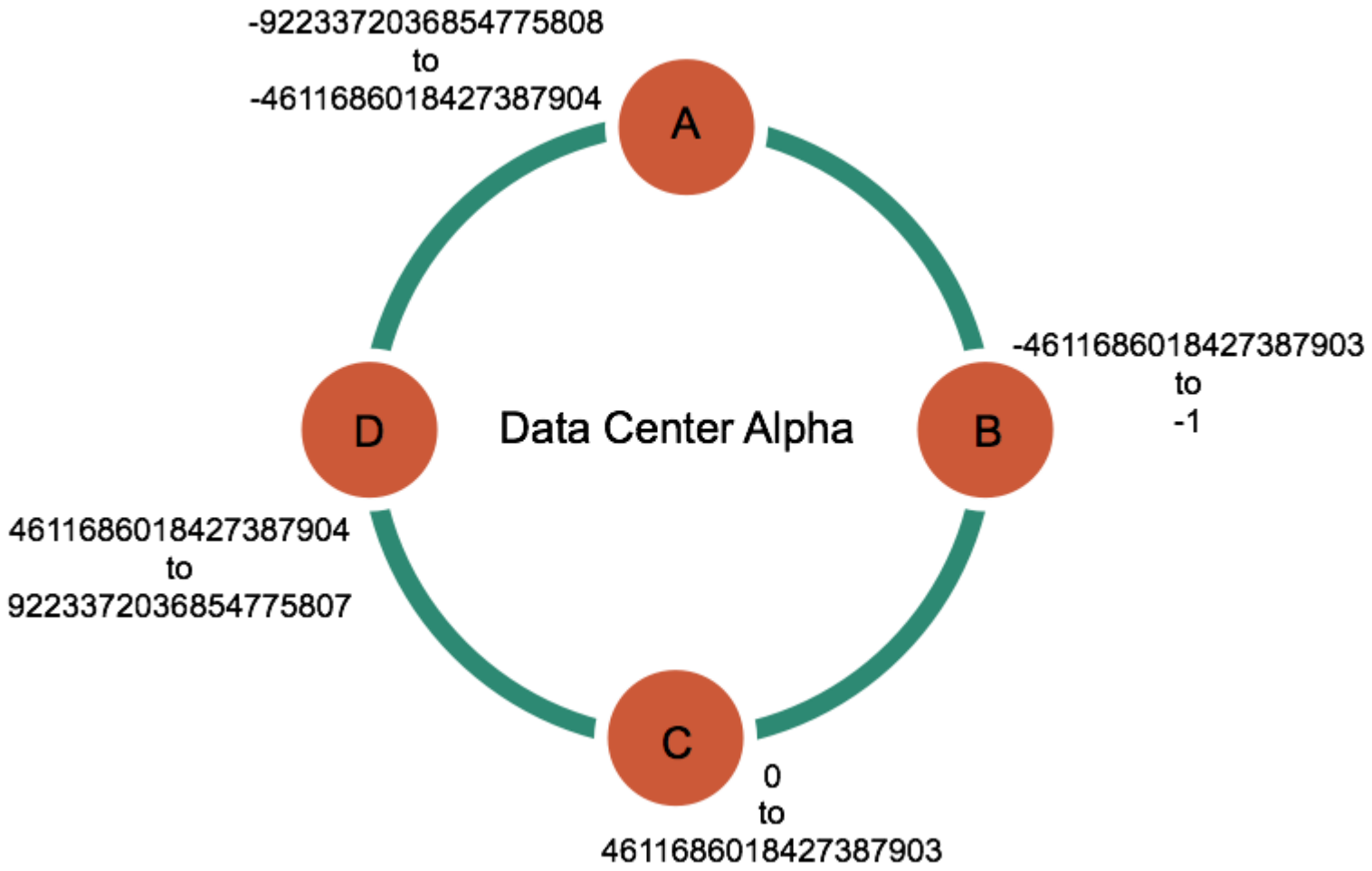
name	age	car	gender
jim	36	camaro	M
carol	37	bmw	F
johnny	12		M
suzy	10		F

Cassandra assigns a hash value to each partition key:

Partition key	Murmur3 hash value
jim	-2245462676723223822
carol	7723358927203680754
johnny	-6723372854036780875
suzy	1168604627387940318

Each node in the cluster is responsible for a range of data based on the hash value.

Hash values in a four node cluster



Cassandra places the data on each node according to the value of the partition key and the range that the node is responsible for.
For example, in a four node cluster, the data in this example is distributed as follows:

Node	Start range	End range	Partition key	Hash value
A	-9223372036854775808	-4611686018427387904	johnny	-6723372854036780875
B	-4611686018427387903	-1	jim	-2245462676723223822
C	0	4611686018427387903	suzy	1168604627387940318
D	4611686018427387904	9223372036854775807	carol	7723358927203680754

Virtual nodes

- Virtual nodes, known as Vnodes, distribute data across nodes at a finer granularity than can be easily achieved if calculated tokens are used.
- Vnodes simplify many tasks in Cassandra:
- Tokens are automatically calculated and assigned to each node.
- Rebalancing a cluster is automatically accomplished when adding or removing nodes.
- When a node joins the cluster, it assumes responsibility for an even portion of data from the other nodes in the cluster.
- If a node fails, the load is spread evenly across other nodes in the cluster.
- Rebuilding a dead node is faster because it involves every other node in the cluster.
- The proportion of vnodes assigned to each machine in a cluster can be assigned, so smaller and larger computers can be used in building a cluster.
- Note: Cassandra recommends using 8 vnodes (tokens). Using 8 vnodes distributes the workload between systems with a ~10% variance and has minimal impact on performance.

How data is distributed across a cluster (using virtual nodes)

- Prior to Cassandra 1.2, you had to calculate and assign a single token to each node in a cluster.
- Each token determined the node's position in the ring and its portion of data according to its hash value.
- In Cassandra 1.2 and later, each node is allowed many tokens. The new paradigm is called virtual nodes (vnodes).
- Vnodes allow each node to own a large number of small partition ranges distributed throughout the cluster.
- Vnodes also use consistent hashing to distribute data but using them doesn't require token generation and assignment.

Virtual Nodes

- 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.
- Data is assigned to nodes by using a hash function to calculate a token for the partition key.
- 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.

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.

This also made adding or replacing a node an expensive operation, as rebalancing the cluster required moving a lot of data.

Historically, each node has been assigned 256 of these tokens, meaning that it represents 256 virtual nodes

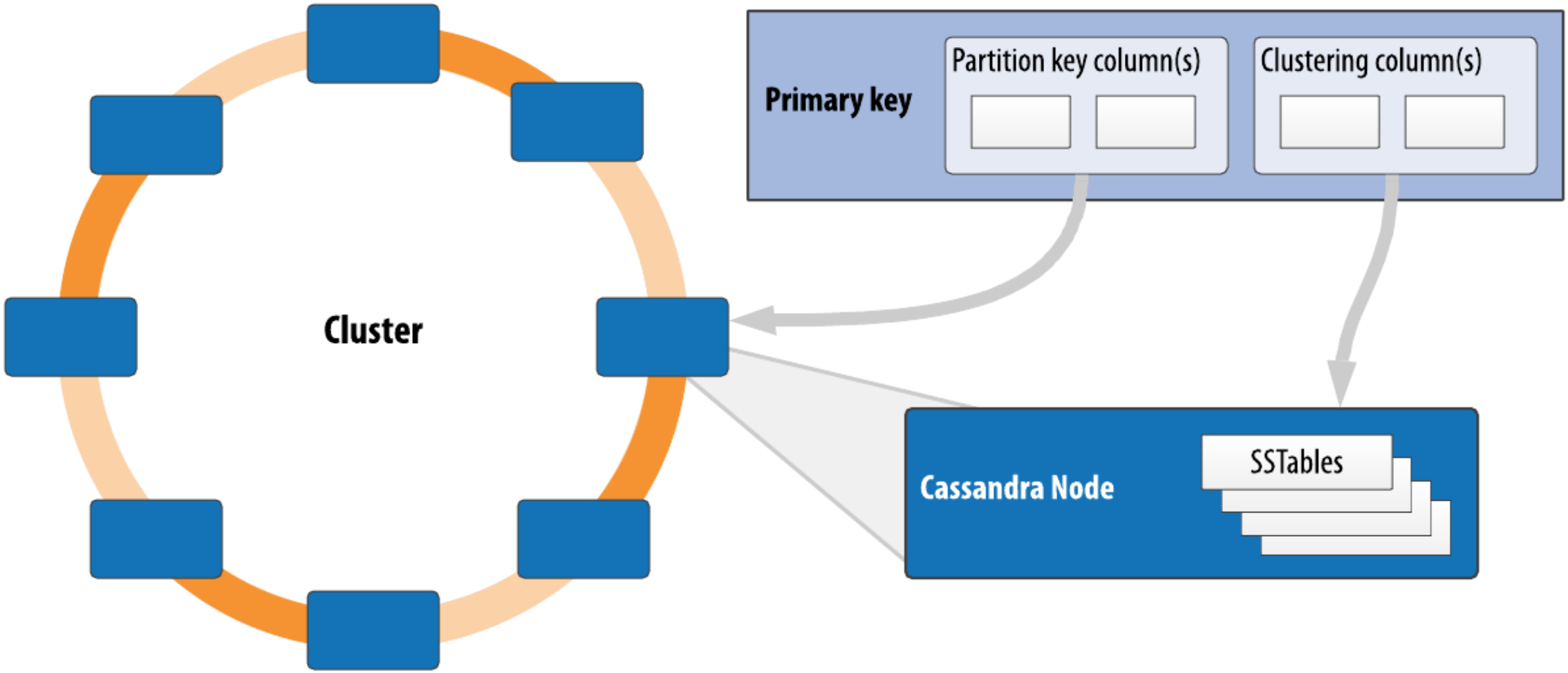
Virtual nodes have been enabled by default since 2.0.

Virtual Nodes

- Easier to maintain a cluster containing heterogeneous machines.
- For nodes in cluster that have more computing resources available to them, 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.ReplicationAwareTokenAllocator class.
- Advantage of virtual nodes is that they speed up some of the more heavyweight Cassandra operations such as bootstrapping a new node, decommissioning a node, and repairing a node.
- This is because the load associated with operations on multiple smaller ranges is spread more evenly across the nodes in the cluster.

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.
- Role of the partitioner is to compute the token based on the partition key columns.
- 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

- Cassandra provides several different partitioners in the org.apache.cassandra.dht package (DHT stands for distributed hash table).
- Murmur3Partitioner -Added in 1.2;Default partitioner since then;
  - Efficient Java implementation on the murmur algorithm developed by Austin Appleby.
  - Generates 64-bit hashes.
- Previous default -RandomPartitioner.
- Can also create your own partitioner by implementing the org.apache.cassandra.dht.IPartitioner class and placing it on Cassandra's classpath.
- Note, however, that the default partitioner is not frequently changed in practice
- Can't change the partitioner after initializing a cluster.

Partitioners

- Cassandra offers the following partitioners that can be set in the cassandra.yaml file.

1. **Murmur3Partitioner** (default): uniformly distributes data across the cluster based on MurmurHash hash values.
2. **RandomPartitioner**: uniformly distributes data across the cluster based on MD5 hash values.
3. **ByteOrderedPartitioner**: keeps an ordered distribution of data lexically by key bytes

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.
- For a replication factor of 3 - three nodes in the ring will have copies of each row.
- **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

- SimpleStrategy and NetworkTopologyStrategy.
- **SimpleStrategy** - places replicas at consecutive nodes around the ring, starting with the node indicated by the partitioner.
- **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.
- 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.

Replication Strategies

- Asymmetrical replication groupings are also possible.
- For example, can have three replicas in one datacenter to serve real-time application requests and use a single replica elsewhere for running analytics.

- Replication strategy is defined per keyspace, and is set during keyspace creation

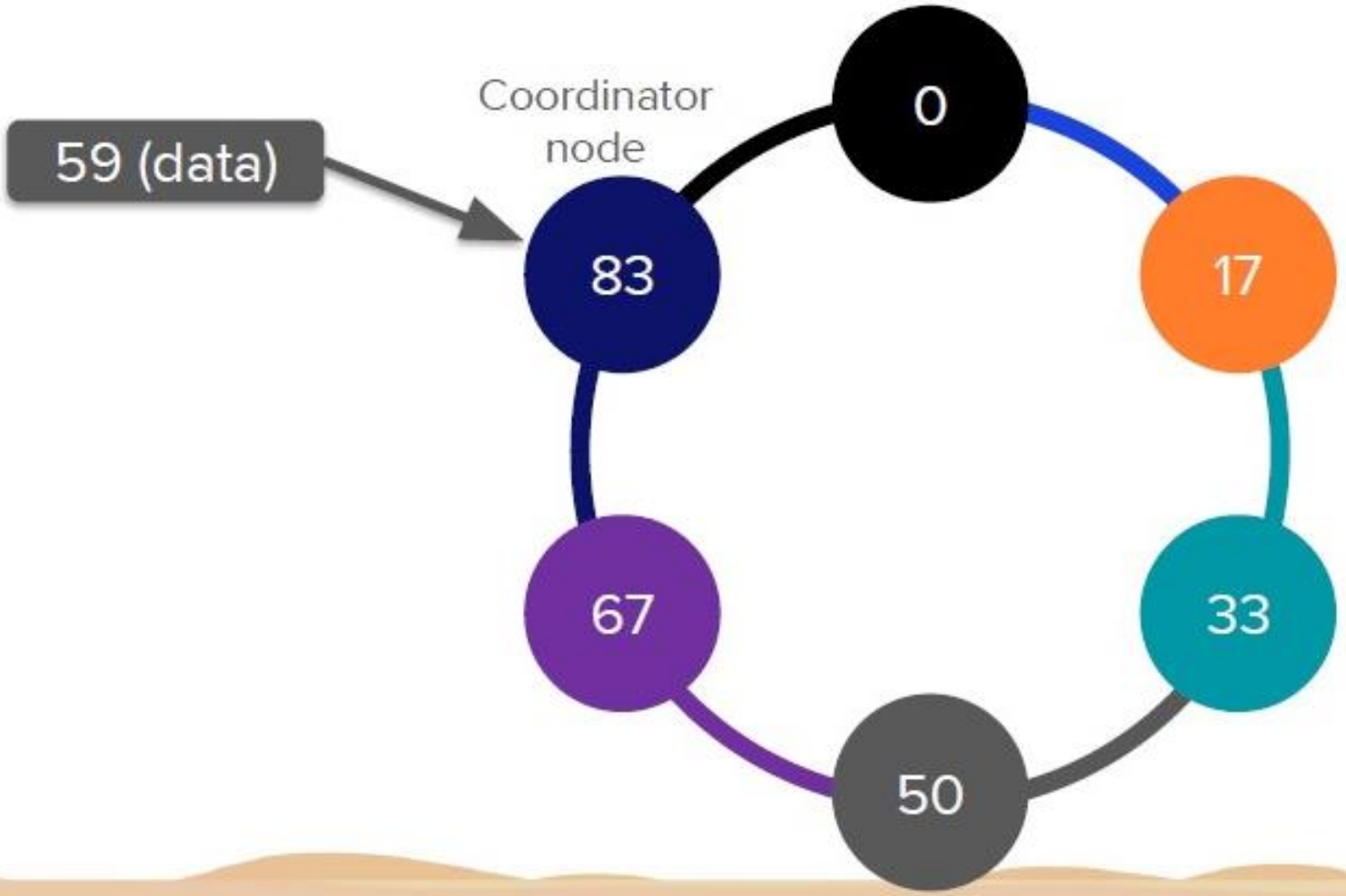
SimpleStrategy

- Use only for a single datacenter and one rack.
- SimpleStrategy places the first replica on a node determined by the partitioner.
- Additional replicas are placed on the next nodes clockwise in the ring without considering topology (rack or datacenter location).

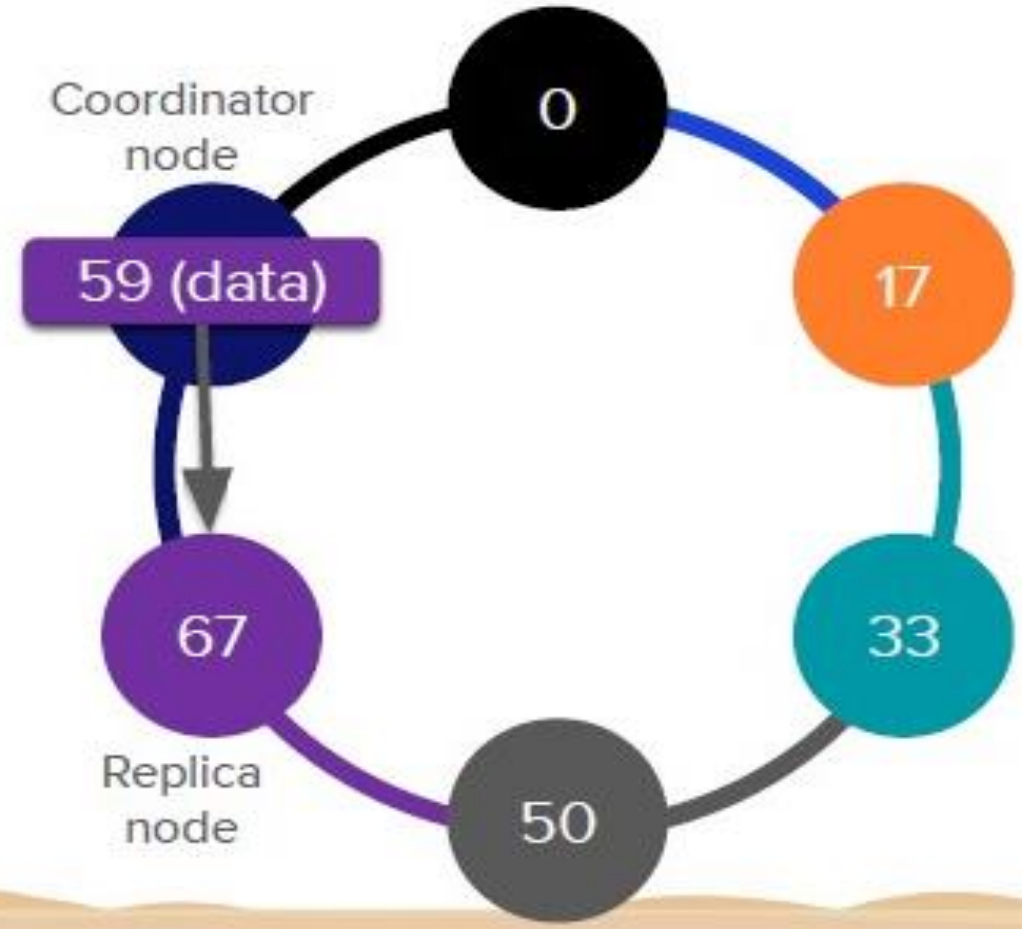
NetworkTopologyStrategy

- Use NetworkTopologyStrategy when you have (or plan to have) your cluster deployed across multiple datacenters.
- This strategy specifies how many replicas you want in each datacenter.
- NetworkTopologyStrategy places replicas in the same datacenter by walking the ring clockwise until reaching the first node in another rack.
- NetworkTopologyStrategy attempts to place replicas on distinct racks because nodes in the same rack (or similar physical grouping) often fail at the same time due to power, cooling, or network issues.
- When deciding how many replicas to configure in each datacenter, the two primary considerations are (1) being able to satisfy reads locally, without incurring cross data-center latency, and (2) failure scenarios.

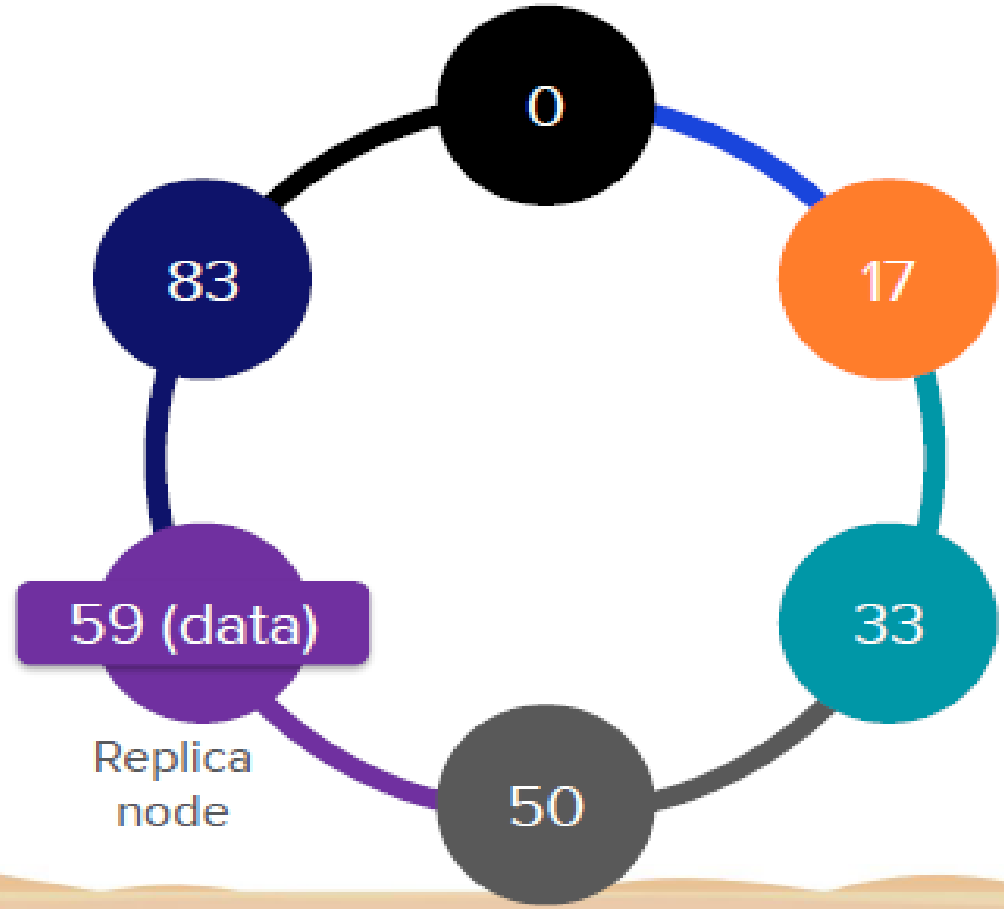
How the Ring Works



How the Ring Works

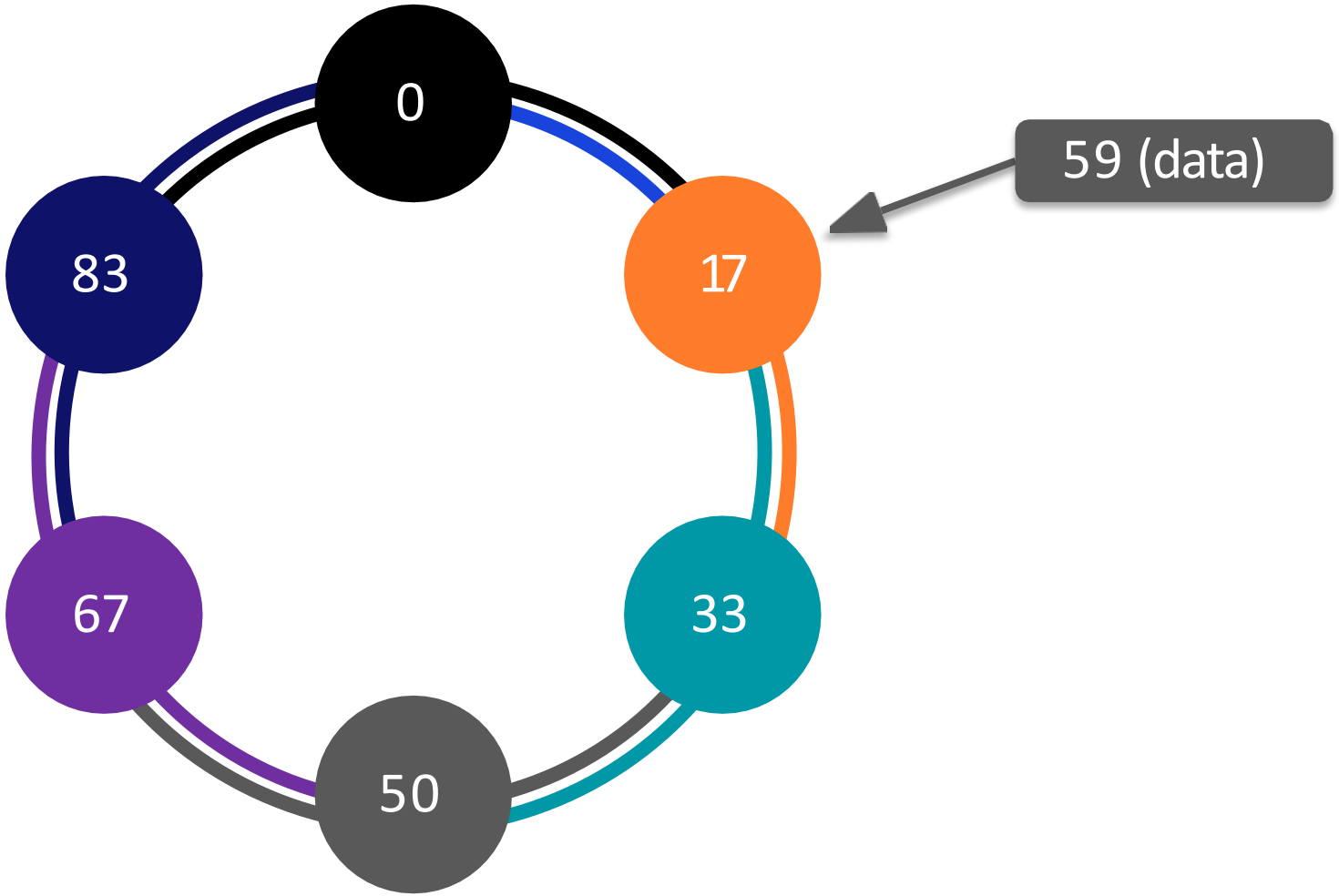


How the Ring Works



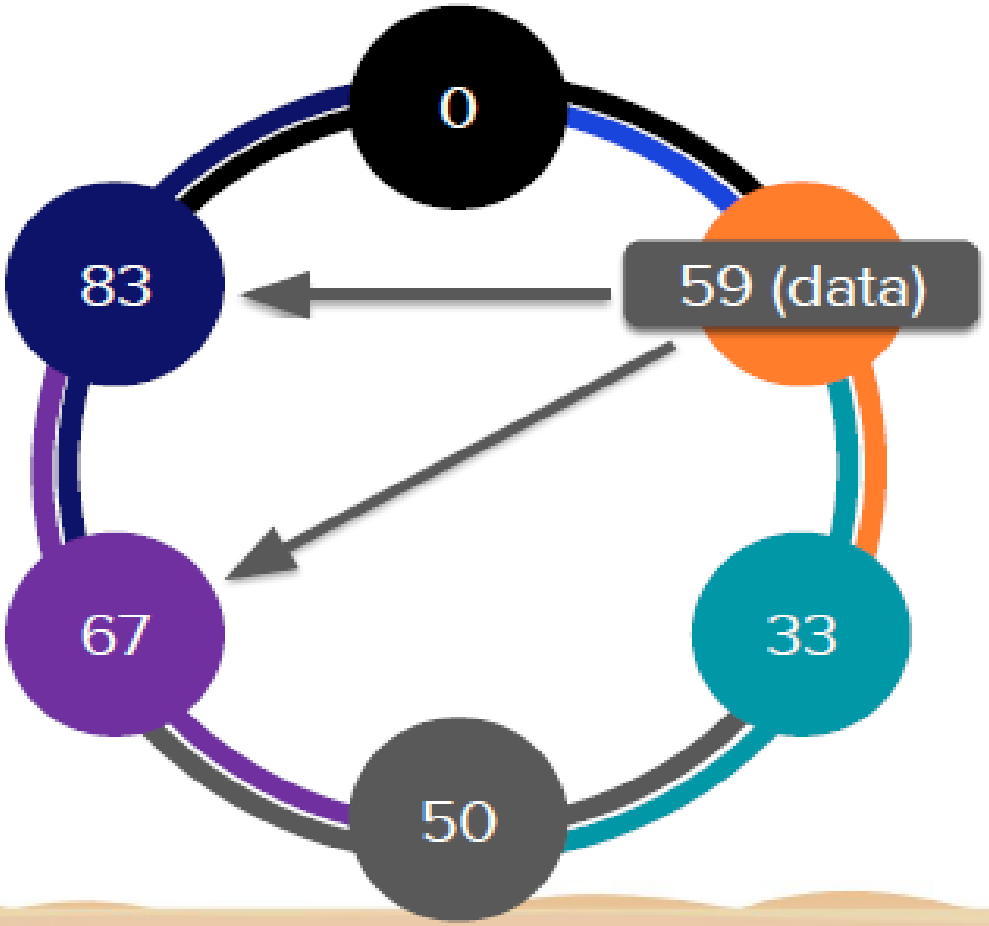
Replication within the Ring

RF = 2



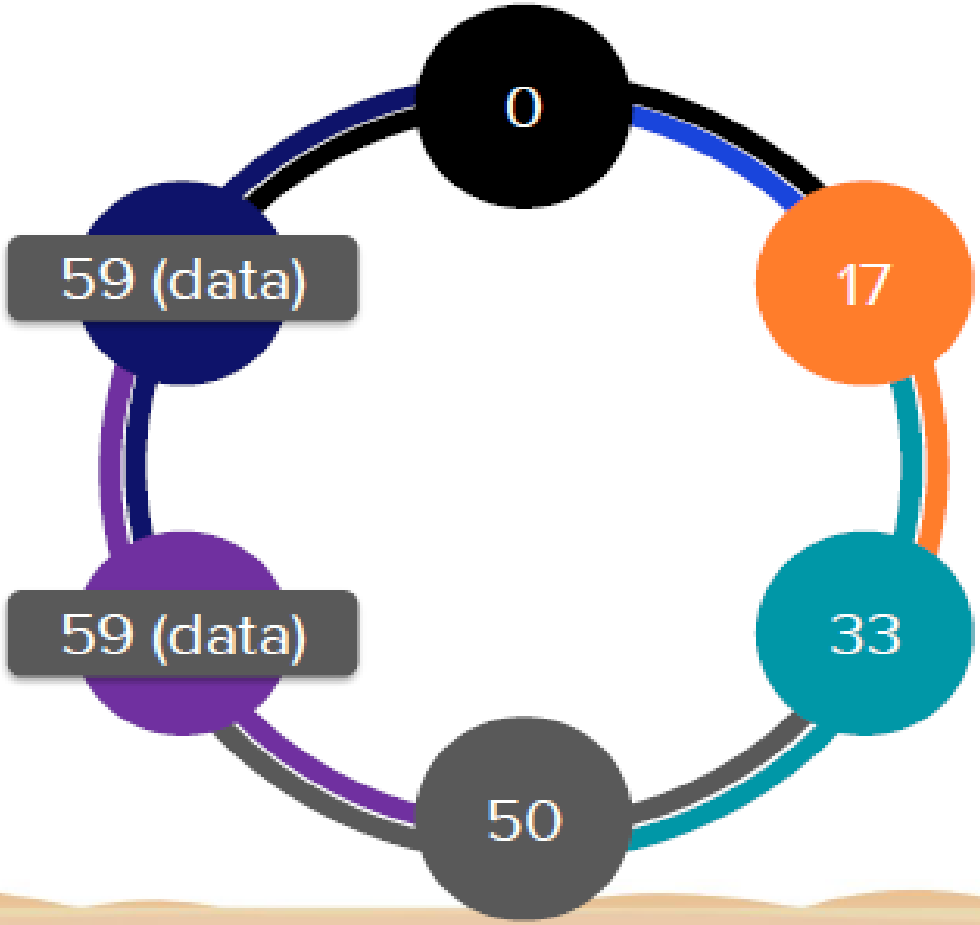
Replication within the Ring

RF = 2



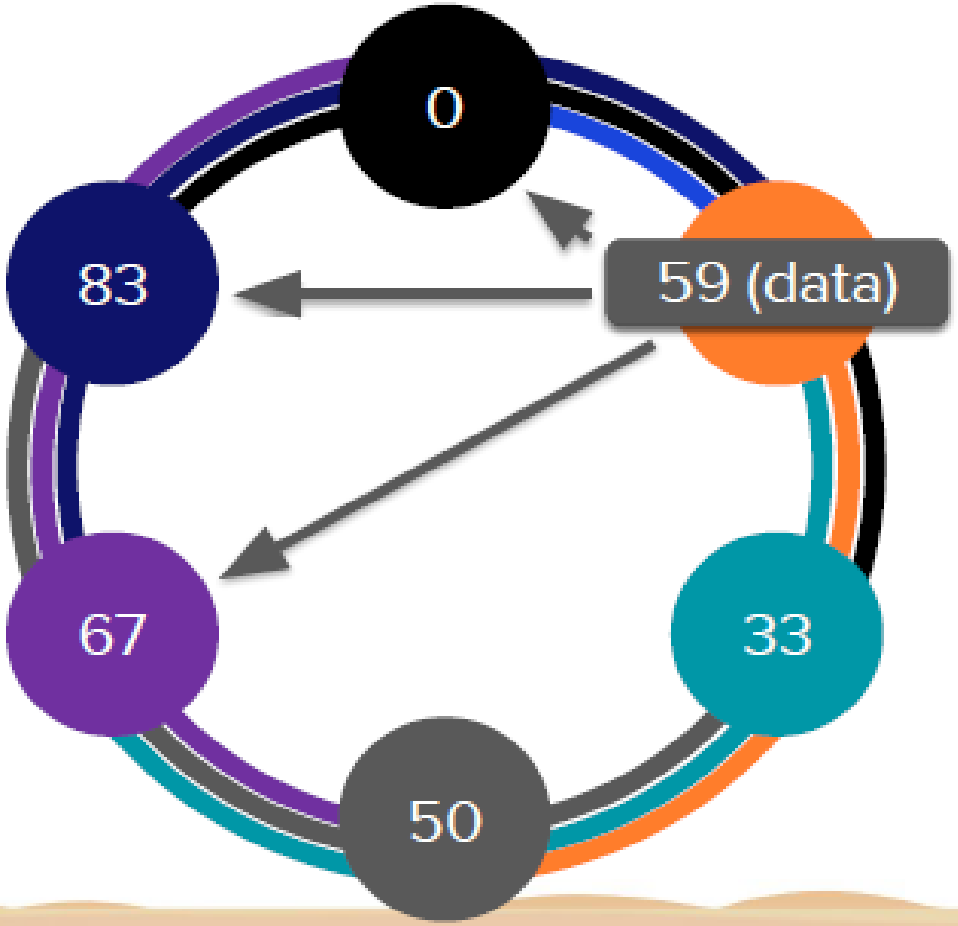
Replication within the Ring

RF = 2



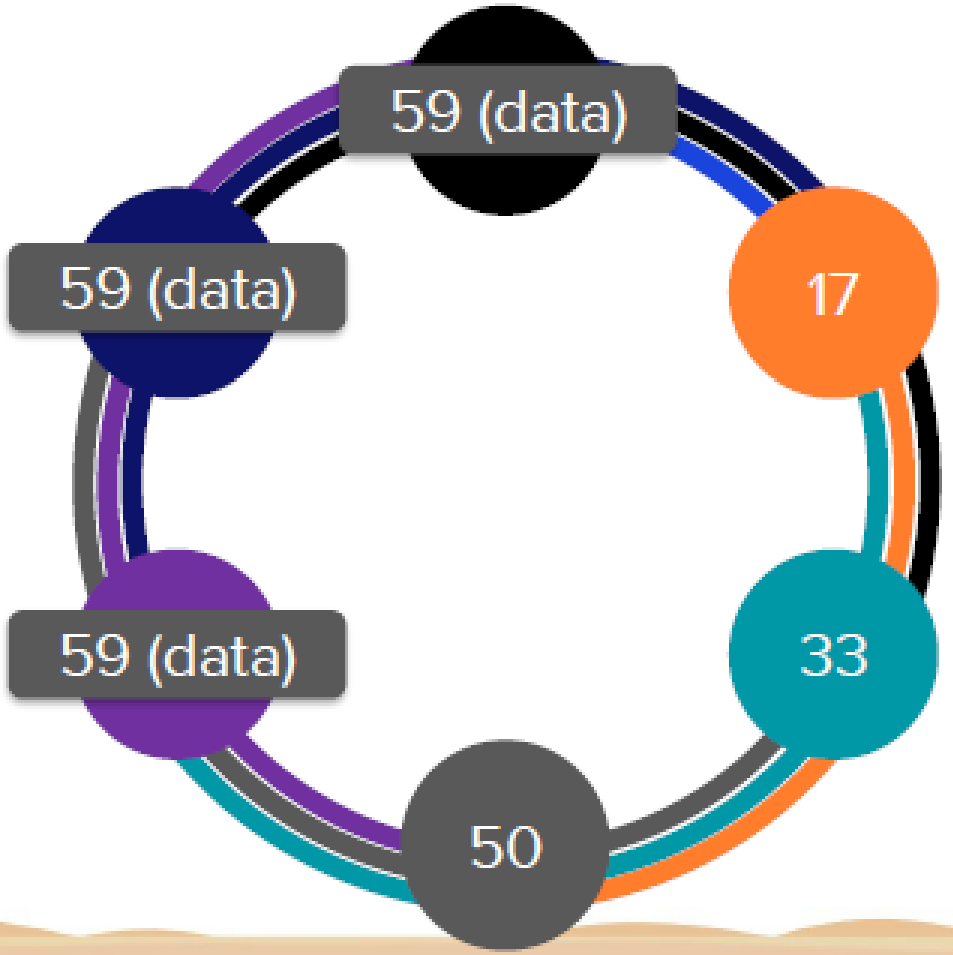
Replication within the Ring

RF = 3



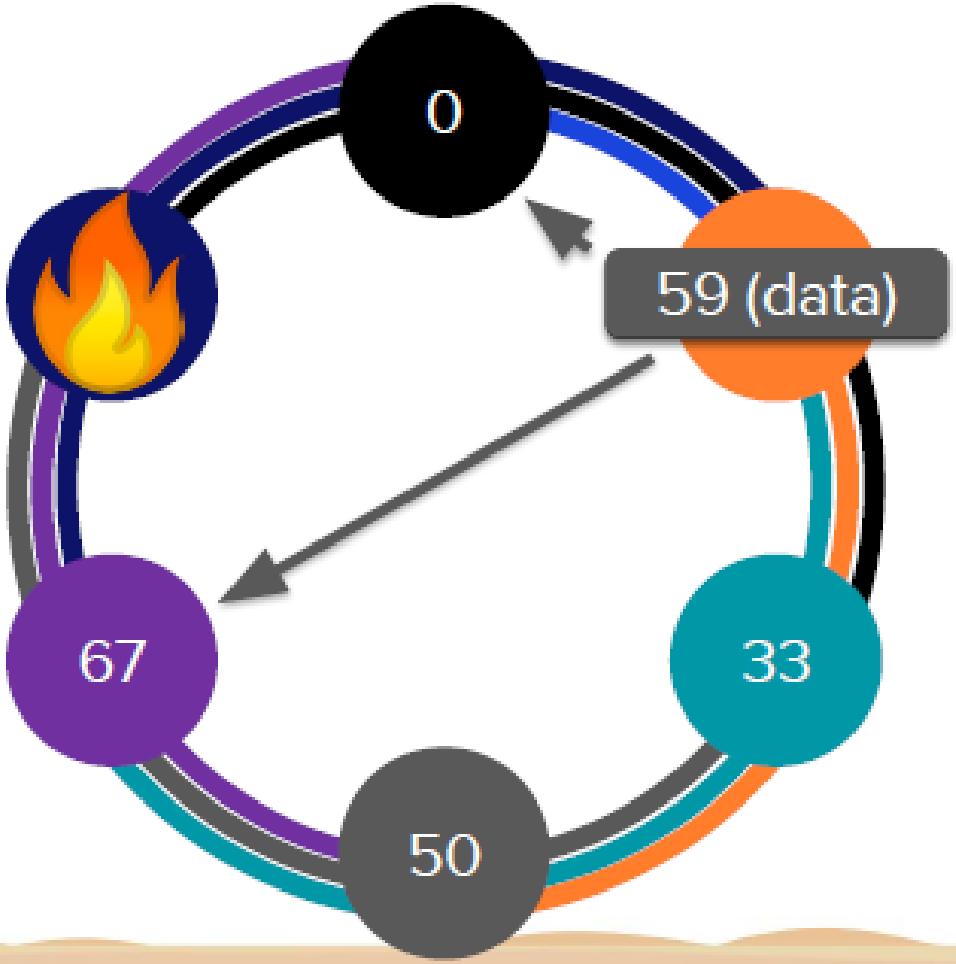
Replication within the Ring

RF = 3



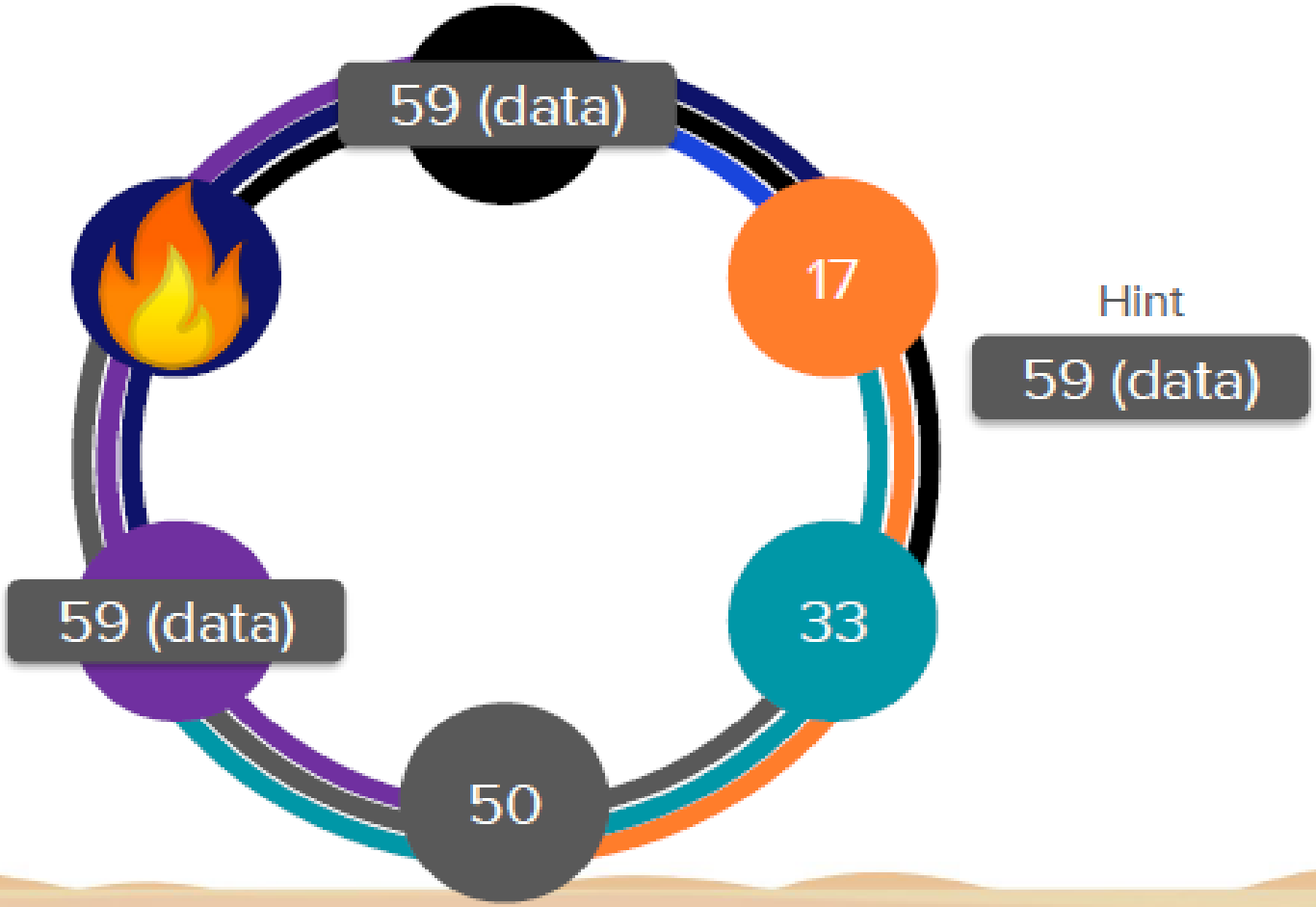
Node Failure

RF = 3



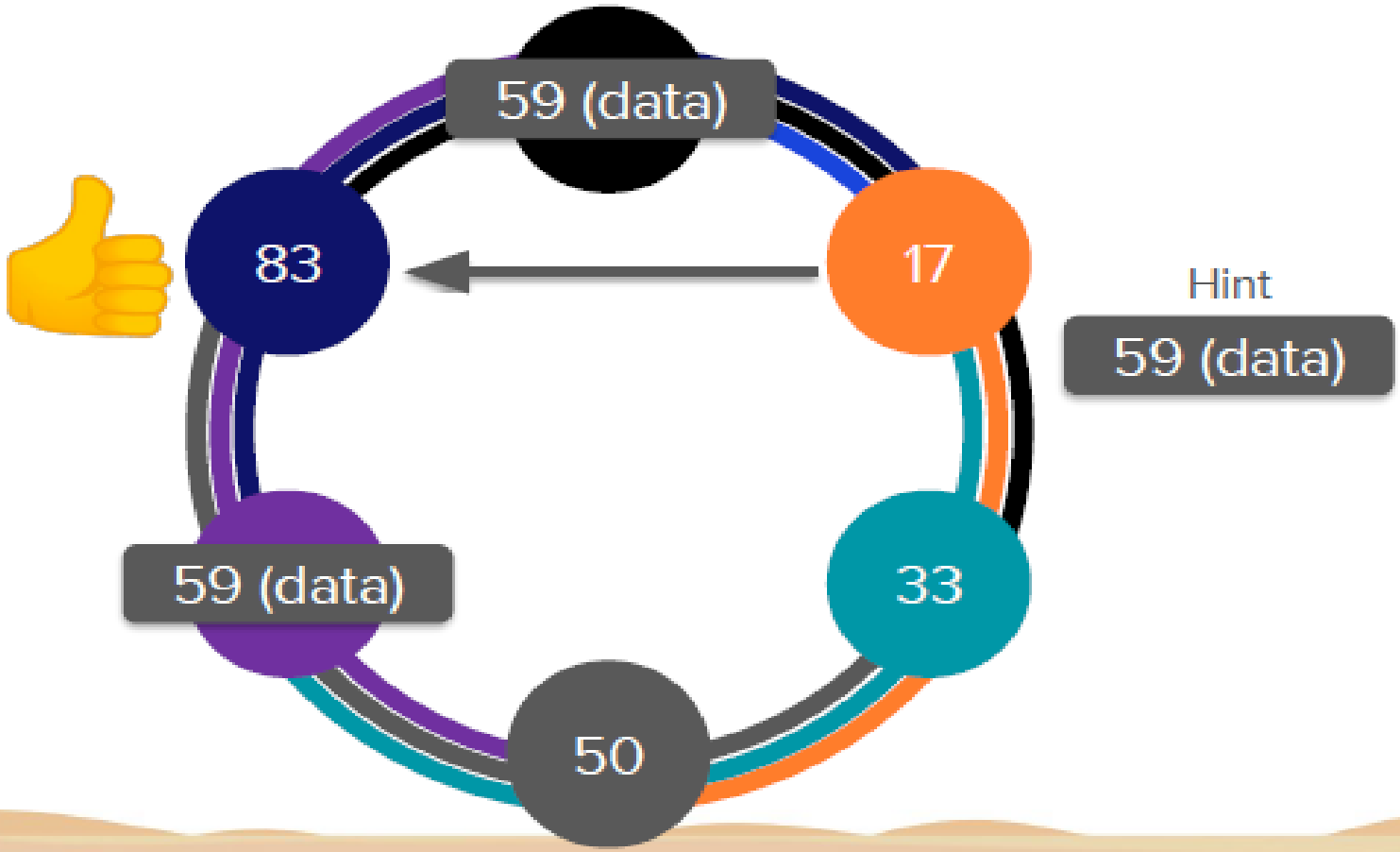
Node Failure

RF = 3



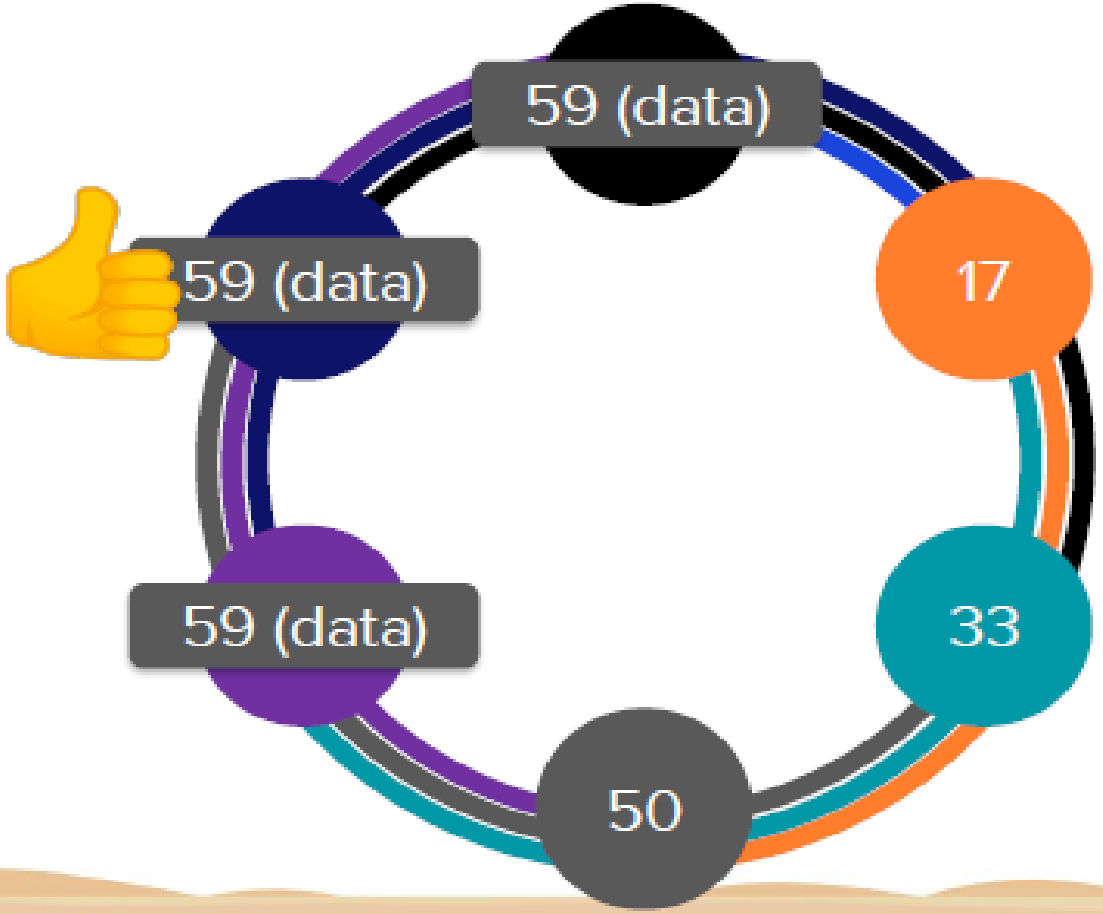
Node Failure

RF = 3

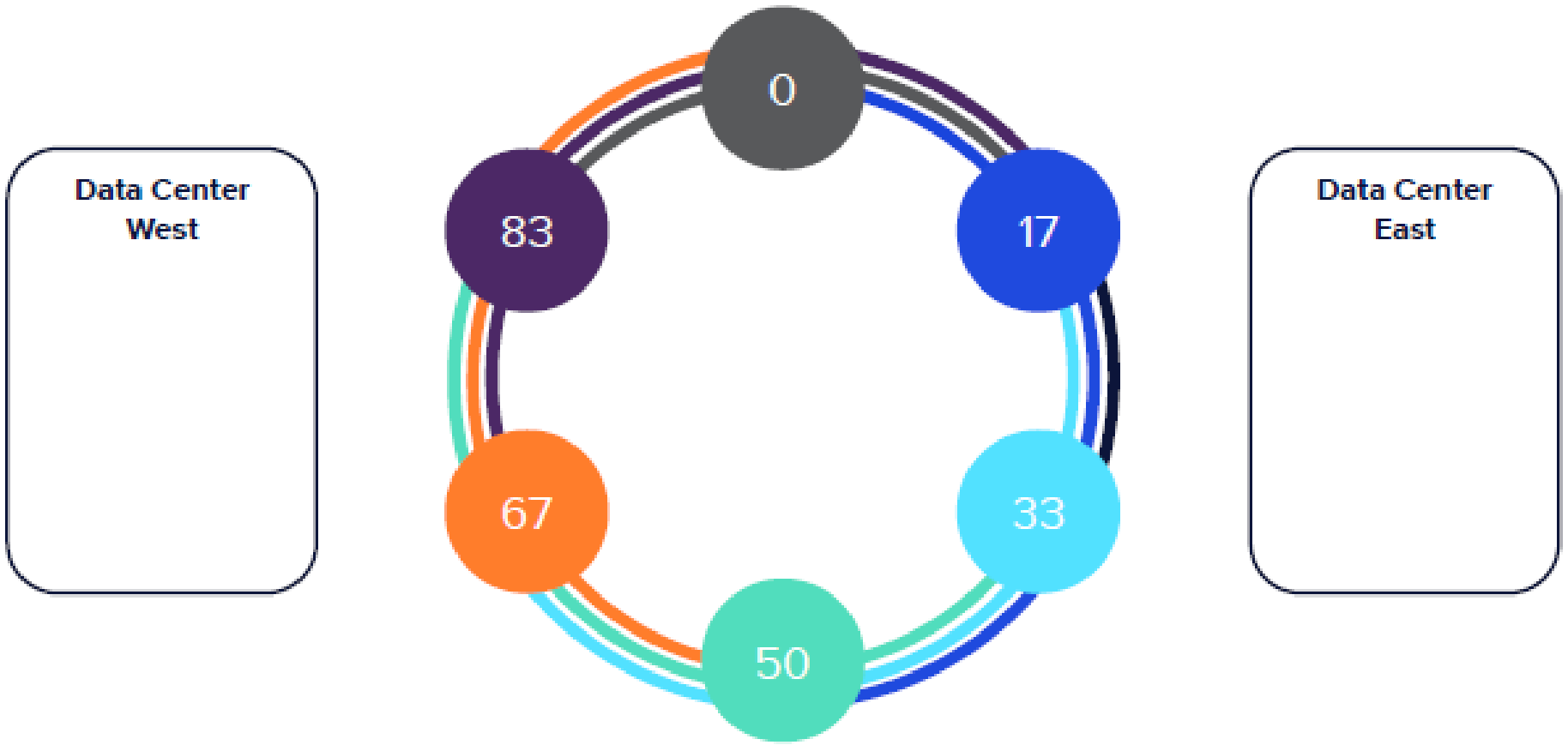


Node Failure – Recovered!

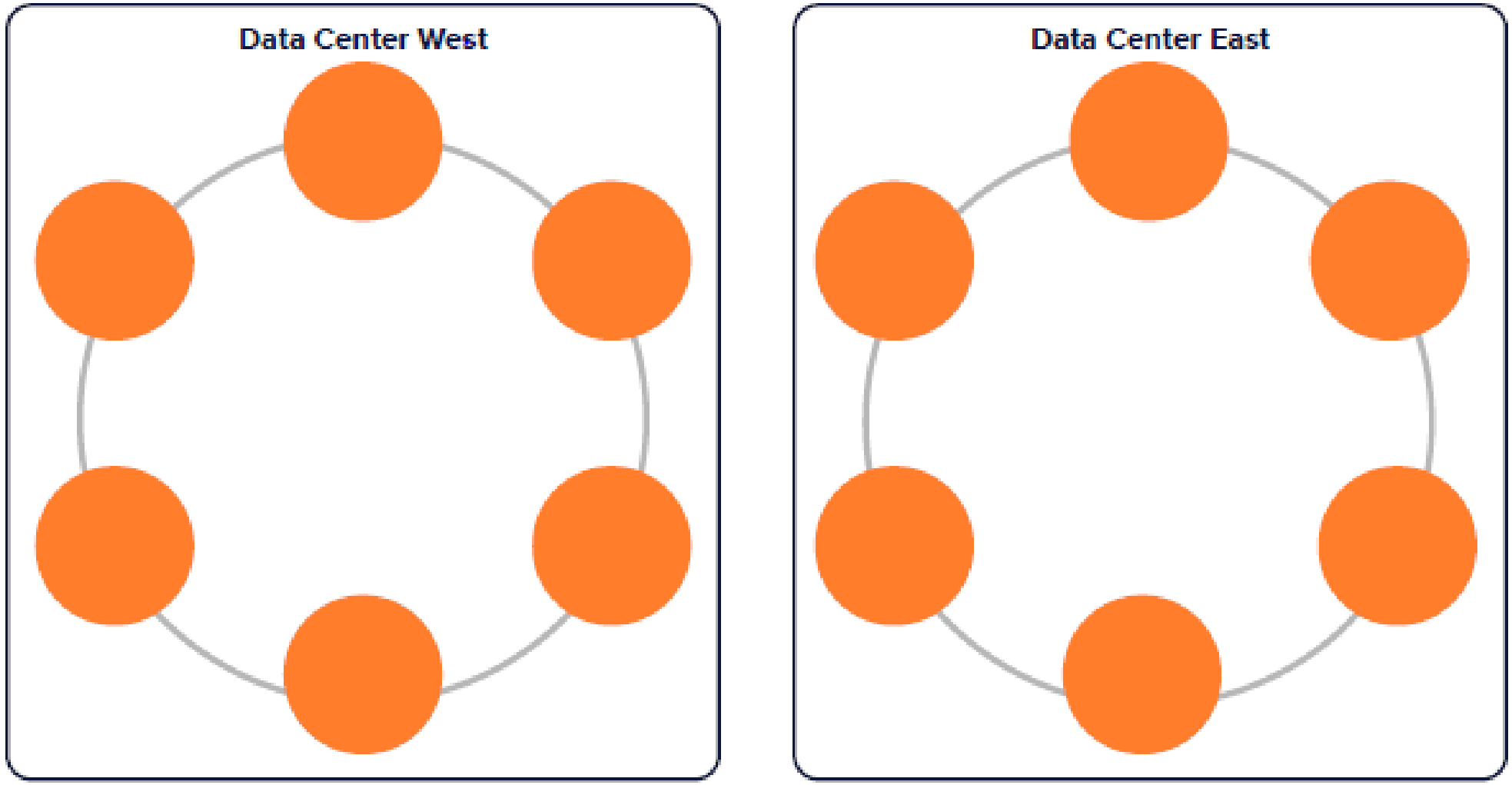
RF = 3



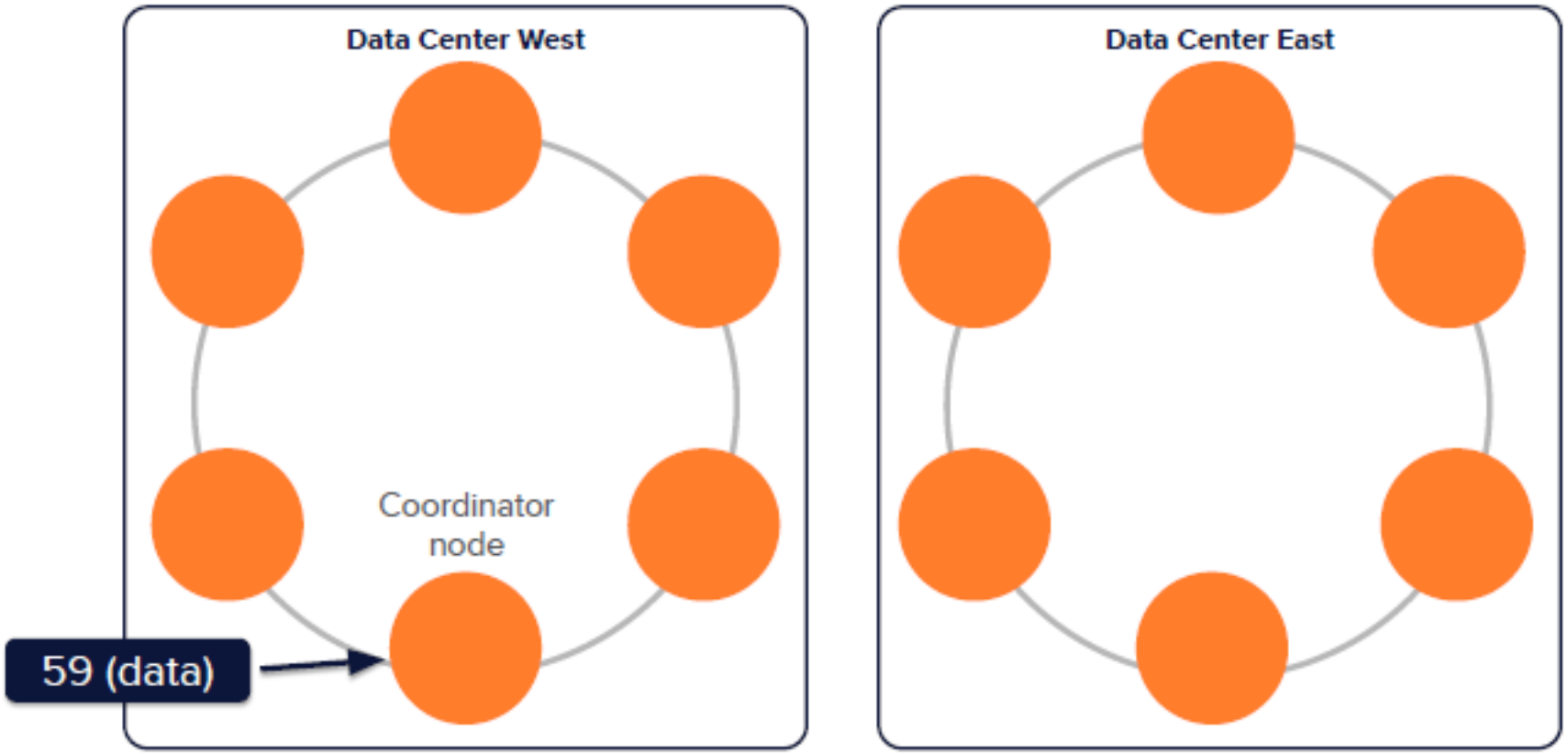
Multi-Data Center Replication



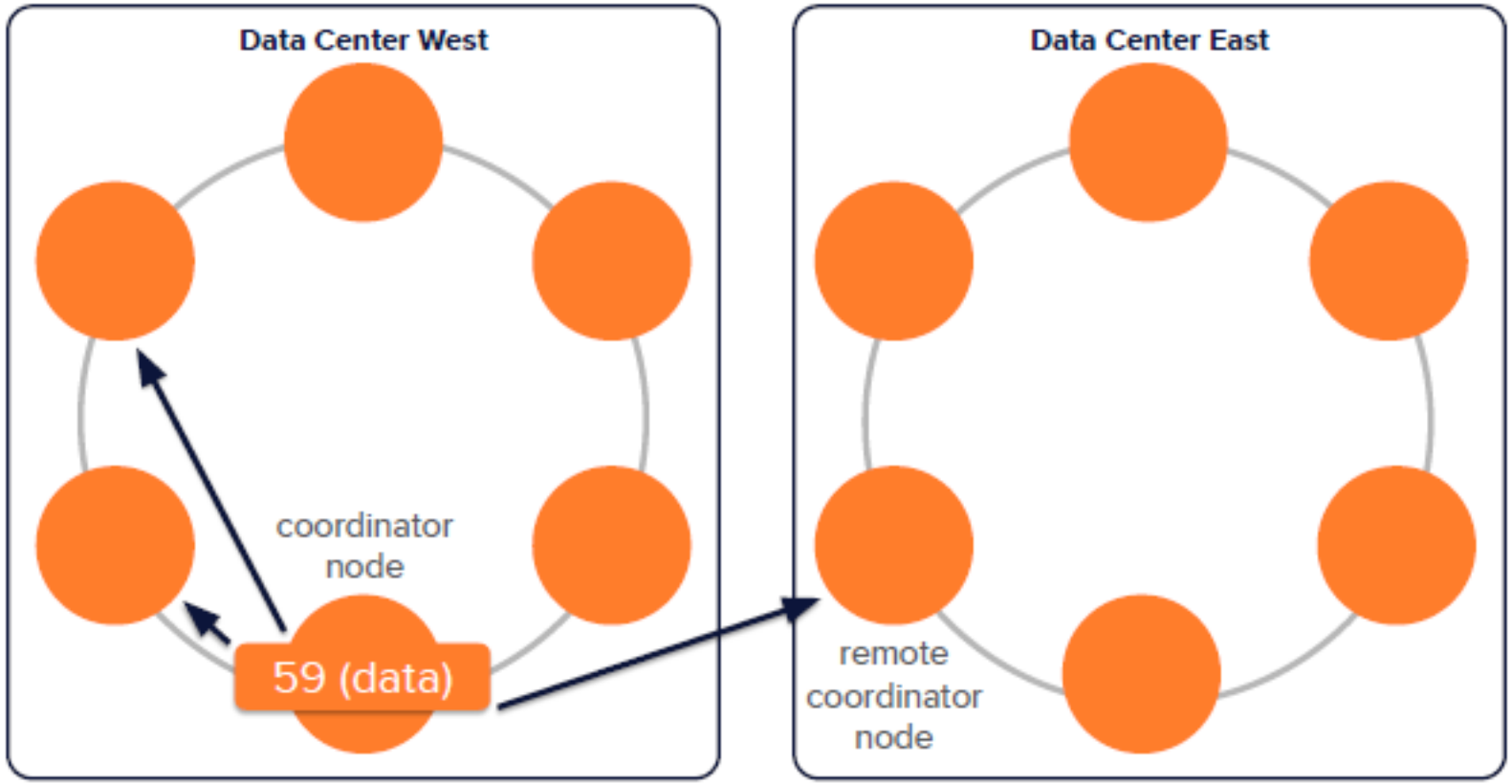
Multi-Data Center Replication (RF=2 in each DC)



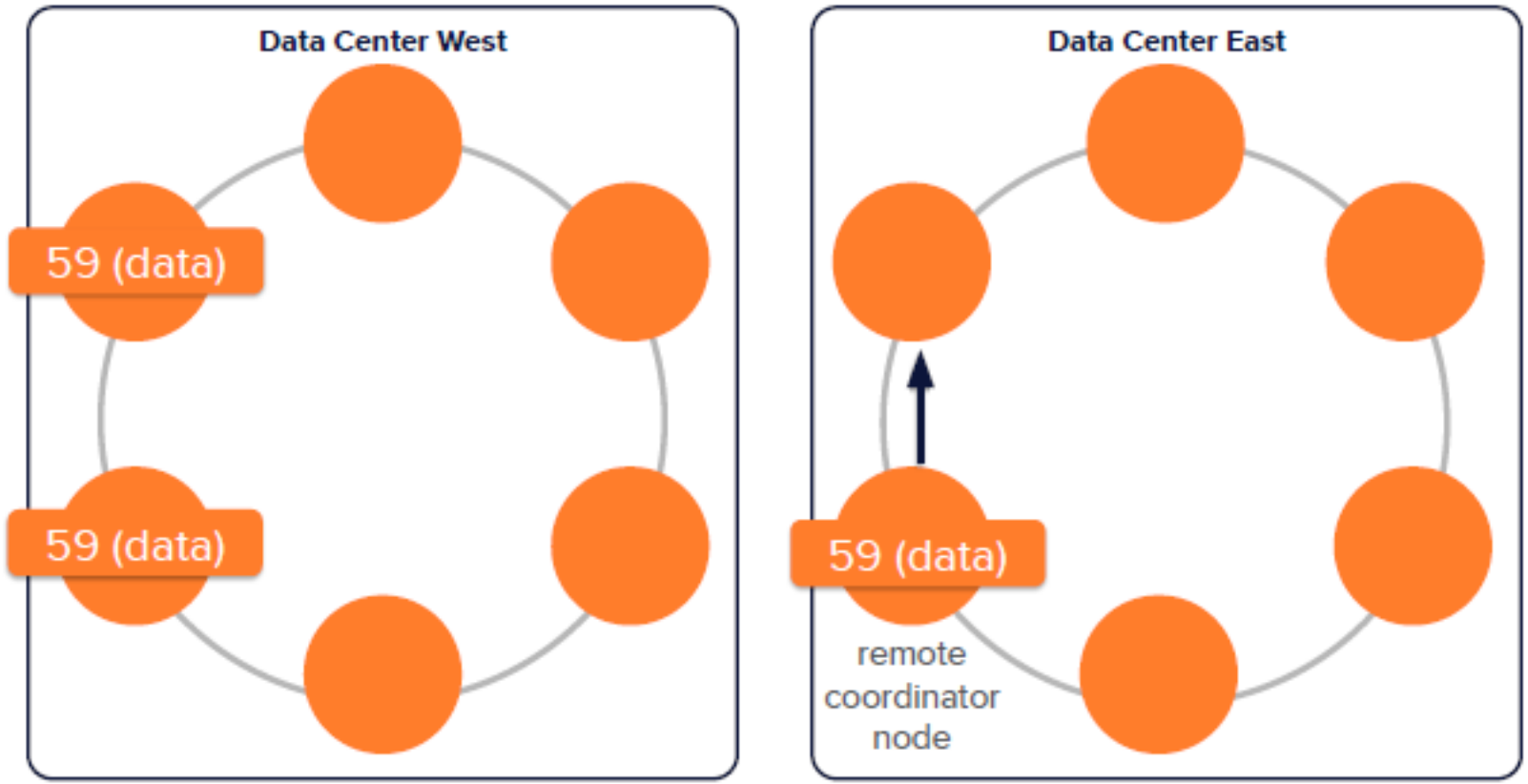
Multi-Data Center Replication (RF=2 in each DC)



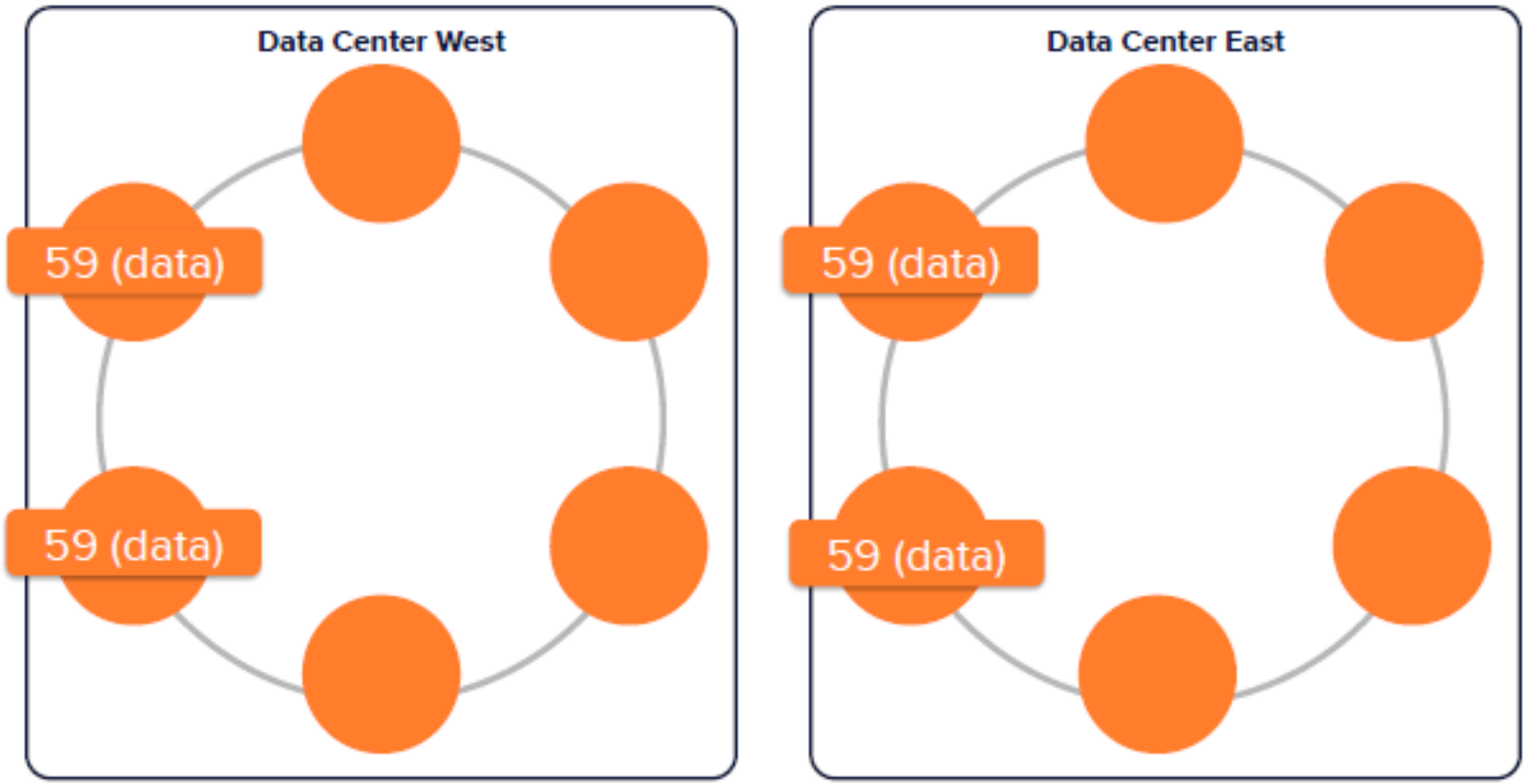
Multi-Data Center Replication (RF=2 in each DC)



Multi-Data Center Replication (RF=2 in each DC)



Multi-Data Center Replication (RF=2 in each DC)

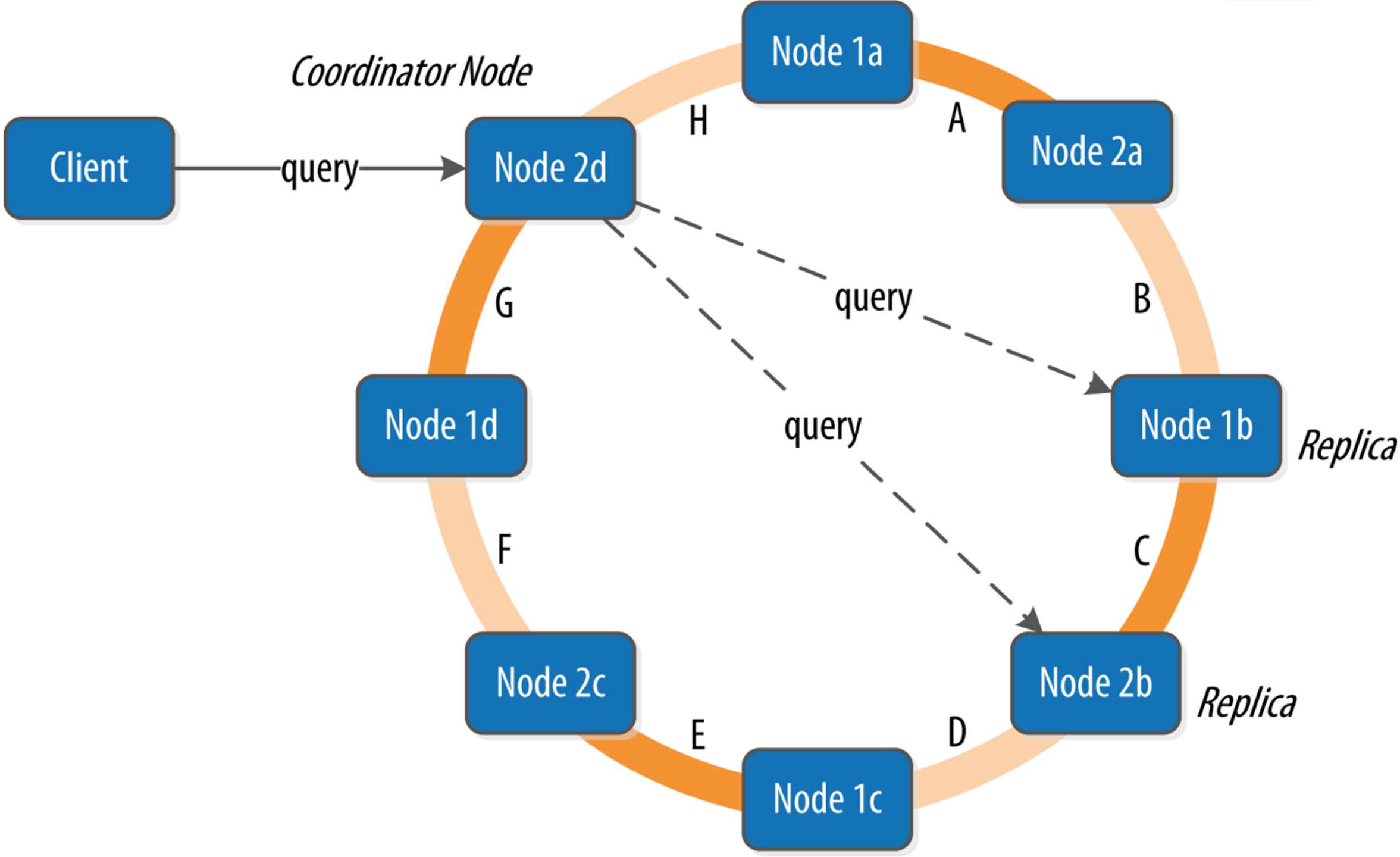


Consistency Levels

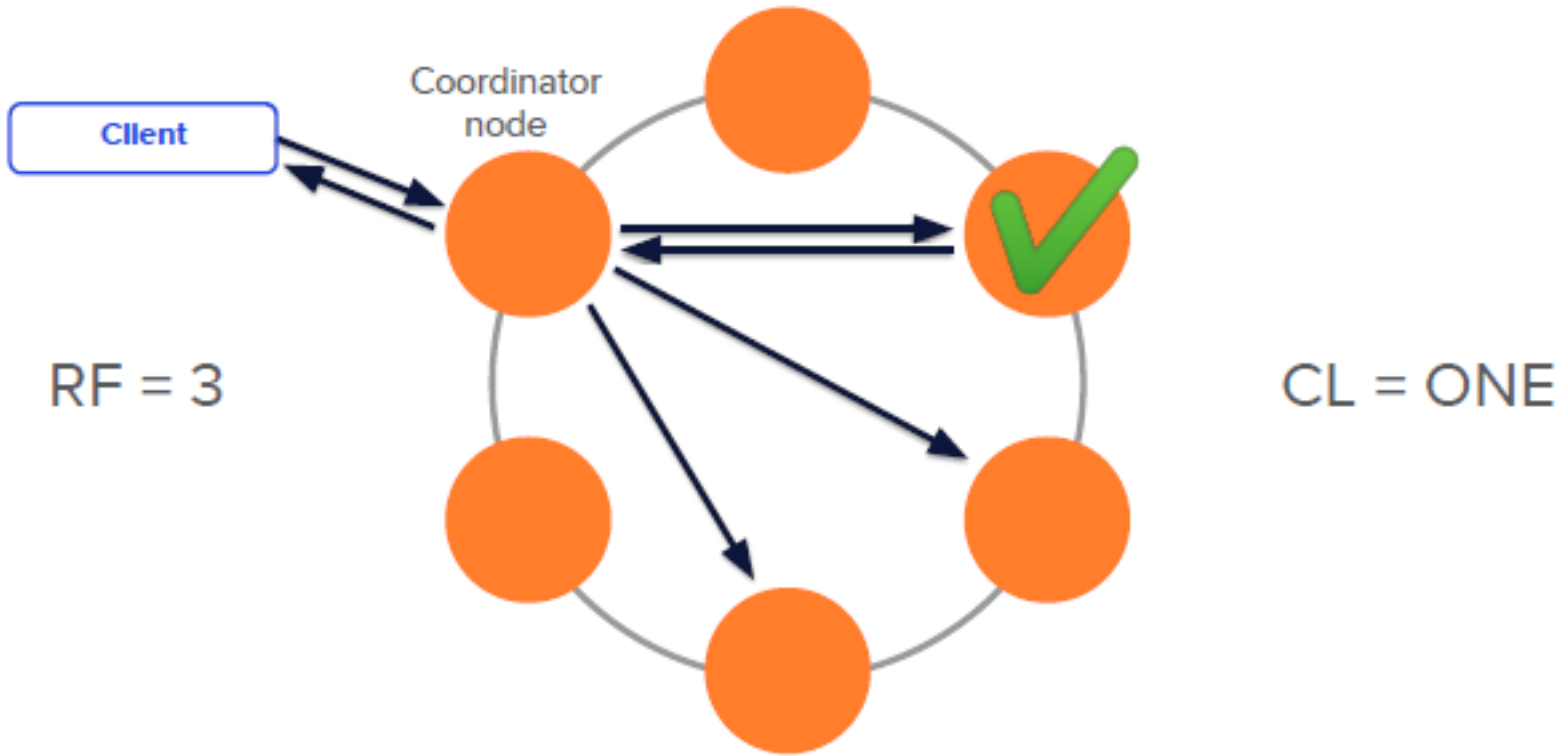
- Cassandra provides tuneable consistency levels that allow you to make these trade-offs at a fine-grained level.
- A consistency level on each read or write query indicates how much consistency you require.
- A higher consistency level means that more nodes need to respond to a read or write query, giving you more assurance that the values present on each replica are the same.
- 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.
- 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. Q = floor RF/2 + 1

Queries and Coordinator Nodes

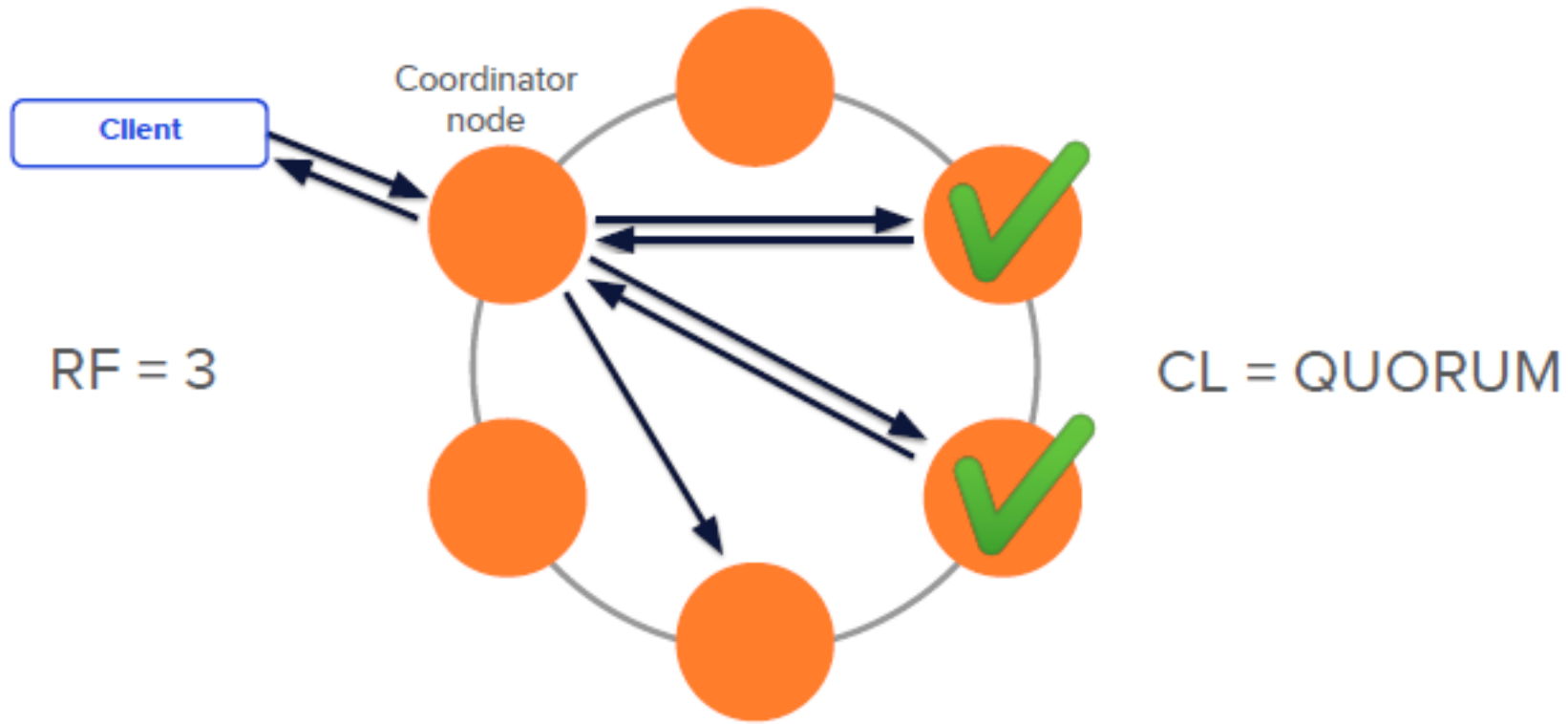
- A client may connect to any node in the cluster to initiate a read or write query.
- This node is known as the coordinator node.
- Coordinator identifies which nodes are replicas for the data that is being written or read and forwards the queries to them.
- For a write, the coordinator node contacts 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 consistencylevel is met, and returns the data to the client.


Coordinator Node
Client
query
Node 2d
Node 1a
Node 2a
Node 1b
Replica
Node 2b
Replica
Node 1c
Node 2c
Node 1d
query
query
A
B
C
D
E
F
G
H

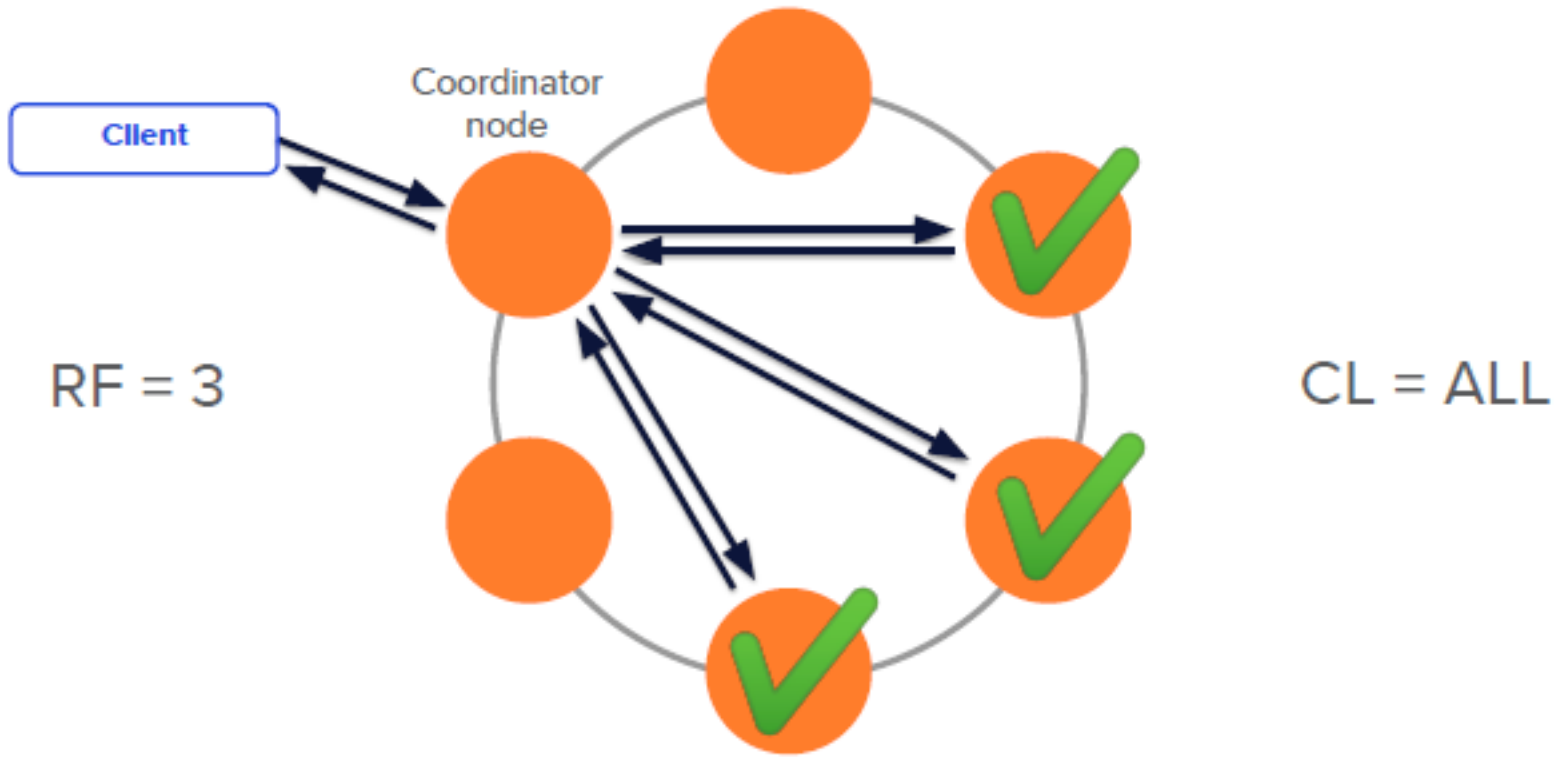
Consistency Levels



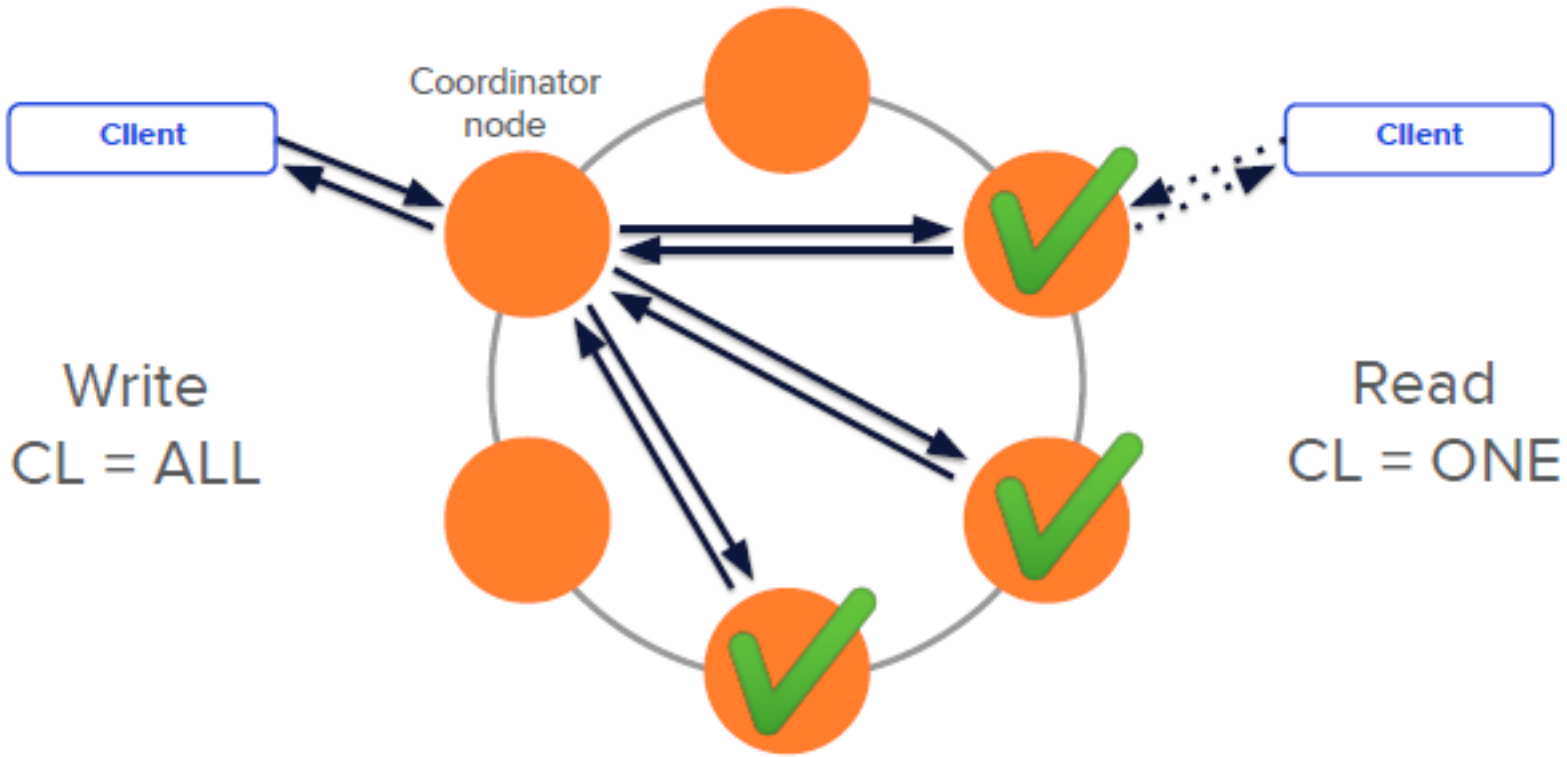
Consistency Levels



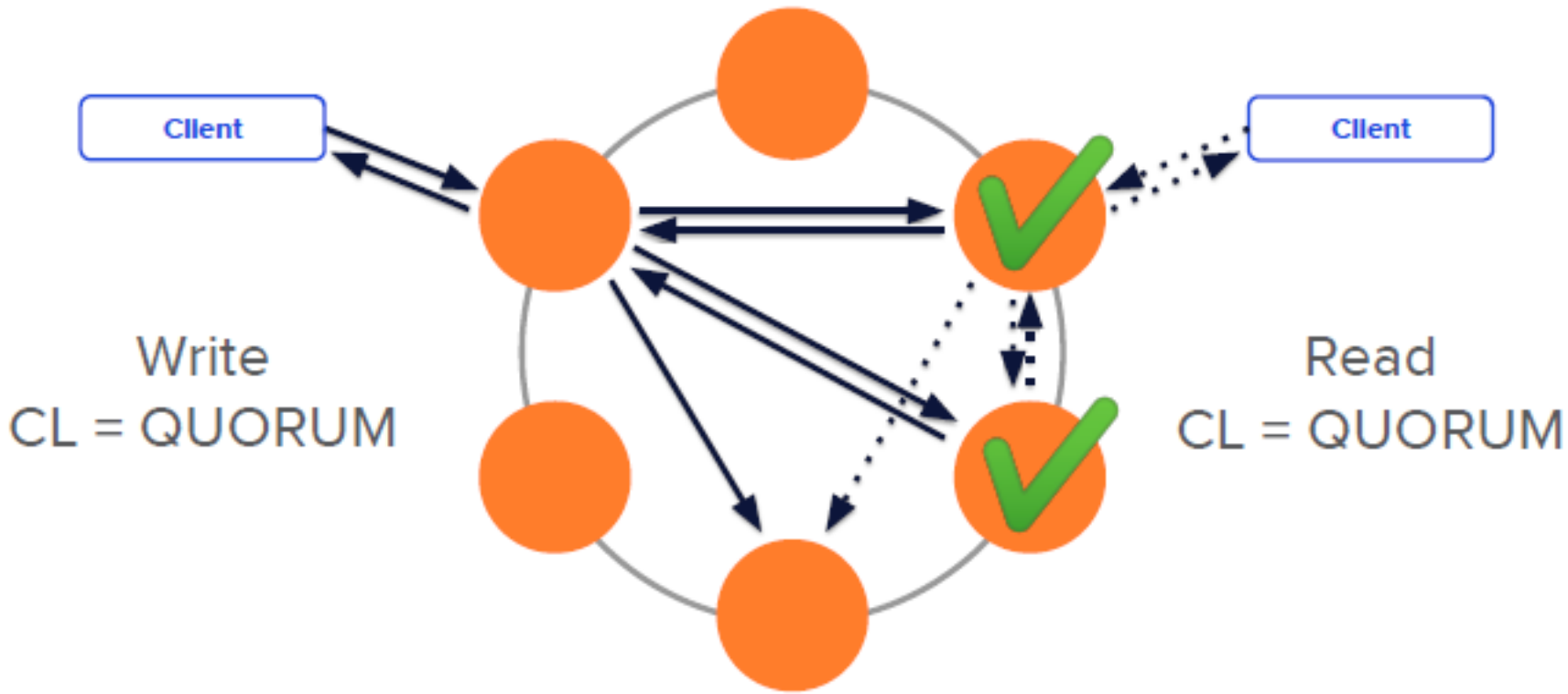
Consistency Levels



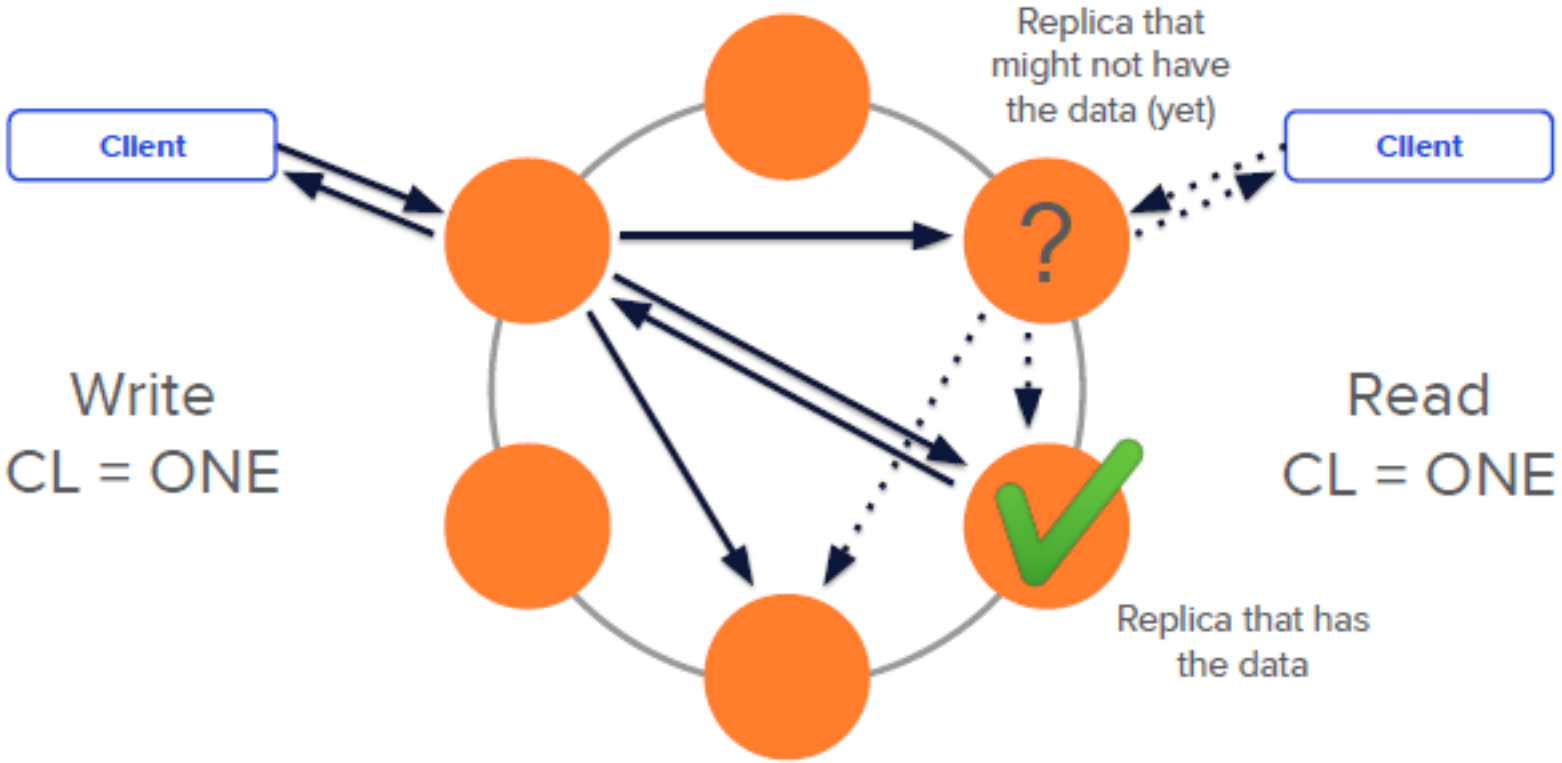
Strong Consistency – One Way



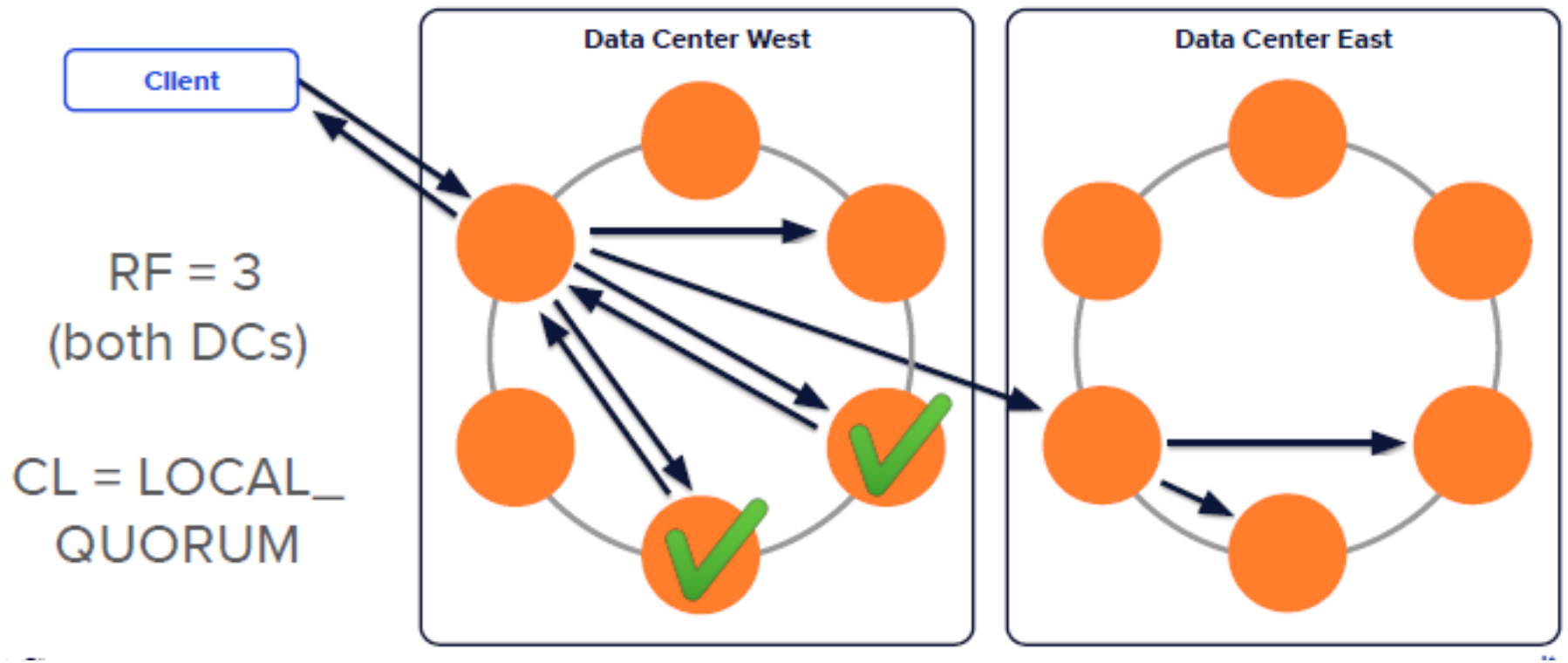
Strong Consistency – A Better Way



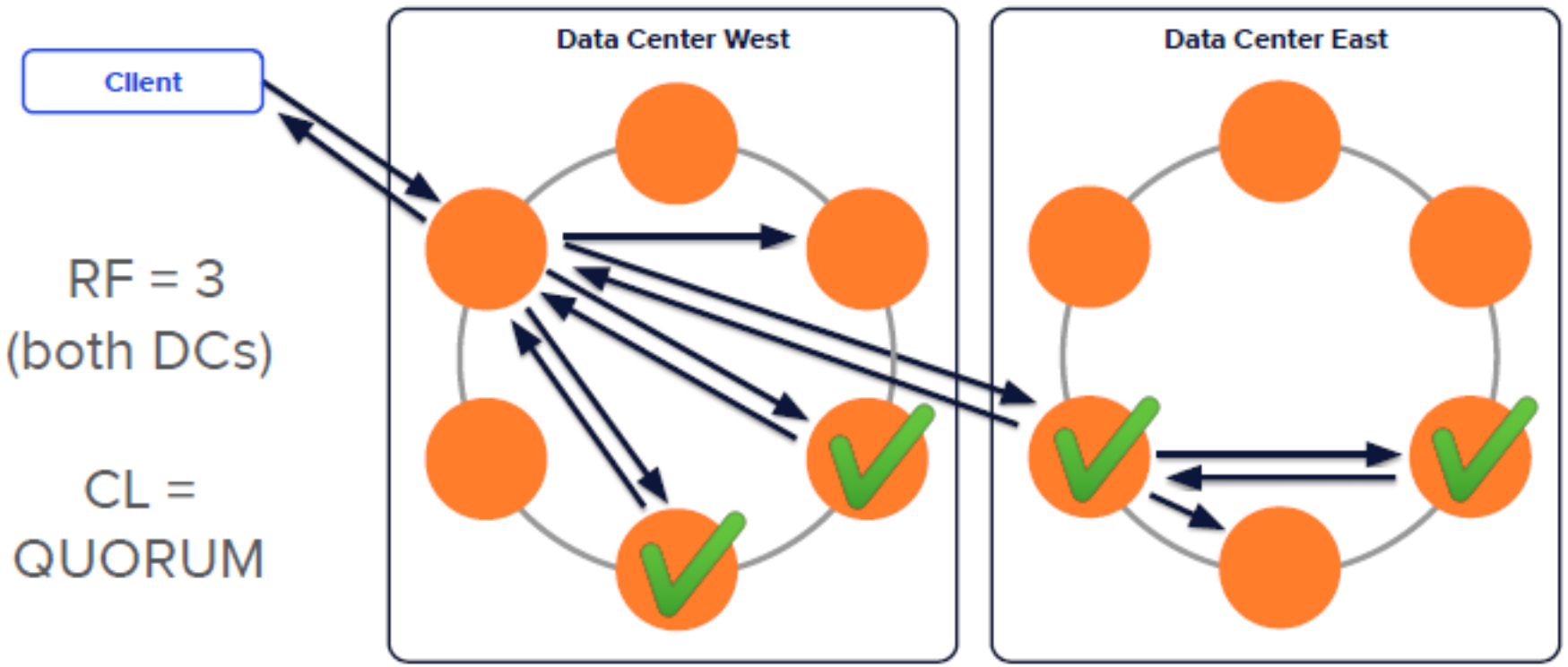
Weak(er) Consistency



Consistency Across Data Centers



Consistency Across Data Centers



Consistency Settings

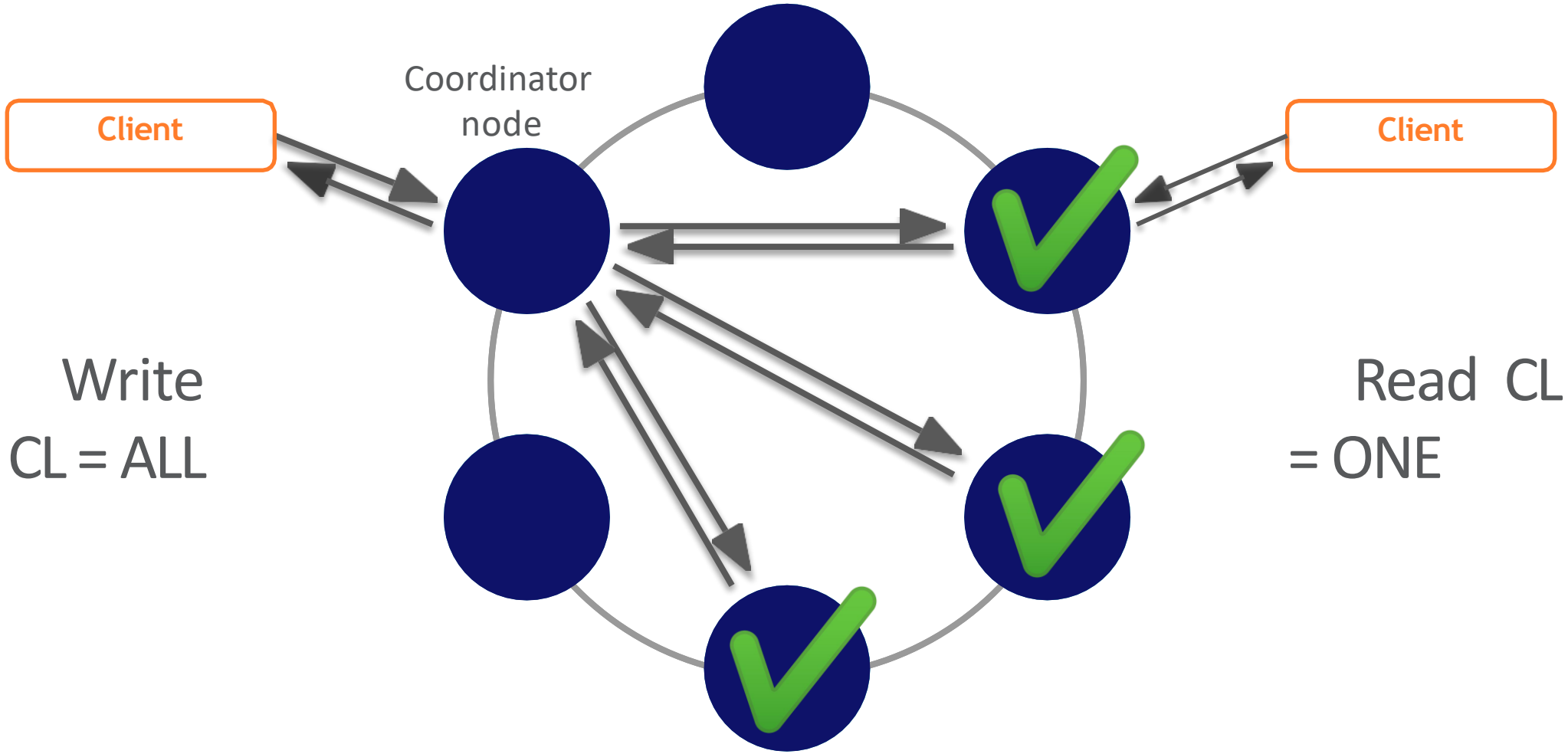
- Weakest to strongest

Setting	Description
ANY	Storing a hint at minimum is satisfactory
ONE, TWO, THREE	Checks closest node(s) to coordinator
QUORUM	Majority vote, (sum_of_replication_factors / 2) + 1
LOCAL_ONE	Closest node to coordinator in same data center
LOCAL_QUORUM	Closest quorum of nodes in same data center
EACH_QUORUM	Quorum of nodes in each data center, applies to writes only
ALL	Every node must participate

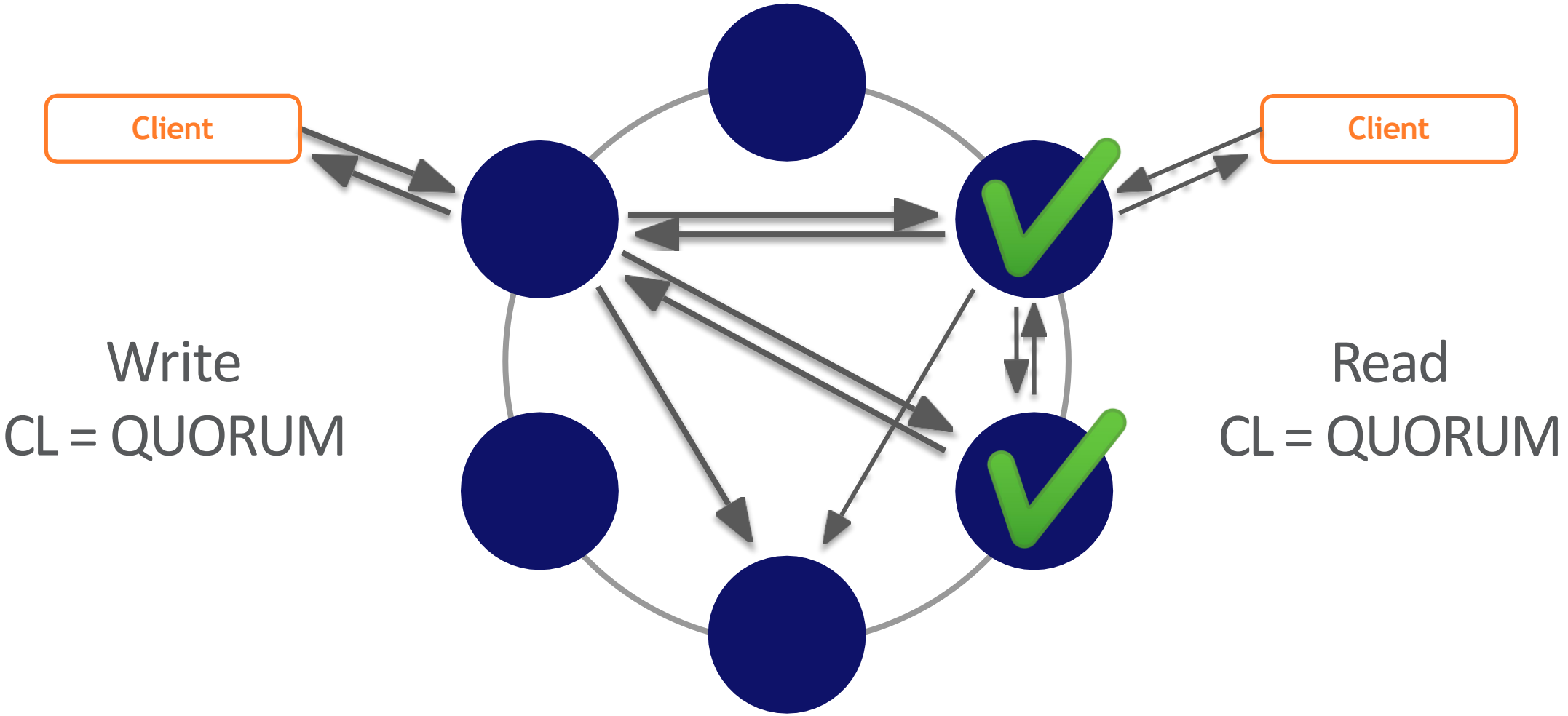
Consistency Tradeoffs

- The higher the consistency, the less chance you may get stale data
  - Pay for this with latency
  - Depends on your situational needs
- IoT and Sensor systems with write-heavy workloads may benefit from CL=ONE for writes if data isn't required immediately

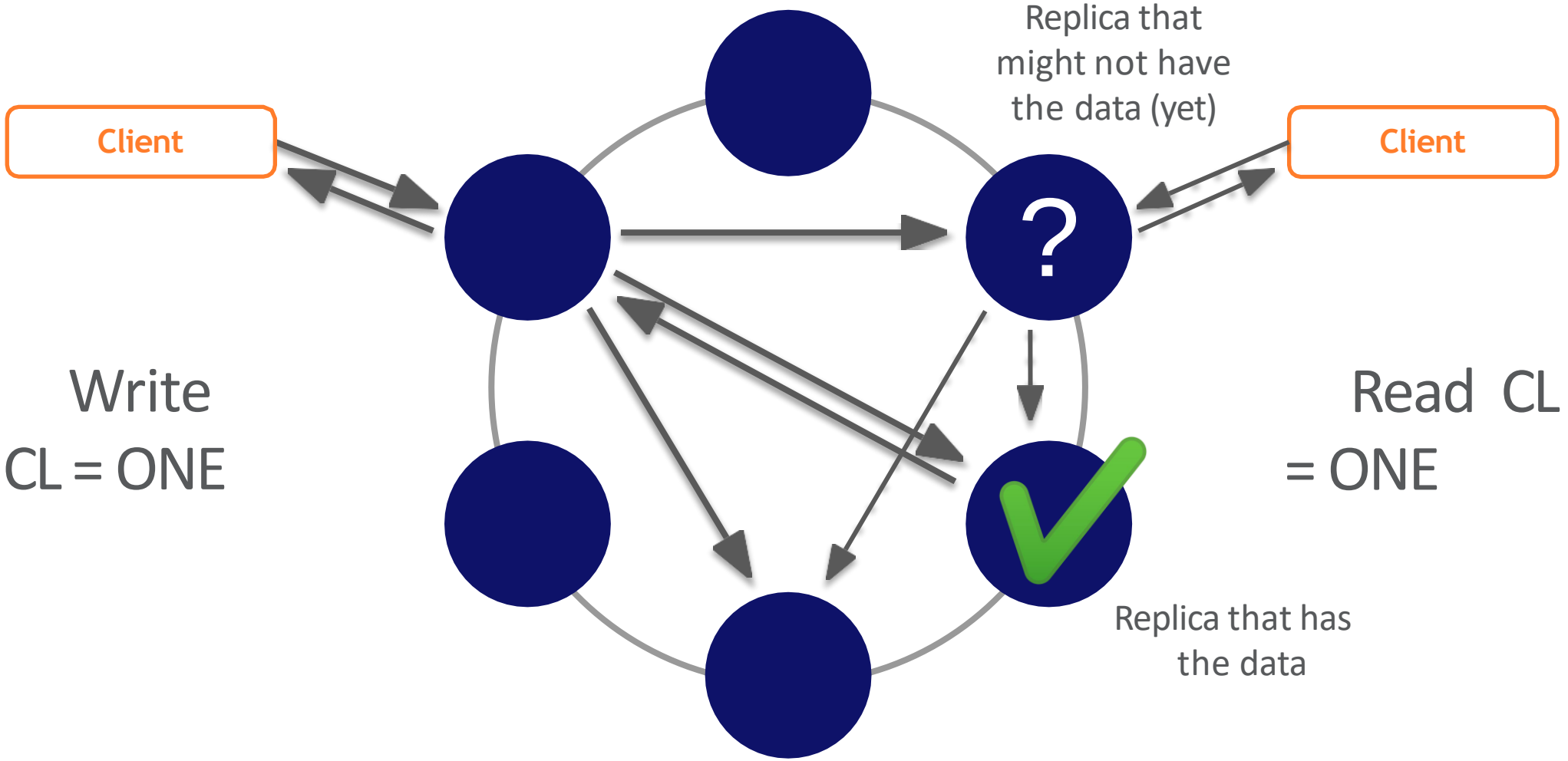
Immediate Consistency – One Way



Immediate Consistency – A Better Way



Weak(er) Consistency



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.
- To ensure general availability of the ring in such a situation, Cassandra implements a feature called hinted handoff.
- Might think of a hint as a little Post-it Note that contains the information from the write request.
- 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.
- This allows Cassandra to be always available for writes, and generally enables a cluster to sustain the same write load even when some of the nodes are down.
- It also reduces the time that a failed node will be inconsistent after it does come back online.

Hinted Handoff

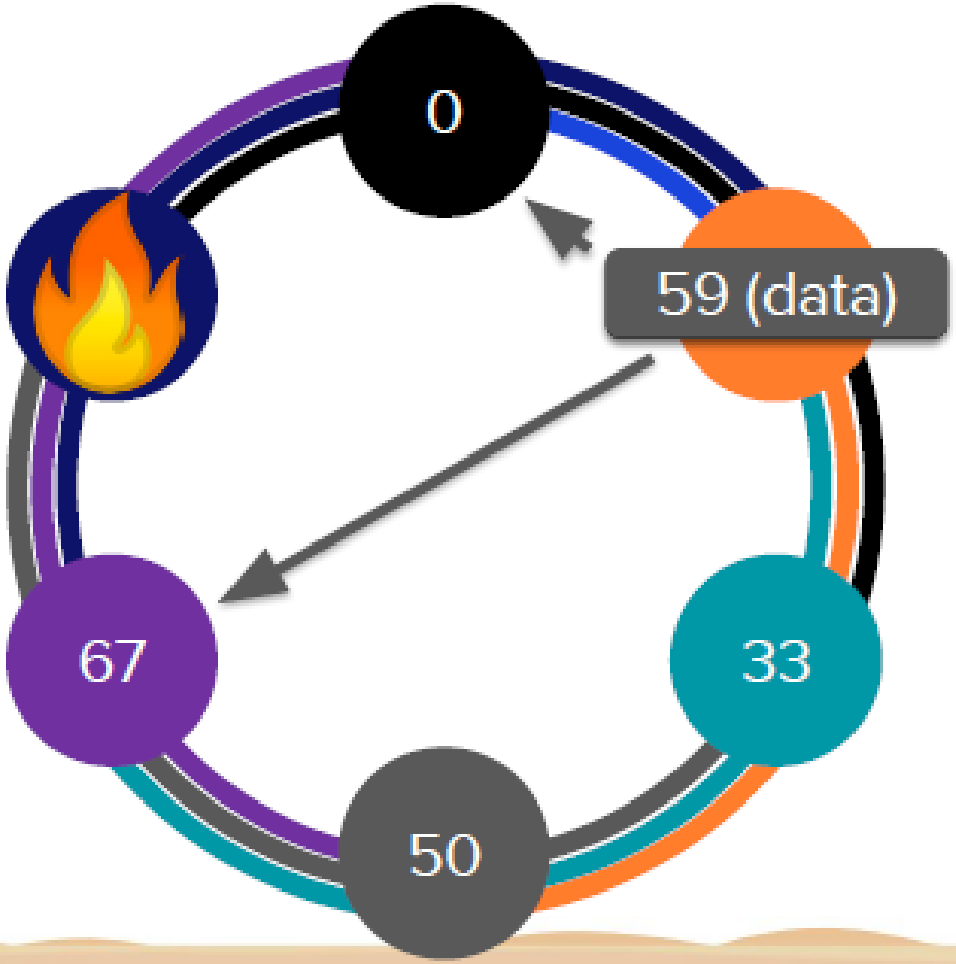
- In general, hints do not count as writes for the purposes of consistency level.
- exception is the consistency level ANY, which was added in 0.6.
- ANY means that a hinted handoff alone will count as sufficient toward the success of a write operation.
- That is, even if only a hint was able to be recorded, the write still counts as successful.
- Note that the write is considered durable, but the data may not be readable until the hint is delivered to the target replica.

Problem with hinted handoffs

- If a node is offline for some time, the hints can build up considerably on other nodes.
- Then, when the other nodes notice that the failed node has come back online, they tend to flood that node with requests, just at the moment it is most vulnerable (when it is struggling to come back into play after a failure).
- To address this problem, Cassandra limits the storage of hints to a configurable time window.
- Also possible to disable hinted handoff entirely.

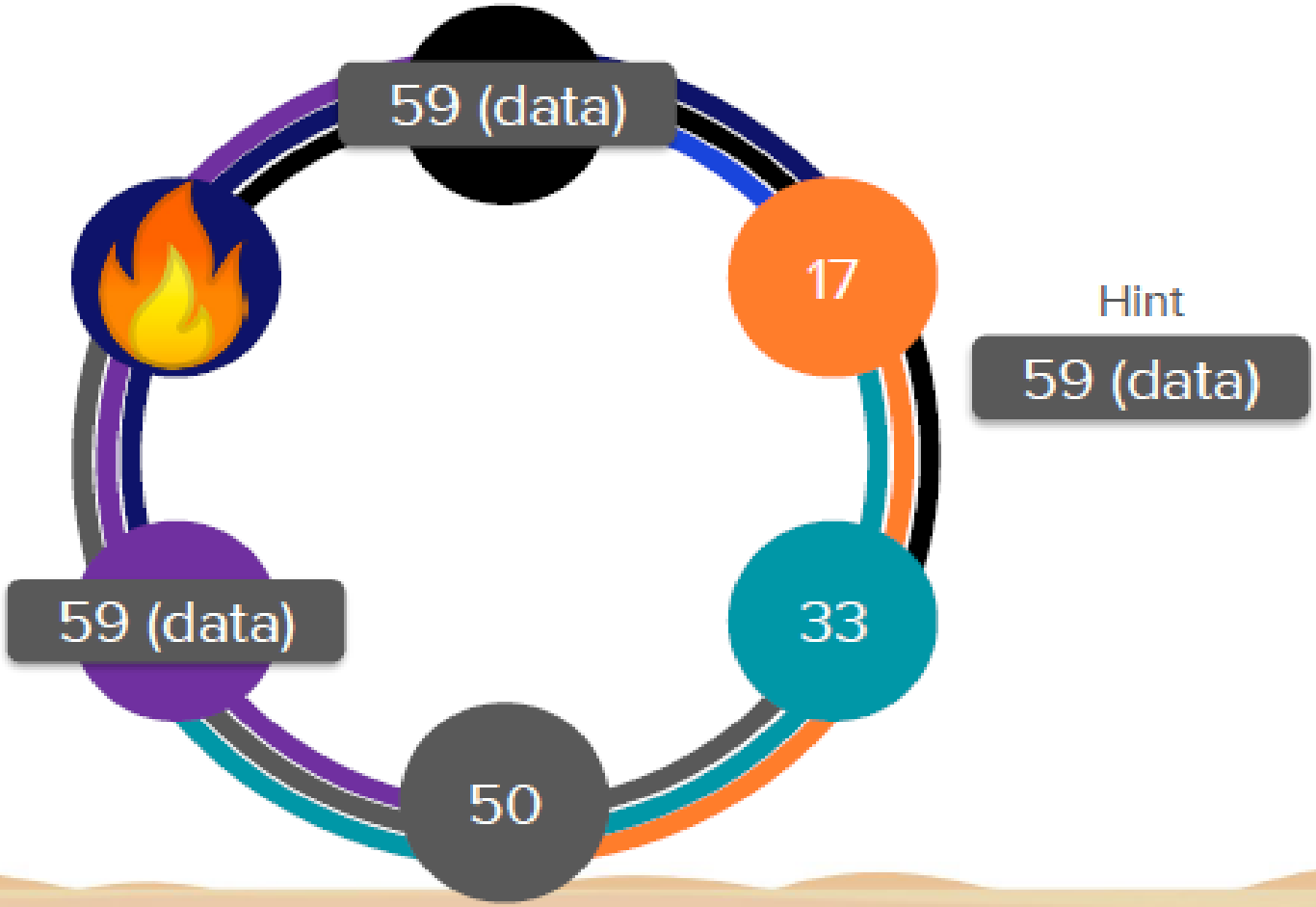
Node Failure

RF = 3



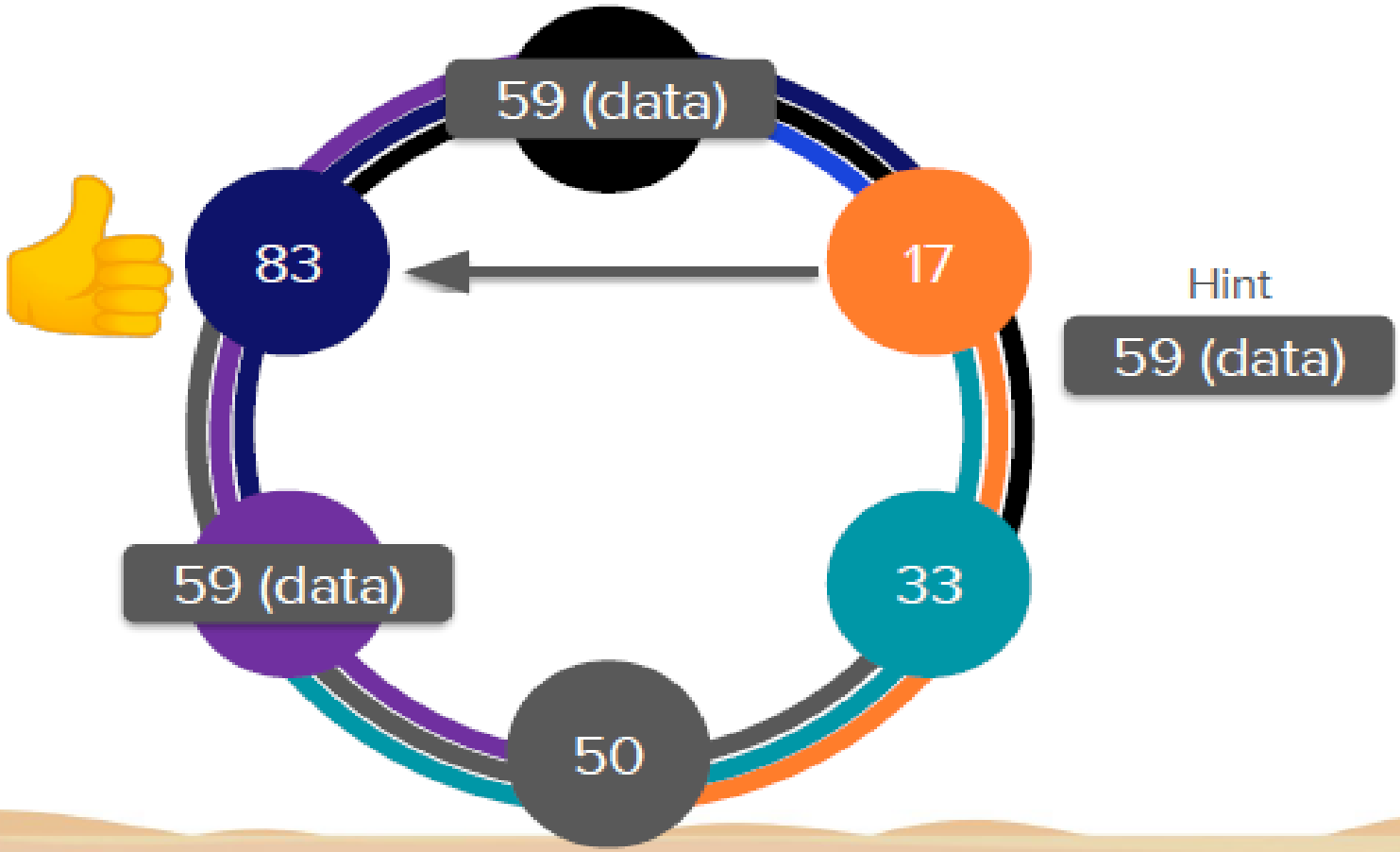
Node Failure

RF = 3



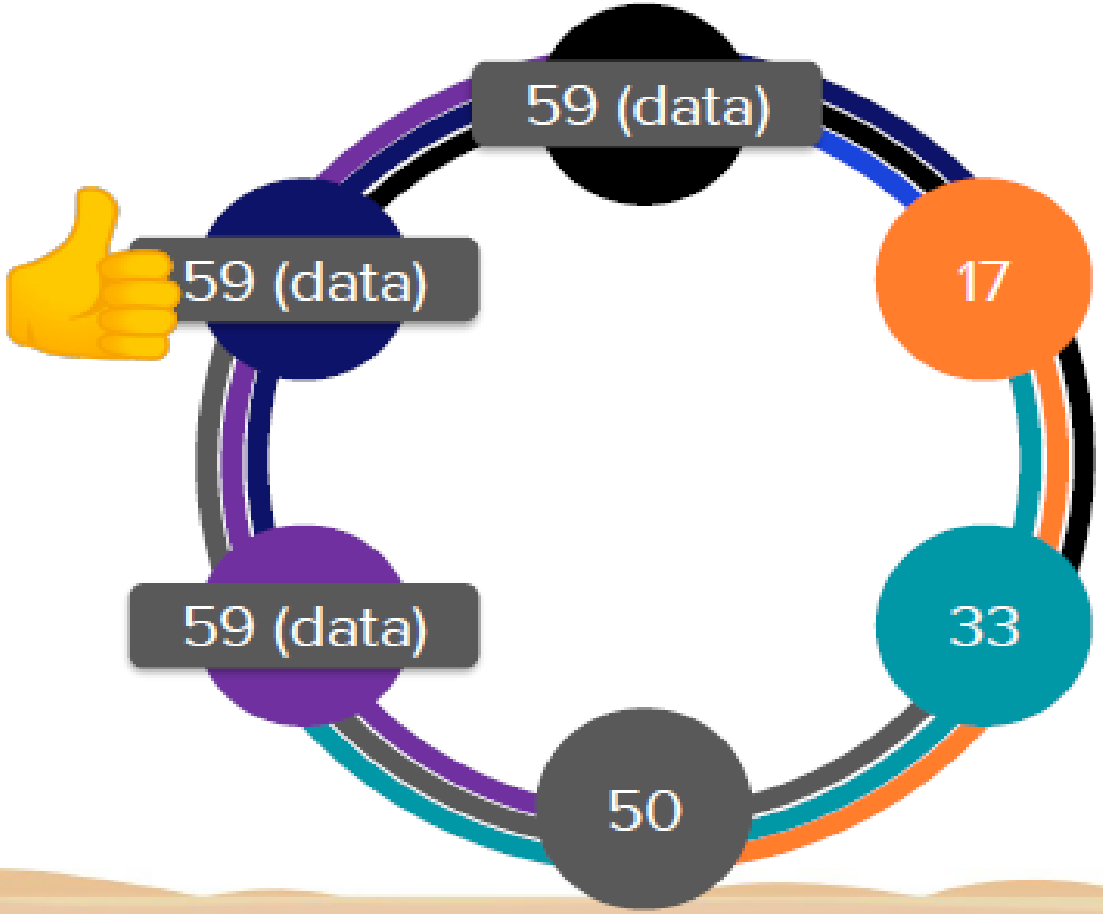
Node Failure

RF = 3



Node Failure – Recovered!

RF = 3



Anti-Entropy

- 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

Replica synchronization

- 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.
- If an insufficient number of nodes have the latest value, a read repair is performed immediately to update the out-of-date replicas.
- Otherwise, the repairs can be performed in the background after the read returns.
- This design is observed by Cassandra as well as by straight key-value stores such as Project Voldemort and Riak.

Anti-entropy repair

- Sometimes called manual repair is a manually initiated operation performed on nodes as part of a regular maintenance process.
- Type of repair is executed by using nodetool,
- 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.
- each table has its own Merkle tree; the tree is created as a snapshot during a validation compaction and is kept only as long as is required to send it to the neighboring nodes on the ring.
- Advantage of this implementation is that it reduces network I/O.

Commit Logs

- When a node receives a write operation, it immediately writes the data to a commit log.
- 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.
- 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.

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.
- In early implementations of Cassandra, memtables were stored on the JVM heap, but improvements starting with the 2.1 release have moved some memtable data to native memory, with configuration options to specify the amount of on-heap and native memory available.
- This makes Cassandra less susceptible to fluctuations in performance due to Java garbage collection.
- Optionally, Cassandra may also write data to in memory key or row caches.
- 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.

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 any-thing 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.

Like regular logfiles, commit logs have a configurable rollover threshold, and once this file size threshold is reached, the log will roll over, carrying with it any extant dirty bit flags.

SSTables

- Once a memtable is flushed to disk as an SSTable, it is immutable and cannot be changed by the application.
- 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.
- Since the 1.0 release, Cassandra has supported the compression of SSTables in order to maximize use of the available storage.
- This compression is configurable per table.
- All writes are sequential, which is the primary reason that writes perform so well in Cassandra.
- No reads or seeks of any kind are required for writing a value to Cassandra because all writes are **append** operations.
- This makes the speed of your disk one key limitation on performance. Compaction is intended to amortize the reorganization of data, but it uses sequential I/O to do so.

Memtables, SSTables, and Commit Logs

Performance benefit is gained by splitting; the write operation is just an immediate append, and then compaction helps to organize for better future read performance.

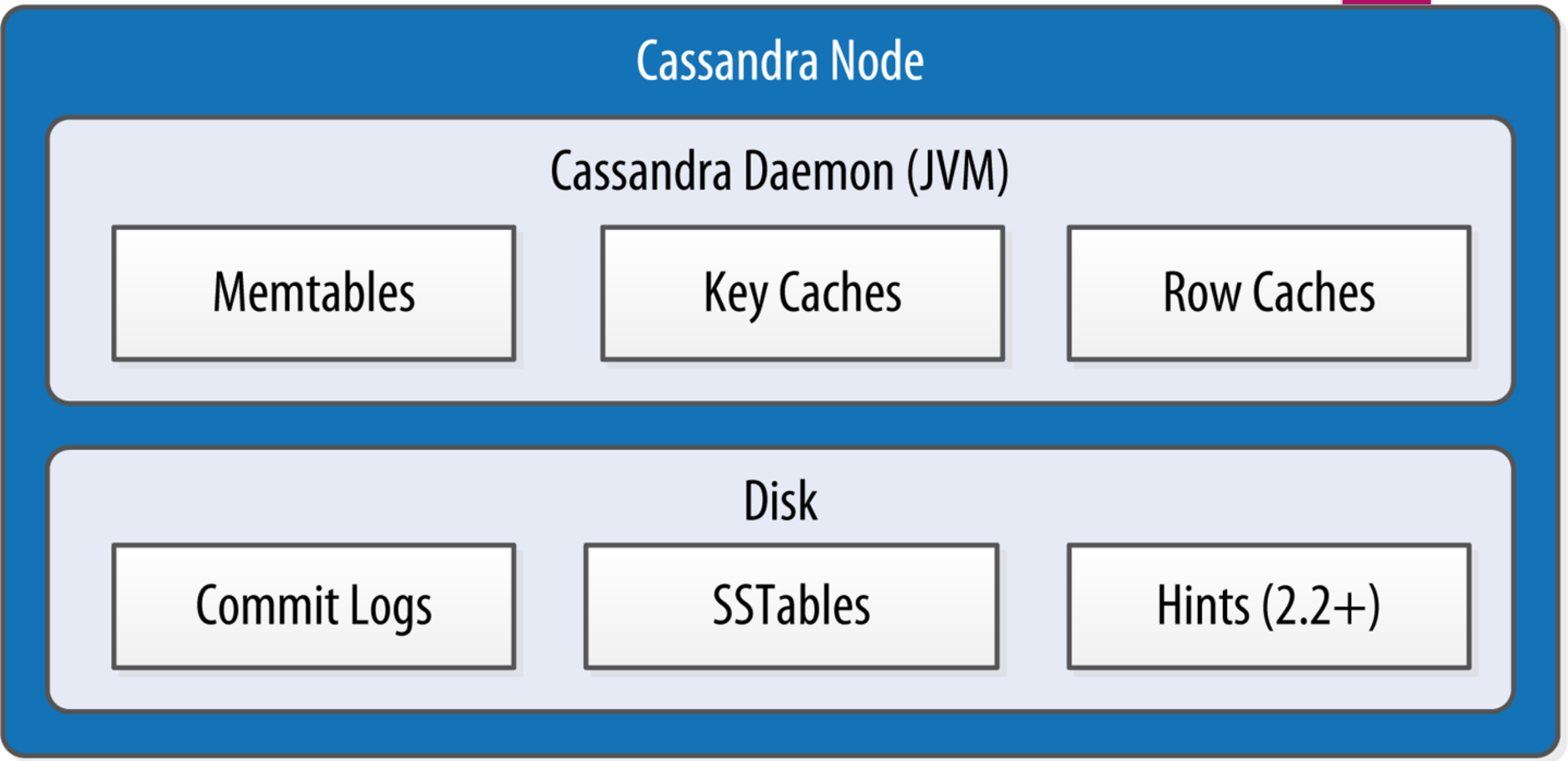
If Cassandra naively inserted values where they ultimately belonged, writing clients would pay for seeks up front.

On reads, Cassandra will read both SSTables and memtables to find data values, as the memtable may contain values that have not yet been flushed to disk

What Are Durable Writes?

```
CREATE KEYSPACE my_keyspace WITH replication =
{'class': 'SimpleStrategy',
'replication_factor': '1'} AND durable_writes =true;
```

- durable_writes property controls whether Cassandra will use the commit log for writes to the tables in the keyspace.
- Defaults to true, meaning that the commit log will be updated on modifications.
- Setting the value to false increases the speed of writes, but also risks losing data if the node goes down before the data is flushed from memtables into SSTables.


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

Write Path

RAM

DISK

Write Path

RAM

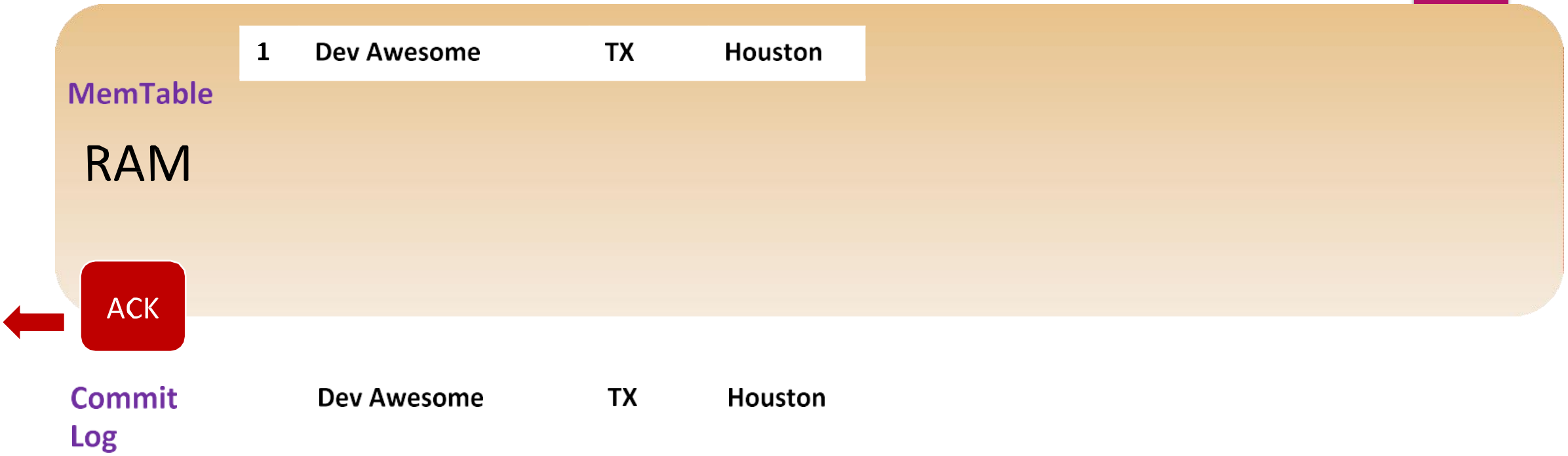
1	Dev Awesome	TX	Houston

DISK

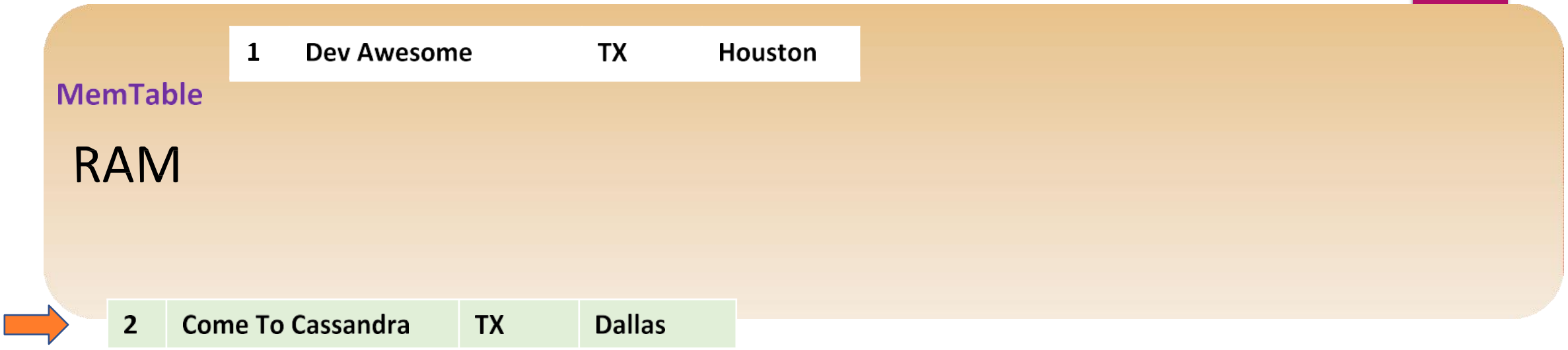
Write Path


1 Dev Awesome TX Houston
MemTable
RAM
Commit Log
Dev Awesome TX Houston

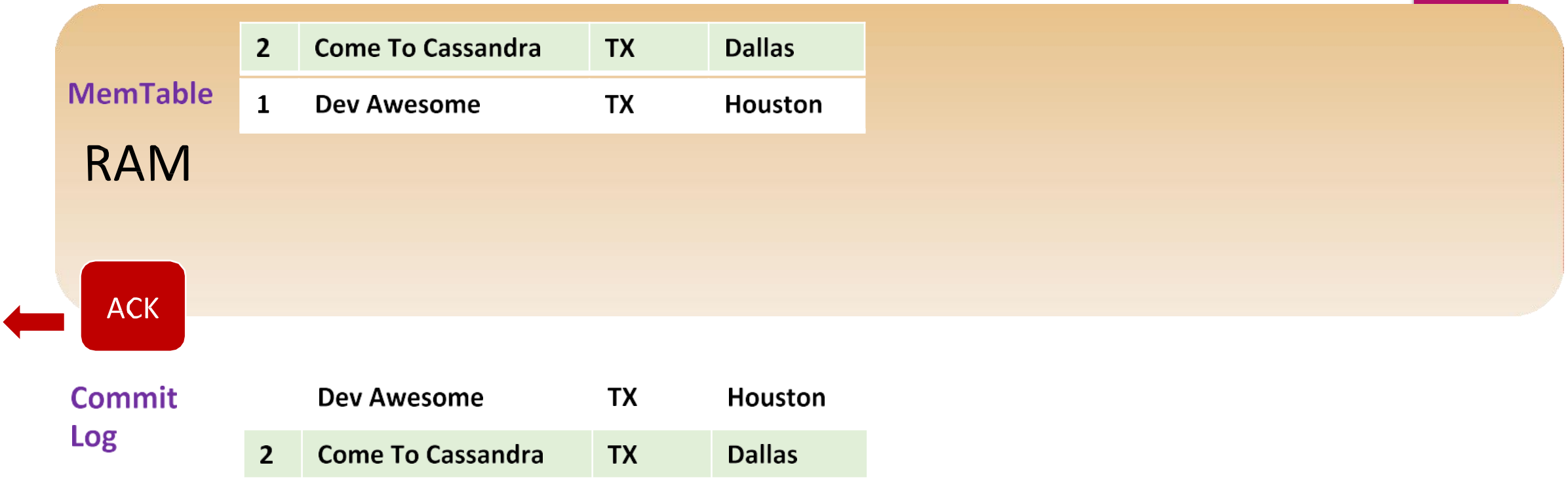
Write Path


1 Dev Awesome TX Houston
MemTable
RAM
ACK
Commit Log
Dev Awesome TX Houston

Write Path


1 Dev Awesome TX Houston
MemTable
RAM
2 Come To Cassandra TX Dallas
Commit Log
Dev Awesome TX Houston

Write Path


MemTable
2 Come To Cassandra TX Dallas
1 Dev Awesome TX Houston
RAM
ACK
Commit Log
Dev Awesome TX Houston
2 Come To Cassandra TX Dallas

Write Path

MemTable

RAM

2	Come To Cassandra	TX	Dallas
1	Dev Awesome	TX	Houston

3	Lone Node	TX	Snyder

Commit Log

	Dev Awesome	TX	Houston
2	Come To Cassandra	TX	Dallas

Write Path

MemTable

RAM

2	Come To Cassandra	TX	Dallas
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

Commit Log

	Dev Awesome	TX	Houston
2	Come To Cassandra	TX	Dallas
3	Lone Node	TX	Snyder

Write Path

MemTable

RAM

2	Come To Cassandra	TX	Dallas
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

4	IgotUr Data	TX	Austin

Commit Log

	Dev Awesome	TX	Houston
2	Come To Cassandra	TX	Dallas
3	Lone Node	TX	Snyder

Write Path

MemTable

RAM

4	IgotUr Data	TX	Austin
2	Come To Cassandra	TX	Dallas
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

ACK

Commit Log

	Dev Awesome	TX	Houston
2	Come To Cassandra	TX	Dallas
3	Lone Node	TX	Snyder
4	IgotUr Data	TX	Austin

Write Path

MemTable

RAM

4	IgotUr Data	TX	Austin
2	Come To Cassandra	TX	Dallas
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

5	Lone Star	TX	El Paso

Commit Log

	Dev Awesome	TX	Houston
2	Come To Cassandra	TX	Dallas
3	Lone Node	TX	Snyder
4	IgotUr Data	TX	Austin

Write Path

MemTable

RAM

4	IgotUr Data	TX	Austin
2	Come To Cassandra	TX	Dallas
5	Lone Star	TX	El Paso
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

ACK

Commit Log

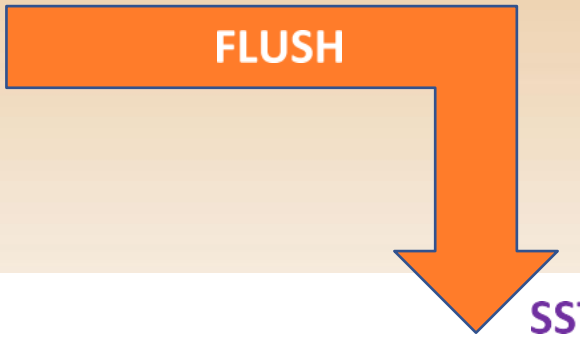
	Dev Awesome	TX	Houston
2	Come To Cassandra	TX	Dallas
3	Lone Node	TX	Snyder
4	IgotUr Data	TX	Austin
5	Lone Star	TX	El Paso

Write Path

MemTable

RAM

4	IgotUr Data	TX	Austin
2	Come To Cassa	TX	Dallas
5	Lone Star	TX	El Paso
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder



Commit Log

	Dev Awesome	TX	Houston
2	Come To Cassandra	TX	Dallas
3	Lone Node	TX	Snyder
4	IgotUr Data	TX	Austin
5	Lone Star	TX	El Paso

SSTABLE

4	IgotUr Data	TX	Austin
2	Come To Cassandra	TX	Dallas
5	Lone Star	TX	El Paso
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

Write Path

MemTable

RAM

Commit Log

	Dev Awesome	TX	Houston
2	Come To Cassandra		Dallas
3	Lone Node	X	Snyder
4	IgotUr Data		Austin
5	Lone Star	TX	El Paso

SSTABLE

4	IgotUr Data	TX	Austin
2	Come To Cassandra	TX	Dallas
5	Lone Star	TX	El Paso
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

Write Path

MemTable

RAM

DISK

SSTABLE
(IMMUTABLE)

4	IgotUr Data	TX	Austin
2	Come To Cassandra	TX	Dallas
5	Lone Star	TX	El Paso
1	Dev Awesome	TX	Houston
3	Lone Node	TX	Snyder

Bloom Filters

- Bloom filters are used to boost the performance of reads.
- Are very fast, nondeterministic algorithms for testing whether an element is a member of a set.
- 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.
- The digest, by definition, uses a much smaller amount of memory than the original data would.
- The filters are stored in memory and are used to improve performance by reducing the need for disk access on key lookups.
- Disk access is typically much slower than memory access.
- So, in a way, a Bloom filter is a special kind of key cache.

Bloom Filters

- Cassandra maintains a Bloom filter for each SSTable.
- When a query is performed, the Bloom filter is checked first before accessing disk.
- Because false-negatives are not possible, if the filter indicates that the element does not exist in the set, it certainly doesn't; but if the filter thinks that the element is in the set, the disk is accessed to make sure.
- Cassandra provides the ability to increase Bloom filter accuracy (reducing the number of false-positives) by increasing the filter size, at the cost of more memory.
- This false-positive chance is tuneable per table.

How is data read?

- To satisfy a read, Cassandra must combine results from the active memtable and potentially multiple SSTables.

Cassandra processes data at several stages on the read path to discover where the data is stored, starting with the data in the memtable and finishing with SSTables:

- Check the memtable
- Check row cache, if enabled
- Checks Bloom filter
- Checks partition key cache, if enabled
- Goes directly to the compression offset map if a partition key is found in the partition key cache, or checks the partition summary if not
- If the partition summary is checked, then the partition index is accessed
- Locates the data on disk using the compression offset map
- Fetches the data from the SSTable on disk

Step 1:

- If the memtable has the desired partition data, then the data is read and then merged with the data from the SSTables.
- The SSTable data is accessed as shown in the following steps.

Step 2: Row Cache

- Row cache, if enabled, stores a subset of the partition data stored on disk in the SSTables in memory.
- The subset stored in the row cache use a configurable amount of memory for a specified period of time.
- Row cache uses LRU (least-recently-used) eviction to reclaim memory when the cache has filled up.
- If row cache is enabled, desired partition data is read from the row cache, potentially saving two seeks to disk for the data.
- Rows stored in row cache are frequently accessed rows that are merged and saved to the row cache from the SSTables as they are accessed.
- After storage, the data is available to later queries.
- Row cache is not write-through. If a write comes in for the row, the cache for that row is invalidated and is not cached again until the row is read.
- Similarly, if a partition is updated, the entire partition is evicted from the cache. When the desired partition data is not found in the row cache, then the Bloom filter is checked.

Step 3: Bloom Filter

- First, Cassandra checks the Bloom filter to discover which SSTables are likely to have the request partition data.
- Is stored in off-heap memory.
- Each SSTable has a Bloom filter associated with it.
- Can establish that a SSTable does not contain certain partition data.
- Can also find the likelihood that partition data is stored in a SSTable.
- Speeds up the process of partition key lookup by narrowing the pool of keys.
- Because the Bloom filter is a probabilistic function, it can result in false positives.
- Not all SSTables identified by the Bloom filter will have data.
- If the Bloom filter does not rule out an SSTable, Cassandra checks the partition key cache
- The Bloom filter grows to approximately 1-2 GB per billion partitions.
- In the extreme case,can have one partition per row, so you can easily have billions of these entries on a single machine. The Bloom filter is tunable if you want to trade memory for performance.

Step 4: Partition Key Cache

- Partition key cache, if enabled, stores a cache of the partition index in off-heap memory.
- Uses a small, configurable amount of memory, and each "hit" saves one seek during the read operation.
- If a partition key is found in the key cache can go directly to the compression offset map to find the compressed block on disk that has the data.
- If a partition key is not found in the key cache, then the partition summary is searched.

Step 5:Partition Summary

- Partition summary is an off-heap in-memory structure that stores a sampling of the partition index.
- A partition index contains all partition keys, whereas a partition summary samples every X keys, and maps the location of every Xth key's location in the index file.
- For example, if the partition summary is set to sample every 20 keys, it will store the location of the first key as the beginning of the SSTable file, the 20th key and its location in the file, and so on.
- While not as exact as knowing the location of the partition key, the partition summary can shorten the scan to find the partition data location.
- After finding the range of possible partition key values, the partition index is searched.
- By configuring the sample frequency, you can trade memory for performance, as the more granularity the partition summary has, the more memory it will use.
- The sample frequency is changed using the index interval property in the table definition.
- A fixed amount of memory is configurable using the index_summary_capacity_in_mb property, and defaults to 5% of the heap size.

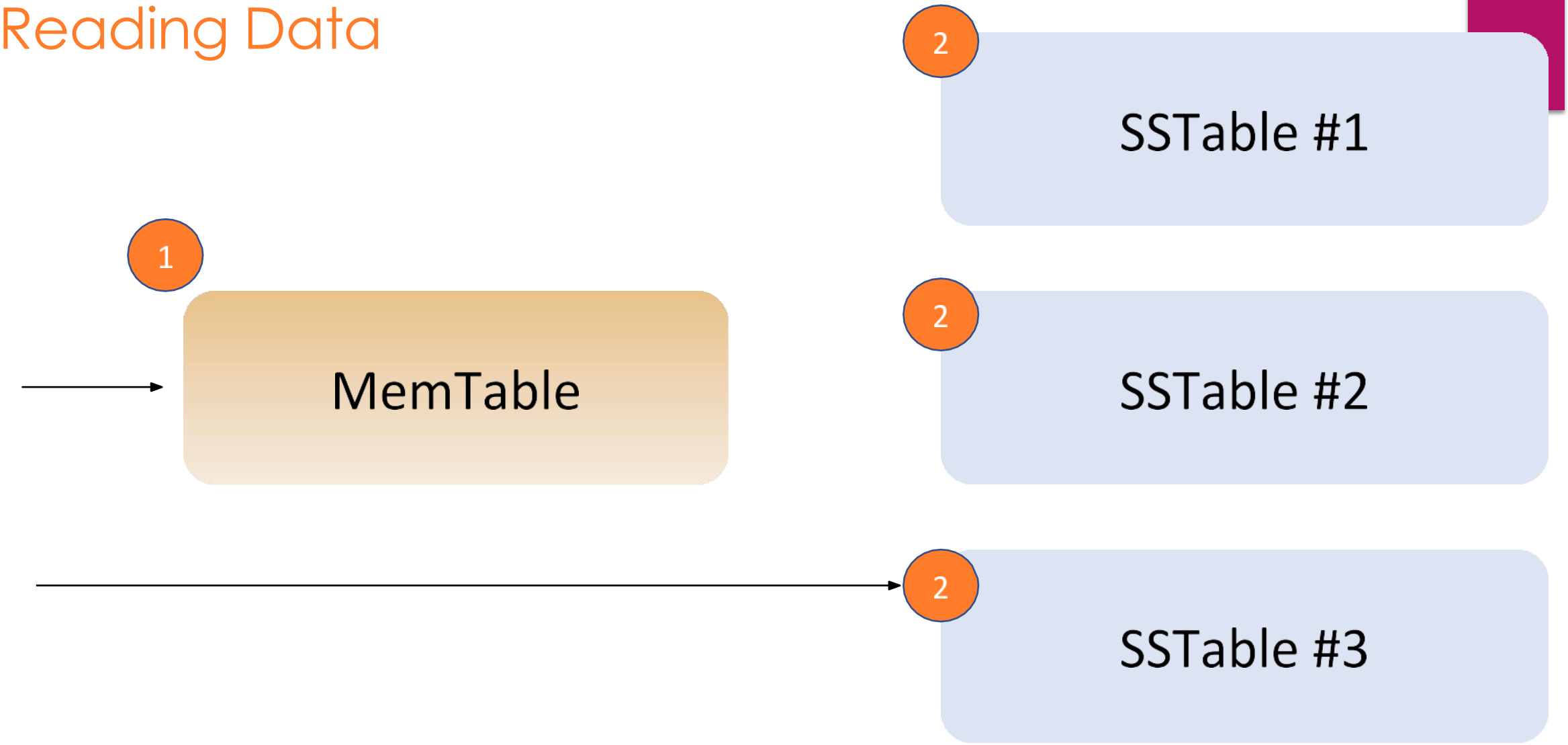
Step 6: Partition Index

- Partition index resides on disk and stores an index of all partition keys mapped to their offset.
- If the partition summary has been checked for a range of partition keys, now the search passes to the partition index to seek the location of the desired partition key.
- A single seek and sequential read of the columns over the passed-in range is performed.
- Using the information found, the partition index now goes to the compression offset map to find the compressed block on disk that has the data.
- If the partition index must be searched, two seeks to disk will be required to find the desired data.

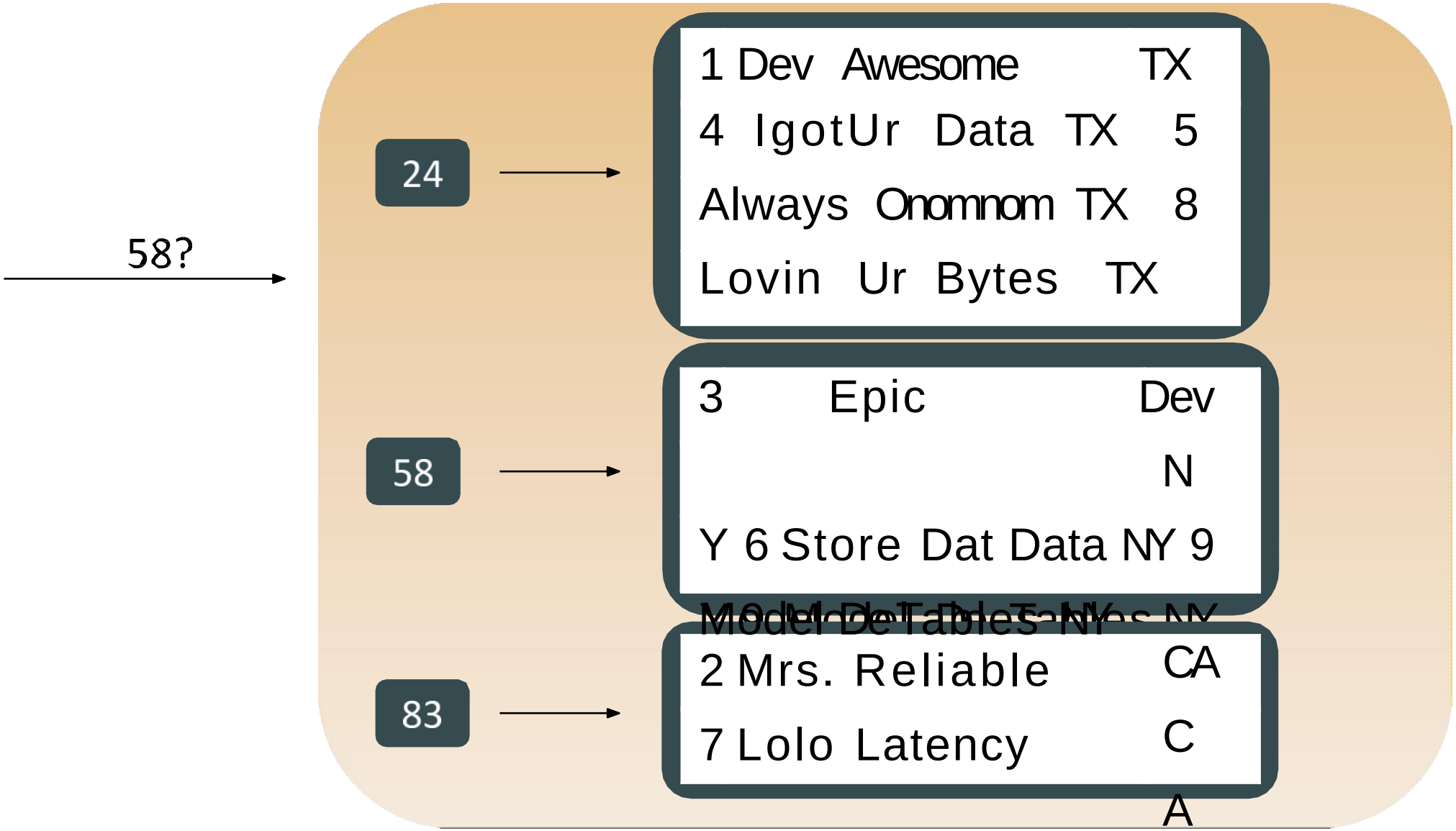
Step 7: Compression offset map

- Compression offset map stores pointers to the exact location on disk that the desired partition data will be found.
- Stored in off-heap memory and is accessed by either the partition key cache or the partition index.
- Desired compressed partition data is fetched from the correct SSTable(s) once the compression offset map identifies the disk location.
- The query receives the result set.
- The compression offset map grows to 1-3 GB per terabyte compressed. The more you compress data, the greater number of compressed blocks you have and the larger the compression offset table.
- Compression is enabled by default even though going through the compression offset map consumes CPU resources. Having compression enabled makes the page cache more effective, and typically, almost always pays off.

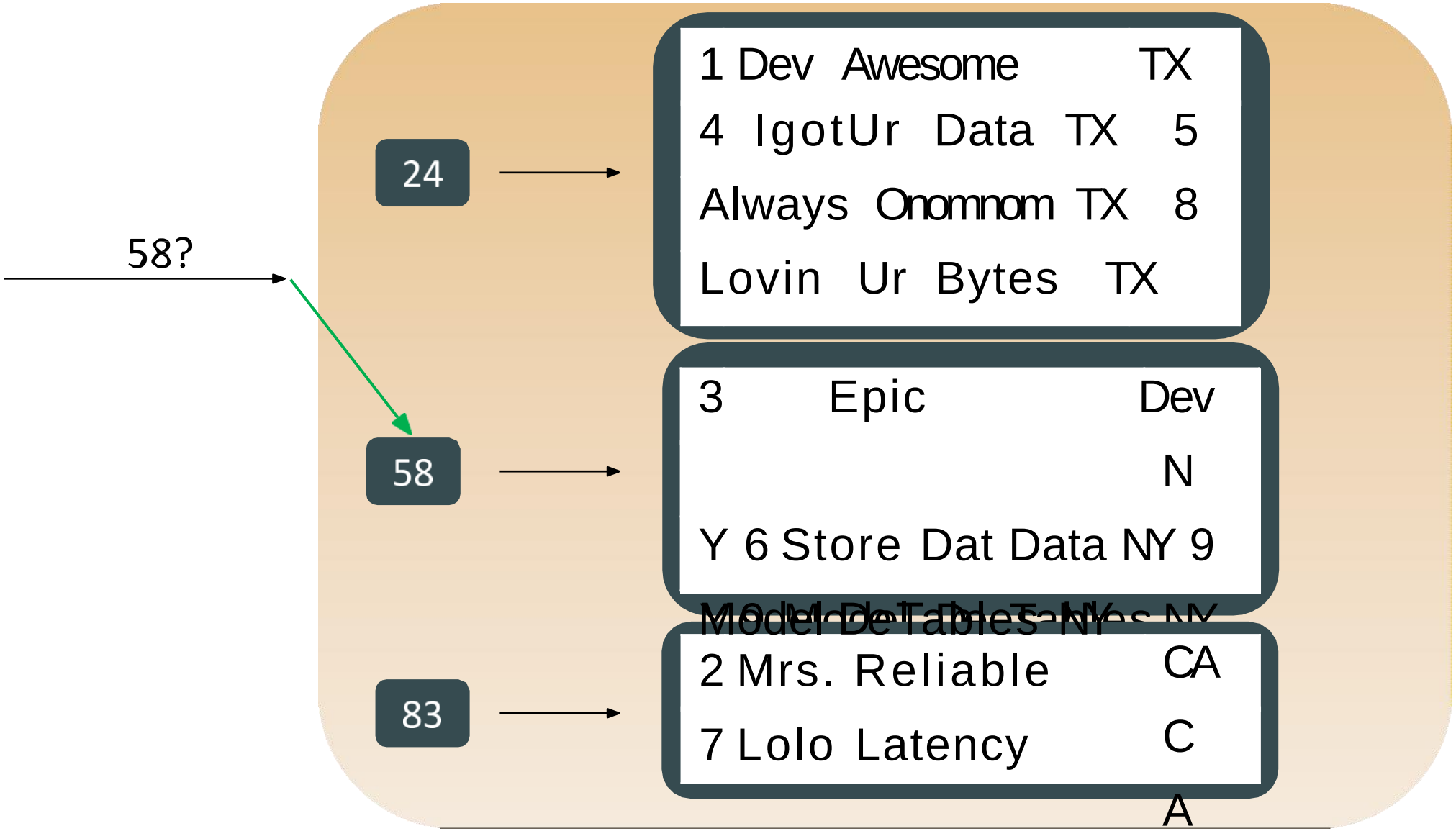
Reading Data



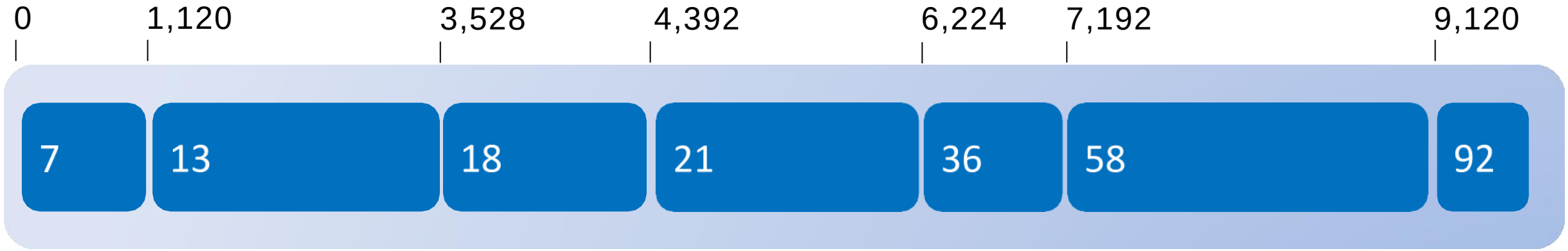
Reading a MemTable



Reading a MemTable



Reading a SSTable

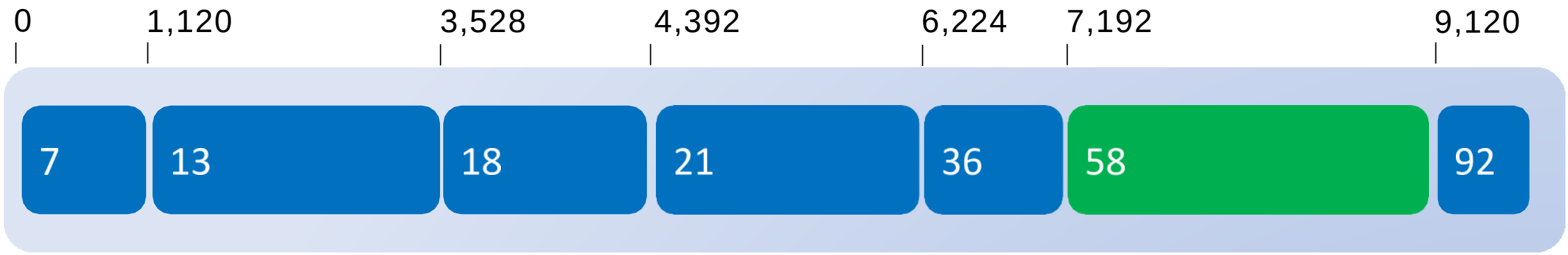


- A SSTable holds ordered partitions
- A partition can be split in multiple SSTables
- We can mark offset of each partition

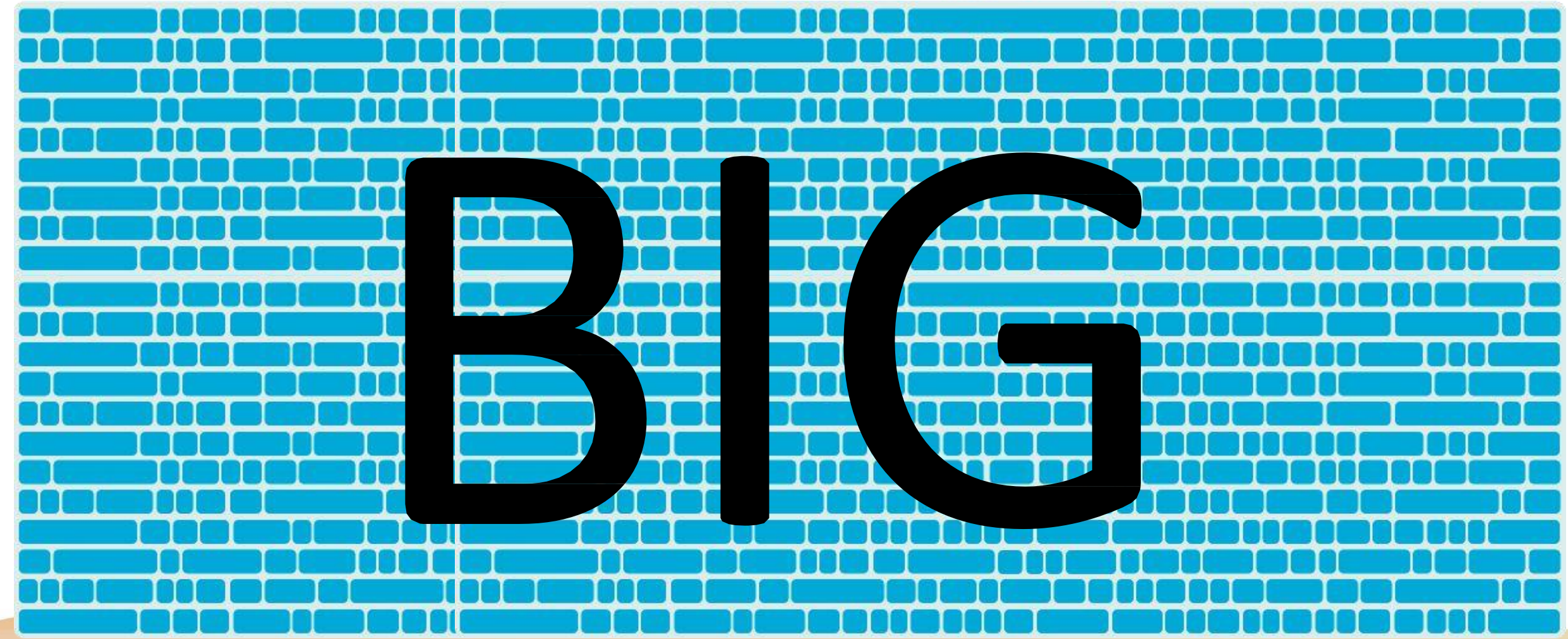
Partition Index (on Disk)

Token	Byte Offset
7	0
13	1,120
18	3,528
21	4,392
36	6,224
58	7,192
92	9,120

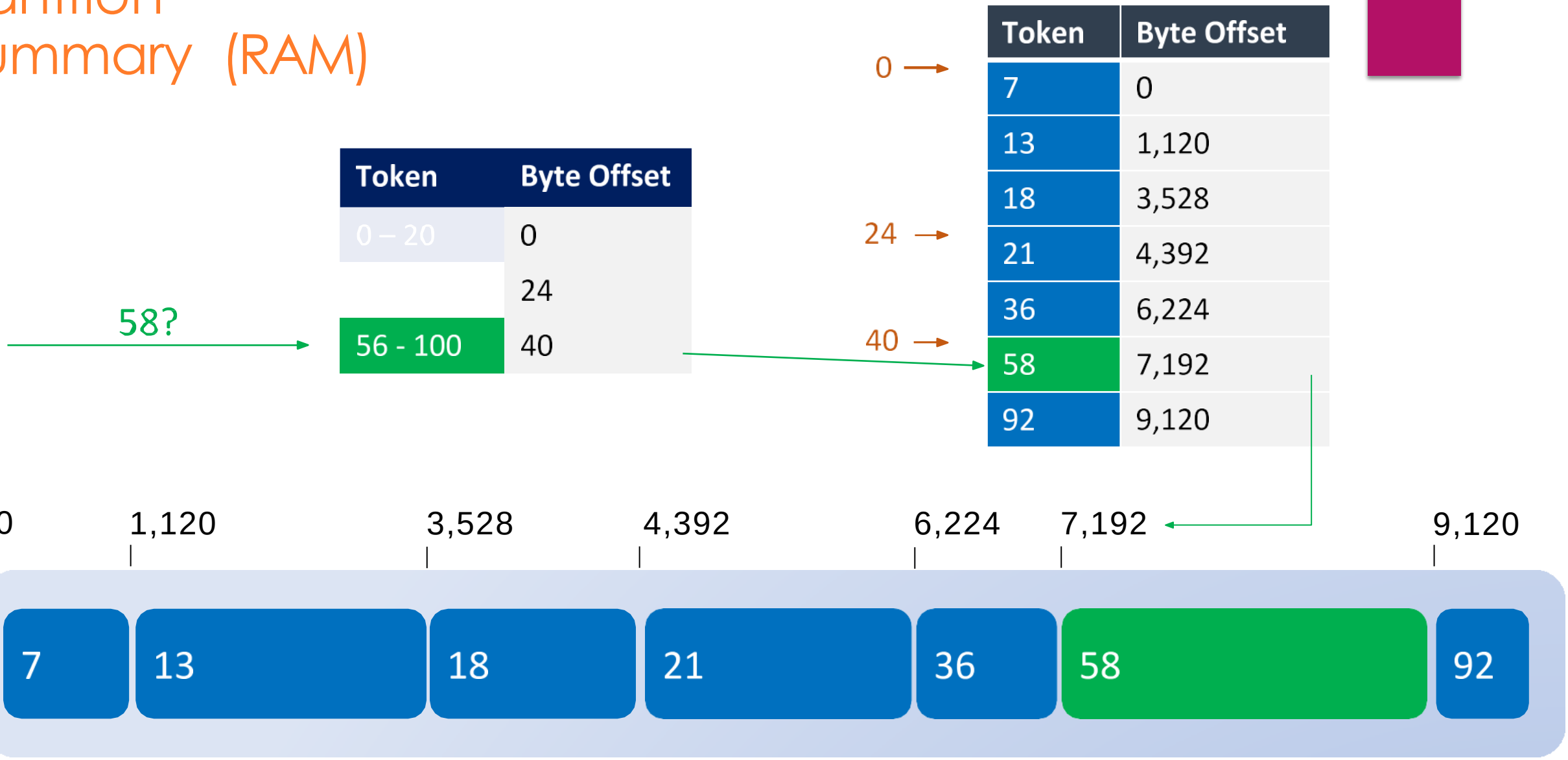
58?



…but a SSTABLE is …



Partition Summary (RAM)



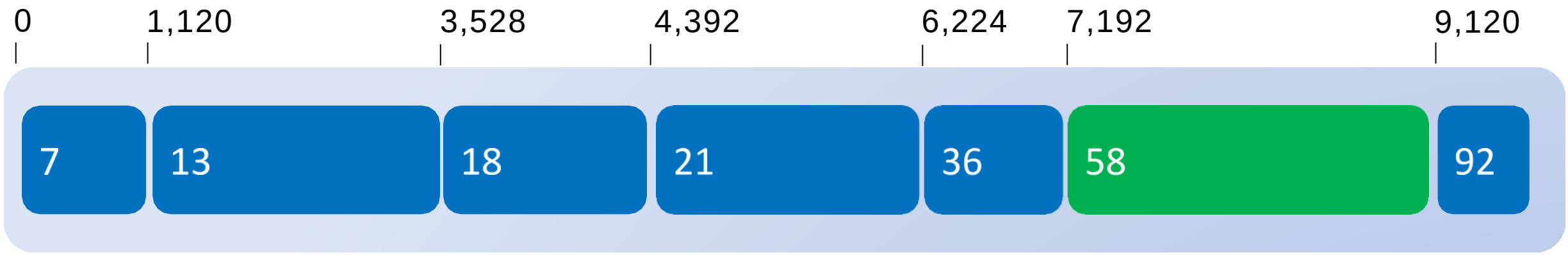
Key Cache (RAM)

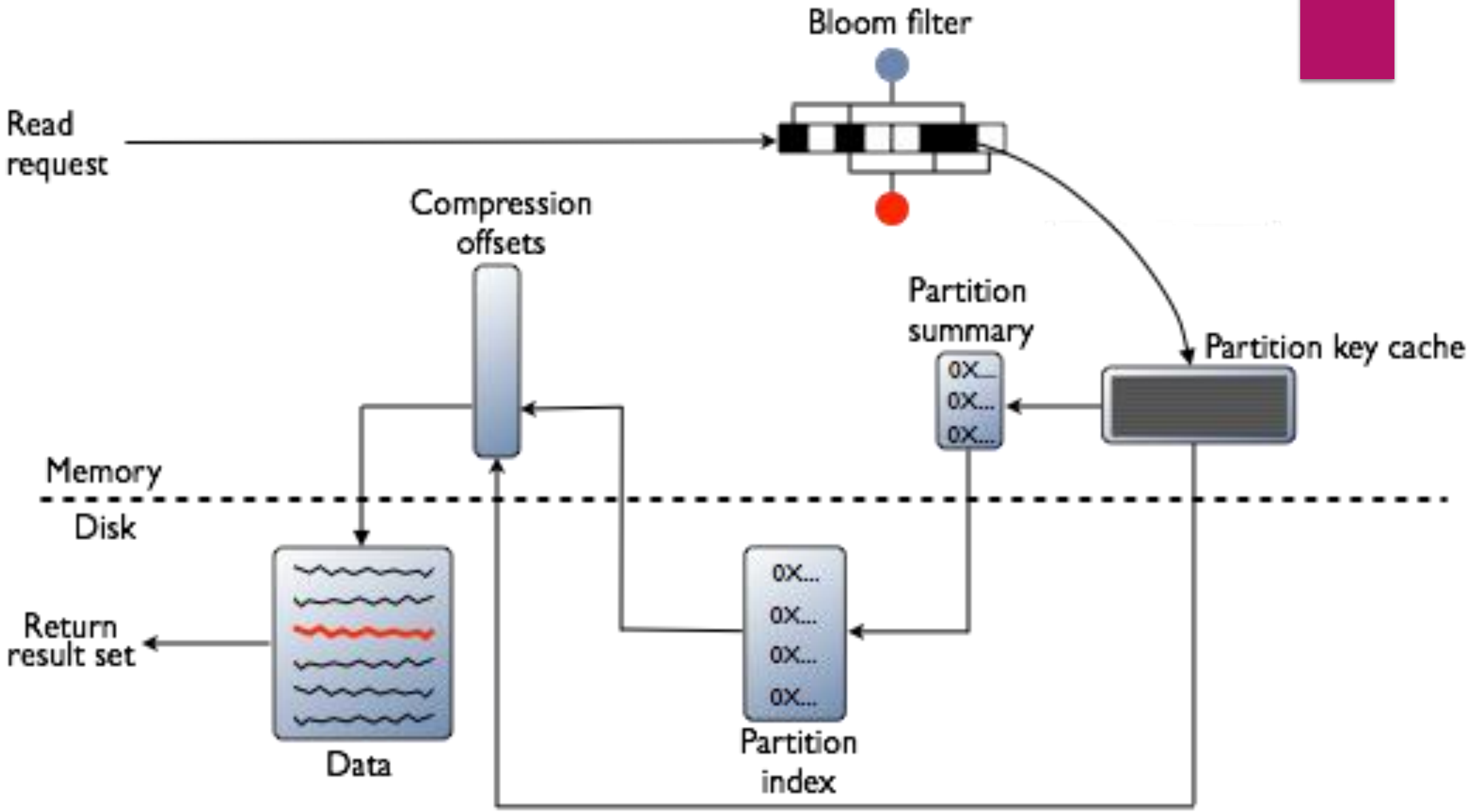
Token	Byte Offset
0 – 20	0
	24
	40

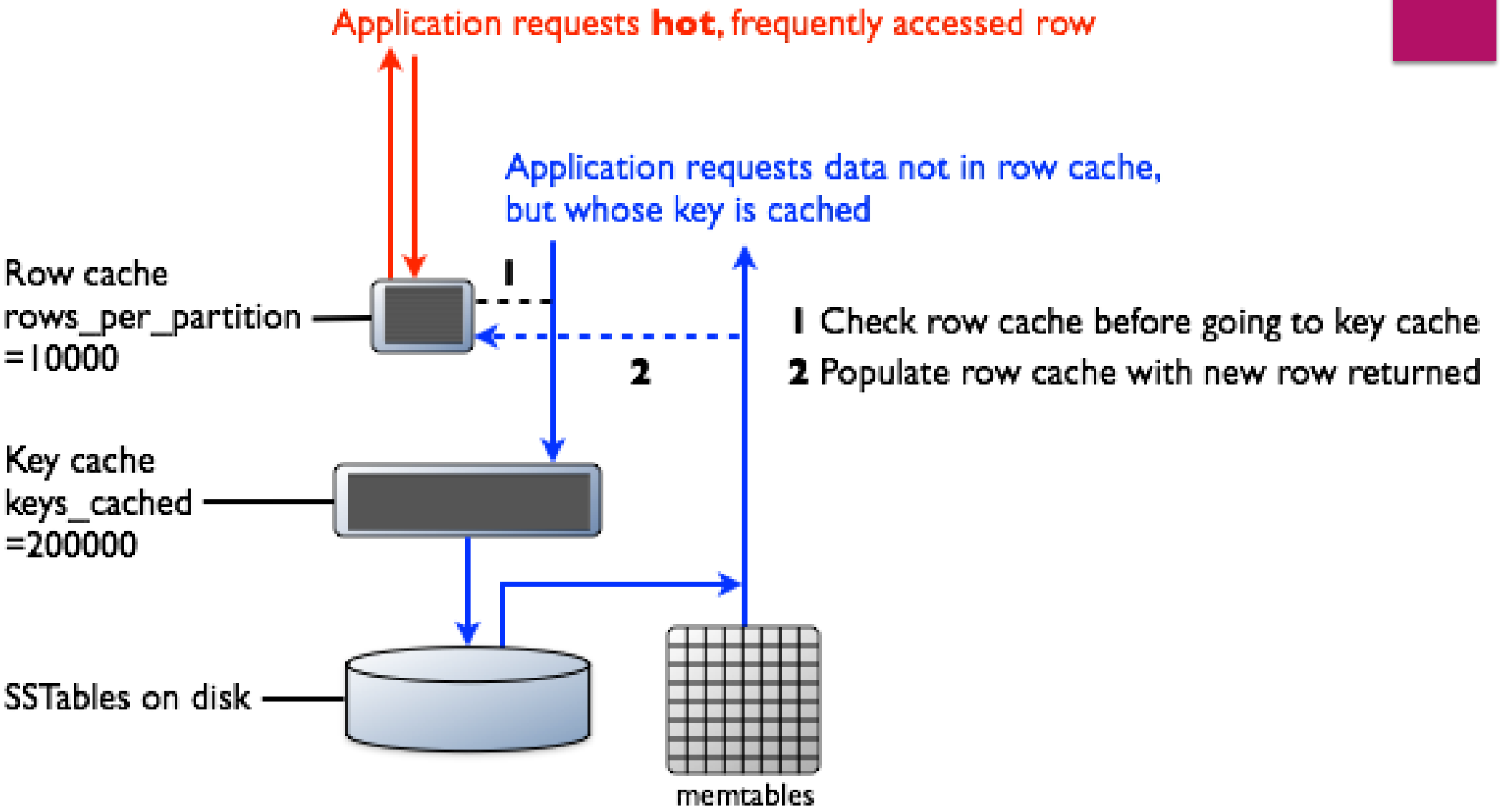
Token	Byte Offset
7	0
13	1,120
18	3,528
21	4,392
36	6,224
58	7,192
92	9,120

58?

Token	Byte Offset
58	7192




Bloom filter
Read request
Compression offsets
Partition summary
Partition key cache
0X...
0X...
0X...
Memory
Disk
Return result set
0X...
0X...
0X...
0X...
Data
Partition index


Application requests **hot**, frequently accessed row
Application requests data not in row cache, but whose key is cached
Row cache
rows_per_partition
=10000
Key cache
keys_cached
=200000
SSTables on disk
memtables
1
2
1 Check row cache before going to key cache
2 Populate row cache with new row returned

For each SSTable – files created

Data (Data.db) ---The SSTable data

Primary Index (Index.db) --- Index of the row keys with pointers to their positions in the data file

Bloom filter **(Filter.db)** --- A structure stored in memory that checks if row data exists in the memtable before accessing SSTables on disk

Compression Information (CompressionInfo.db) --- A file holding information about uncompressed data length, chunk offsets and other compression information

Statistics (Statistics.db) --- Statistical metadata about the content of the SSTable

Digest (Digest.crc32, Digest.adler32, Digest.sha1) ---A file holding adler32 checksum of the data file

CRC (CRC.db) ---A file holding the CRC32 for chunks in an uncompressed file.

SSTable Index Summary (SUMMARY.db) ---A sample of the partition index stored in memory

SSTable Table of Contents (TOC.txt) ---A file that stores the list of all components for the SSTable TOC

Secondary Index (SI_.*.db) --- Built-in secondary index. Multiple SIs may exist per SSTable

Data Caching

- Cassandra includes integrated caching and distributes cache data around the cluster for you.
- Integrated cache solves the cold start problem by virtue of saving your cache to disk periodically and being able to read contents back in when it restarts.
- Never have to start with a cold cache.

Caching

- As an additional mechanism to boost read performance, Cassandra provides three optional forms of caching:
- **Key cache** stores a map of partition keys to row index entries, facilitating faster read access into SSTables stored on disk.
  - Stored on the JVM heap.
- **Row cache** caches entire rows and can greatly speed up read access for frequently accessed rows, at the cost of more memory usage.
  - Row cache is stored in off-heap memory.
- **Counter cache** -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.

partition key cache

- Cache of the partition index for a Cassandra table.
- Using the key cache instead of relying on the OS page cache decreases seek times.
- Enabling just the key cache results in disk (or OS page cache) activity to actually read the requested data rows, but not enabling the key cache results in more reads from disk.
- Read request arrives first to Bloom filter.
- It decides if needed data can be stored in one of managed SSTables.
- If it's not the case, the action stops.
- If it is, Cassandra asks partition key cache where partition holding the data begins in the SSTables.
- Thanks to that, Cassandra goes directly to the row containing expected data.
- Without the use of key cache, Cassandra should look first at index and scan in to find good key range for the queried data.

Key caching

- Key cache is configurable through several entries in cassandra.yaml file:

- key_cache_size_in_mb: the maximum size of the key cache in memory. By default in 3.4 release, this value is equal 5% of JVM heap or 100 mb if 5% of heap is greater than 100mb.
- key_cache_save_period: determines the number of ms after which key caches will persist to disk. The default value is 4 hours. Thanks to this persistance mechanism, Cassandra improves the performances even after reboot, when memory is cleaned.
- key_cache_keys_to_save: number of keys to save. By default this entry is not used.
- This cache should not take a lot of space and its use is encouraged to avoid too many disk seeks during rows reading.

row cache

- Puts a part of partition into memory.
- Configure the number of rows to cache in a partition by setting the rows_per_partition table option.
- To cache rows, if the row key is not already in the cache, Cassandra reads the first portion of the partition, and puts the data in the cache.
- If the newly cached data does not include all cells configured by user, Cassandra performs another read.
- Actual size of the row-cache depends on the workload.
- Should properly benchmark your application to get "the best" row cache size to configure.
- Two row cache options, the old serializing cache provider and a new off-heap cache (OHC) provider.
- New OHC provider has been benchmarked as performing about 15% better than the older option.
- Note: Utilizing appropriate OS page cache will result in better performance than using row caching. Consult resources for page caching for the operating system on which Cassandra is hosted.

Row caching

- If one of stored rows changes, it and its partition must be invalidated from cache.
- So, row cache should privilege rows frequently read but not frequently modified.
- In additional, in the versions before 2.1, row cache worked by whole partition.
- It means that whole partition had to be stored in the cache. And if its size was bigger than the size of available cache memory, the row never was cached.
- Since 2.1 release --Can configure row cache to kept only specific number of rows.
- Row cache size is also configurable, as is the number of rows to store.
- Configuring the number of rows to be stored is a useful feature, making a "Last 10 Items" query very fast to read.
- In Cassandra 2.2 and later, it is stored in fully off-heap memory using a new implementation that relieves garbage collection pressure in the JVM.

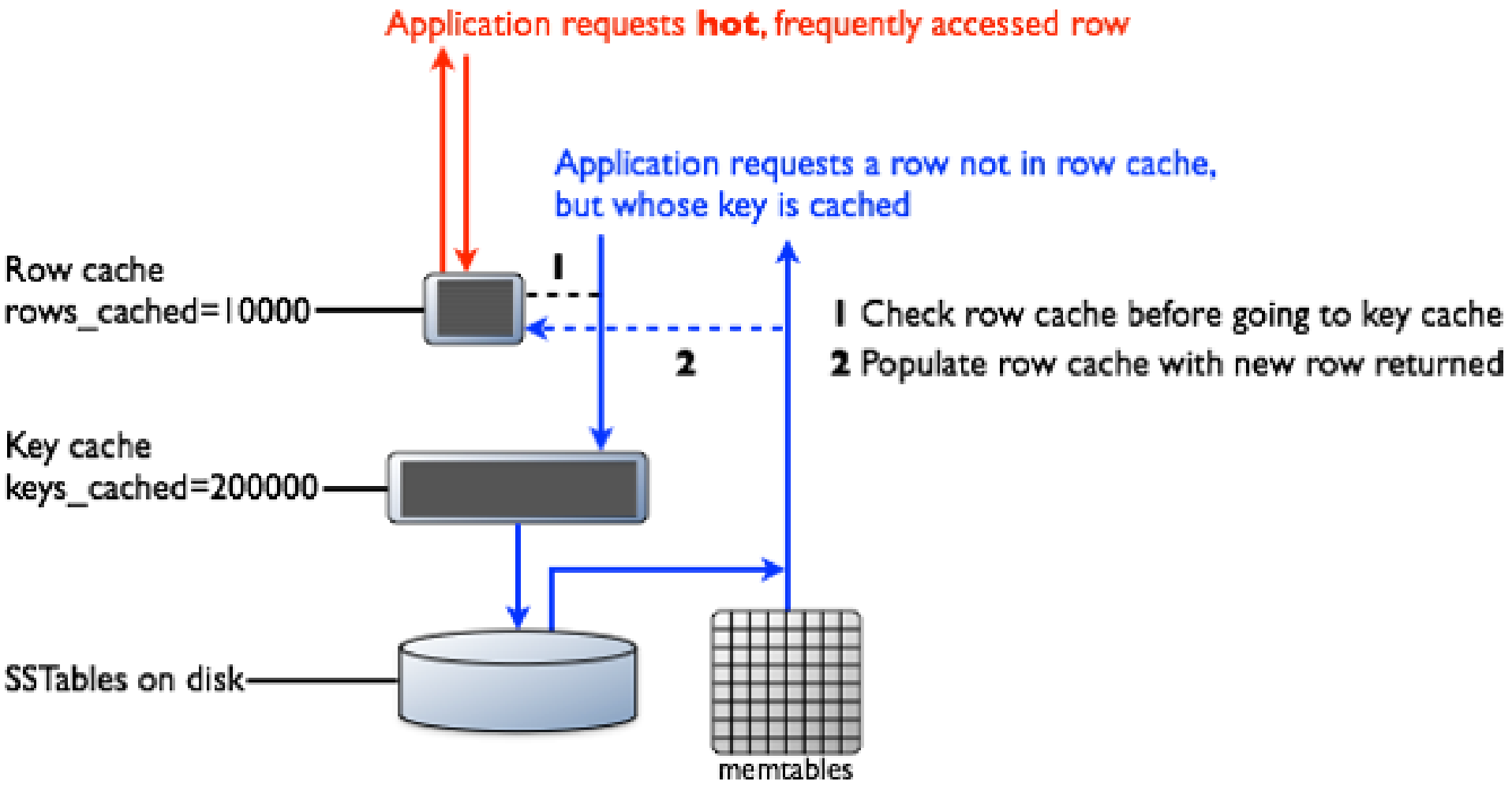
Row caching

- row_cache_size_in_mb - as for key cache, the maximum size. However the rule of 100MB is not applied. By default, the row cache is disabled (value is 0).
- row_cache_save_period - the time after which the cache is persisted to disk.
- row_cache_keys_to_save - the number of row cache keys to save.
- row_cache_class_name - the implementation for row cache management.

Apart from that, row caching can be configured at table creation through caching clause.

Counter cache in Cassandra

- Counter cache -- related to counter columns.
- Role consists to help locks contention for frequently updated cells of counter type.
- counter_cache_size_in_mb, counter_cache_save_period, counter_cache_keys_to_save.



One read operation hits the row cache, returning the requested row without a disk seek. The other read operation requests a row that is not present in the row cache but is present in the partition key cache. After accessing the row in the SSTable, the system returns the data and populates the row cache with this read operation.

Using key cache and row cache

- Typically, enable either the partition key or row cache for a table.
- Enable a row cache only when the number of reads is much bigger (rule of thumb is 95%) than the number of writes.
- Consider using the operating system page cache instead of the row cache, because writes to a partition invalidate the whole partition in the cache.
- Disable caching entirely for archive tables, which are infrequently read.

efficient cache use

- Store lower-demand data or data with extremely long partitions in a table with minimal or no caching.
- Deploy a large number of Cassandra nodes under a relatively light load per node.
- Logically separate heavily-read data into discrete tables.
- When you query a table, turn on tracing to check that the table actually gets data from the cache rather than from disk.
- Requesting rows not at the beginning of the partition is a likely cause for the cache not being used.
- Try removing constraints that might cause the query to skip the beginning of the partition, or place a limit on the query to prevent results from overflowing the cache.
- To ensure that the query hits the cache, try increasing the cache size limit, or restructure the table to position frequently accessed rows at the head of the partition.

Compaction

SSTables are immutable, which helps Cassandra achieve such high write speeds.

However, periodic compaction of these SSTables is important in order to support fast read performance and clean out stale data values.

A compaction operation in Cassandra is performed in order to merge SSTables.

During compaction, the data in SSTables is merged: the keys are merged, columns are combined, obsolete values are discarded, and a new index is created.

Compaction is the process of freeing up space by merging large accumulated datafiles.

This is roughly analogous to rebuilding a table in the relational world.

Compaction

- On compaction, the merged data is sorted, a new index is created over the sorted data, and the freshly merged, sorted, and indexed data is written to a single new SSTable (each SSTable consists of multiple files, including Data, Index, and Filter).
- Another important function of compaction is to improve performance by reducing the number of required seeks.
- There is a bounded number of SSTables to inspect to find the column data for a given key.
- If a key is frequently mutated, it's very likely that the mutations will all end up in flushed SSTables.
- Compacting them prevents the database from having to perform a seek to pull the data from each SSTable in order to locate the current value of each column requested in a read request.
- When compaction is performed, there is a temporary spike in disk I/O and the size of data on disk while old SSTables are read and new SSTables are being written.

Types of compaction

- Minor compaction --- triggered automatically in Cassandra.
- Major compaction ---a user executes a compaction over all sstables on the node.
- User defined compaction --a user triggers a compaction on a given set of sstables.
- Scrub ---try to fix any broken sstables. This can actually remove valid data if that data is corrupted, if that happens you will need to run a full repair on the node.
- Upgradesstables ---upgrade sstables to the latest version. Run this after upgrading to a new major version.

Types of compaction

- Cleanup ---remove any ranges this node does not own anymore, typically triggered on neighbouring nodes after a node has been bootstrapped since that node will take ownership of some ranges from those nodes.
- Secondary index rebuild ---rebuild the secondary indexes on the node.
- Anticompaction ---after repair the ranges that were actually repaired are split out of the sstables that existed when repair started.
- Sub range compaction ---It is possible to only compact a given sub range –
  - Could be useful if you know a token that has been misbehaving - either gathering many updates or many deletes.
  - (nodetool compact -st x -et y) will pick all sstables containing the range between x and y and issue a compaction for those sstables.

When is a minor compaction triggered?

- # When an sstable is added to the node through flushing/streaming etc.
- # When autocompaction is enabled after being disabled (nodetool enableautocompaction)
- # When compaction adds new sstables.
- # A check for new minor compactions every 5 minutes.

Merging sstables

- Compaction is about merging sstables, since partitions in sstables are sorted based on the hash of the partition key it is possible to efficiently merge separate sstables.
- Content of each partition is also sorted so each partition can be merged efficiently.

Types of compaction strategies

SizeTieredCompactionStrategy (STCS) is the default compaction strategy and is recommended for write-intensive tables

LeveledCompactionStrategy (LCS) is recommended for read-intensive tables

TimeWindowCompactionStrategy (TWCS) is intended for time series or otherwise date-based data.

Size Tiered Compaction Strategy

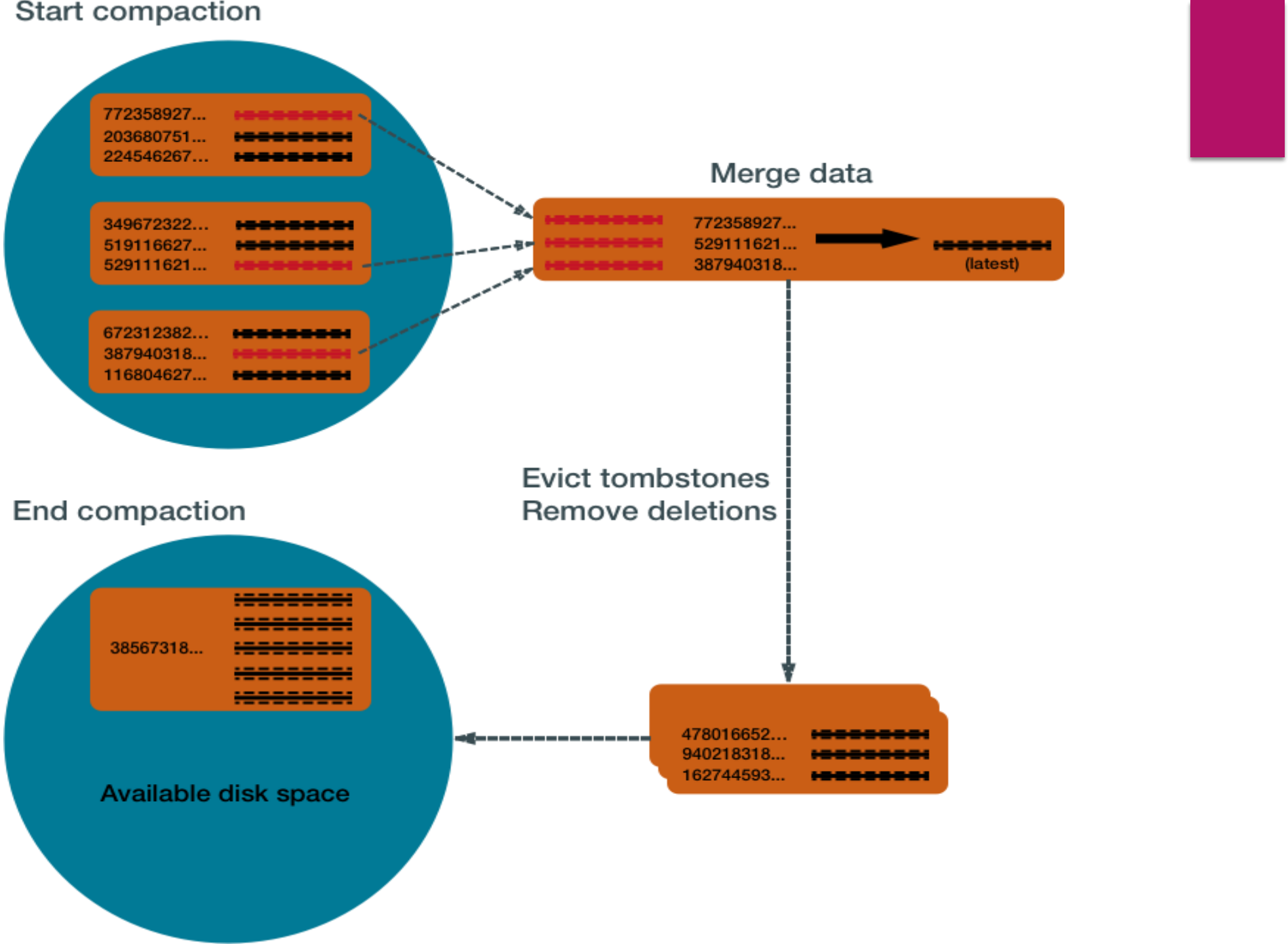
- (STCS) is to merge sstables of approximately the same size.
- All sstables are put in different buckets depending on their size.
- An sstable is added to the bucket if size of the sstable is within bucket_low and bucket_high of the current average size of the sstables already in the bucket.
- This will create several buckets and the most interesting of those buckets will be compacted.
- The most interesting one is decided by figuring out which bucket's sstables takes the most reads.

Leveled Compaction Strategy

- LeveledCompactionStrategy (LCS) is that all sstables are put into different levels where we guarantee that no overlapping sstables are in the same level.
- By overlapping we mean that the first/last token of a single sstable are never overlapping with other sstables.
- This means that for a SELECT we will only have to look for the partition key in a single sstable per level.
- Each level is 10x the size of the previous one and each sstable is 160MB by default. L0 is where sstables are streamed/flushed - no overlap guarantees are given here.
- When picking compaction candidates we have to make sure that the compaction does not create overlap in the target level.
- This is done by always including all overlapping sstables in the next level

Time Window CompactionStrategy

- TimeWindowCompactionStrategy (TWCS) is designed specifically for workloads where it's beneficial to have data on disk grouped by the timestamp of the data, a common goal when the workload is time-series in nature or when all data is written with a TTL.
- In an expiring/TTL workload, the contents of an entire SSTable likely expire at approximately the same time, allowing them to be dropped completely, and space reclaimed much more reliably than when using SizeTieredCompactionStrategy or LeveledCompactionStrategy.
- The basic concept is that TimeWindowCompactionStrategy will create 1 sstable per file for a given window, where a window is simply calculated as the combination of two primary options:
- compaction_window_unit (default: DAYS) ---- A Java TimeUnit (MINUTES, HOURS, or DAYS).
- compaction_window_size (default: 1) --The number of units that make up a window.


Start compaction
772358927...
203680751...
224546267...
349672322...
519116627...
529111621...
672312382...
387940318...
116804627...
Merge data
772358927...
529111621...
387940318...
(latest)
Evict tombstones
Remove deletions
End compaction
38567318...
Available disk space
478016652...
940218318...
162744593...

Deletion and Tombstones

- A node could be down or unreachable when data is deleted, that node could miss a delete.
- When that node comes back online later and a repair occurs, the node could "resurrect" the data that had been previously deleted by re-sharing it with other nodes.
- To prevent deleted data from being reintroduced, Cassandra uses a concept called a tombstone.
- A tombstone is a marker that is kept to indicate data that has been deleted.
- When you execute a delete operation, the data is not immediately deleted.
- Instead, it's treated as an update operation that places a tombstone on the value.
- A tombstone is similar to the idea of a "soft delete" from the relational world.
- Instead of actually executing a delete SQL statement, the application will issue an update statement that changes a value in a column called something like "deleted."
- Programmers sometimes do this to support audit trails.

Why Tombstones

- Tombstone represents the delete and causes all values which occurred before the tombstone to not appear in queries to the database.
- This approach is used instead of removing values because of the distributed nature of Cassandra.
- Deletes without tombstones
- Imagine a three node cluster which has the value [A] replicated to every node.:

[A], [A], [A]

- If one of the nodes fails and and our delete operation only removes existing values we can end up with a cluster that looks like:

[], [], [A]

- Then a repair operation would replace the value of [A] back onto the two nodes which are missing the value.:

[A], [A], [A]

- This would cause our data to be resurrected even though it had been deleted.

Deletes with Tombstones

- Starting again with a three node cluster which has the value [A] replicated to every node.:

[A], [A], [A]

- If instead of removing data we add a tombstone record, our single node failure situation will look like this.:

[A, Tombstone[A]], [A, Tombstone[A]], [A]

- Now when we issue a repair the Tombstone will be copied to the replica, rather than the deleted data being resurrected.:

[A, Tombstone[A]], [A, Tombstone[A]], [A, Tombstone[A]]

- Our repair operation will correctly put the state of the system to what we expect with the record [A] marked as deleted on all nodes.
- Will end up accruing Tombstones which will permanently accumulate disk space.
- To avoid keeping tombstones forever we have a parameter known as gc_grace_seconds for every table in Cassandra.

Deletion and Tombstones

- Tombstones are not kept forever, instead they are removed as part of compaction.
- There is a setting per table called gc_grace_seconds (Garbage Collection Grace Seconds) which represents the amount of time that nodes will wait to garbage collect (or compact) tombstones.
- By default, it's set to 864,000 seconds, the equivalent of 10 days.
- Cassandra keeps track of tombstone age, and once a tombstone is older than gc_grace_seconds, it will be garbage collected.
- The purpose of this delay is to give a node that is unavailable time to recover; if a node is down longer than this value, then it should be treated as failed and replaced.

gc_grace_seconds parameter and Tombstone Removal

- The table level gc_grace_seconds parameter controls how long Cassandra will retain tombstones through compaction events before finally removing them.
- Duration should directly reflect the amount of time a user expects to allow before recovering a failed node.
- After gc_grace_seconds has expired the tombstone may be removed (meaning there will no longer be any record that a certain piece of data was deleted), but as a tombstone can live in one sstable and the data it covers in another, a compaction must also include both sstable for a tombstone to be removed.
- More precisely, to be able to drop an actual tombstone the following needs to be true;
- The tombstone must be older than gc_grace_seconds

gc_grace_seconds parameter and Tombstone Removal

- If partition X contains the tombstone, the sstable containing the partition plus all sstables containing data older than the tombstone containing X must be included in the same compaction.
- Don't need to care if the partition is in an sstable if we can guarantee that all data in that sstable is newer than the tombstone.
- If the tombstone is older than the data it cannot shadow that data.
- If the option only_purge_repaired_tombstones is enabled, tombstones are only removed if the data has also been repaired.
- If a node remains down or disconnected for longer than gc_grace_seconds it's deleted data will be repaired back to the other nodes and re-appear in the cluster.
- This is basically the same as in the "Deletes without Tombstones". Note that tombstones will not be removed until a compaction event even if gc_grace_seconds has elapsed.
- The default value for gc_grace_seconds is 864000 which is equivalent to 10 days. This can be set when creating or altering a table using WITH gc_grace_seconds.

TTL

- Data in Cassandra can have an additional property called time to live - this is used to automatically drop data that has expired once the time is reached.
- Once the TTL has expired the data is converted to a tombstone which stays around for at least gc_grace_seconds.
- Note that if you mix data with TTL and data without TTL (or just different length of the TTL) Cassandra will have a hard time dropping the tombstones created since the partition might span many sstables and not all are compacted at once.

System keyspaces

- Information about the structure of the cluster communicated via gossip is stored in **system.local** and **system.peers**.
- These tables hold information about the local node and other nodes in the cluster, including IP addresses, locations by data center and rack, token ranges, CQL, and protocol versions.
- **system.transferred_ranges** and **system.available_ranges** track token ranges previously managed by each node and any ranges needing allocation.
- Construction of materialized views is tracked in the syst em.view_builds_in_progress and **system.built_views** tables, resulting in theviews available in **system_schema.views**.

System keyspaces

- •User-provided extensions include **system_schema.types** for user-defined **types,system_schema.triggers** for triggers configured per table, **system_schema.functions** for user-defined functions, and **system_schema.aggregates** for user defined aggregates.
- • The **system.paxos** table stores the status of transactions in progress, while the **system.batches** table stores the status of batches.
- • The **system.size_estimates** stores the estimated number of partitions per table and mean partition size.

System keyspaces

- **system_schema.keyspaces**, **system_schema.tables**, and **system_schema.columns** store the definitions of the keyspaces, tables, and indexes defined for the cluster.
- The **system_traces keyspace** contains tables that store information about query traces
- The **system_auth keyspace** contains tables that store information about the users,roles, and permissions

How is data updated?

- Cassandra treats each new row as an upsert: if the new row has the same primary key as that of an existing row, Cassandra processes it as an update to the existing row.
- During a write, Cassandra adds each new row to the database without checking on whether a duplicate record exists.
- This policy makes it possible that many versions of the same row may exist in the database.
- Periodically, the rows stored in memory are streamed to disk into structures called SSTables.
- At certain intervals, Cassandra compacts smaller SSTables into larger SSTables.
- If Cassandra encounters two or more versions of the same row during this process, Cassandra only writes the most recent version to the new SSTable.
- After compaction, Cassandra drops the original SSTables, deleting the outdated rows.

How is data updated?

- Most Cassandra installations store replicas of each row on two or more nodes.
- Each node performs compaction independently.
- This means that even though out-of-date versions of a row have been dropped from one node, they may still exist on another node.
- This is why Cassandra performs another round of comparisons during a read process.
- When a client requests data with a particular primary key, Cassandra retrieves many versions of the row from one or more replicas.
- The version with the most recent timestamp is the only one returned to the client ("last-write-wins").
- Note: Some database operations may only write partial updates of a row, so some versions of a row may include some columns, but not all.
- During a compaction or write, Cassandra assembles a complete version of each row from the partial updates, using the most recent version of each column.