Cassandra

What is Cassandra?

- Massively linearly scalable NoSQL database
  - Fully distributed, with no single point of failure
  - Free and open source, with deep developer support
  - Highly performant, with near-linear *horizontal* scaling in proper use cases
  - Fully peer-to-peer—no master/slave architecture
  - Data center aware

What are Keyspaces?

- A namespace for tables in a cluster
  - All data will reside in some keyspace
  - Main function: Control replication
  - Data with different replication requirements will be in different keyspaces
  - Somewhat analogous to a schema in a relational model
- Replication Factor (RF) and replication strategy specified when creating a keyspace
  - They may be changed later
- Below is the CQL (Cassandra Query Language) to create a keyspace
  - It uses SimpleStrategy and has an RF of 1
  - You can change the RF using the ALTER KEYSPACE command

```
CREATE  KEYSPACE stockwatcher WITH REPLICATION  =
{ ‘class’ : ‘SimpleStrategy’, replication_factor’: 1};
```

What are tables?

- Tables store data in a Cassandra database
  - Define columns and their metadata (e.g. Data type)
  - Analogous to a relational table
  - Were called Column Families in the Thrift API
- CREATE TABLE is used in CQL to define a table
  - A table must reside in a keyspace
  - Either selected with USE, or as part of the name of the table,e.g. Stockwatcher.user
  - A table must specify a primary key

```
USE stockwatcher;  //Execute anytime before the CREATE
CREATE TABLE user (username TEXT PRIMARY KEY,
                   userid TIMEUUID,
                   phonenumber BIGINT);
```

Inserting Data

- For this simple table, we can insert into a row using the INSERT examples below
  - This results in the rows being written to Cassandra
  - This should also seem familiar to SQL users
- All writes for a row (including inserts/updates/deletes) are done atomically and in isolation
  - Inserting or updating (multiple) columns in a row is one write operation
  - We’ll cover this in more detail shortly

```
INSERT INTO User (first_name, last_name, display_name)
   VALUES (‘Lebron’, ‘James’, ‘King James’);
INSERT INTO User (first_name, last_name, display_name)
   VALUES (‘Eldrick’, ‘Woods’, ‘Tiger Woods’);
INSERT INTO User (first_name, last_name, display_name)
   VALUES (‘Eli’, ‘Manning’, ‘Two-Time Super Bowl Winner’);
```

INSERT is Always UPSERT

- An insert with an existing primary key becomes an update
  - Cassandra will just write the new column value(s) provided
  - Each column inserted will supersede any older values
- For two concurrent writes with the same primary key, the last write wins
  - i.e. The last write to finish will be returned in subsequent queries
- Review the following inserts, and the results of the CQL

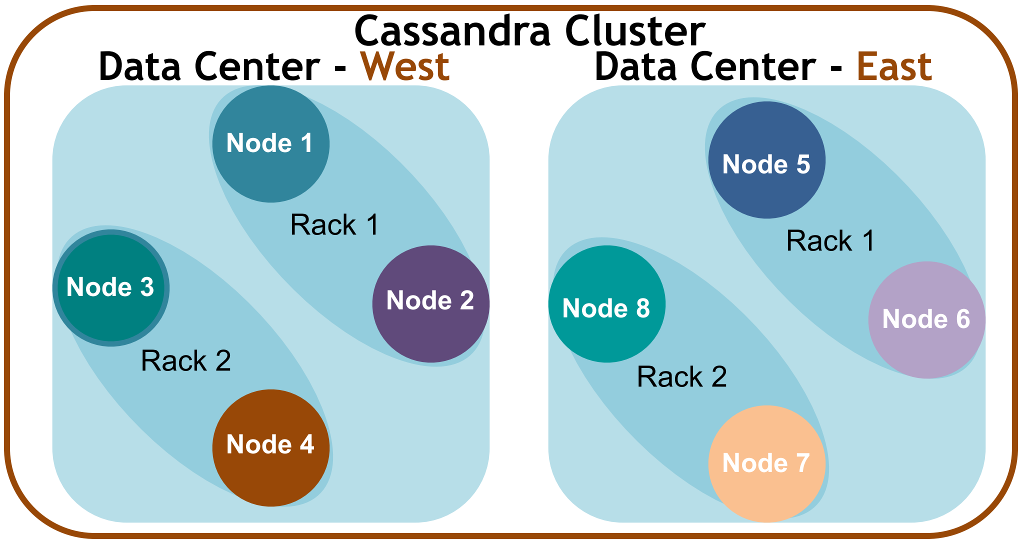
```
INSERT INTO User (first_name, last_name, display_name)
     VALUES (‘Lebron’, ‘James’, ‘Lebron James’);
INSERT INTO User (first_name, display_name)
     VALUES (‘L’, ‘Lebron James’);
```

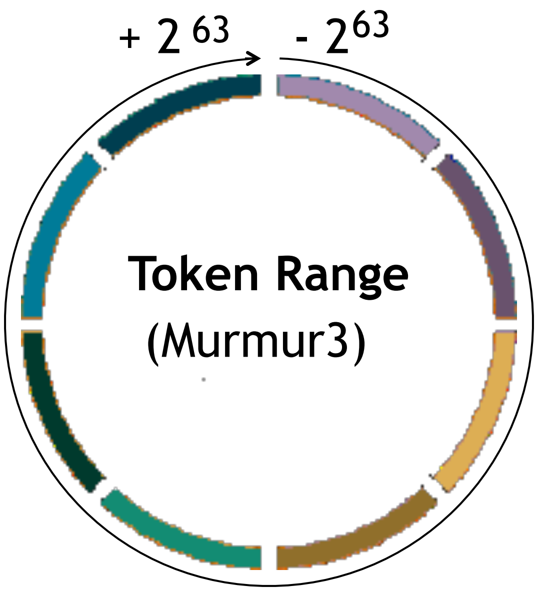
```
Cqlsh:stockwatcher> select * from user;

 display_name | first_name | last_name
--------------+------------+-----------
 Lebron James |          L |     James
```

What is a cluster?

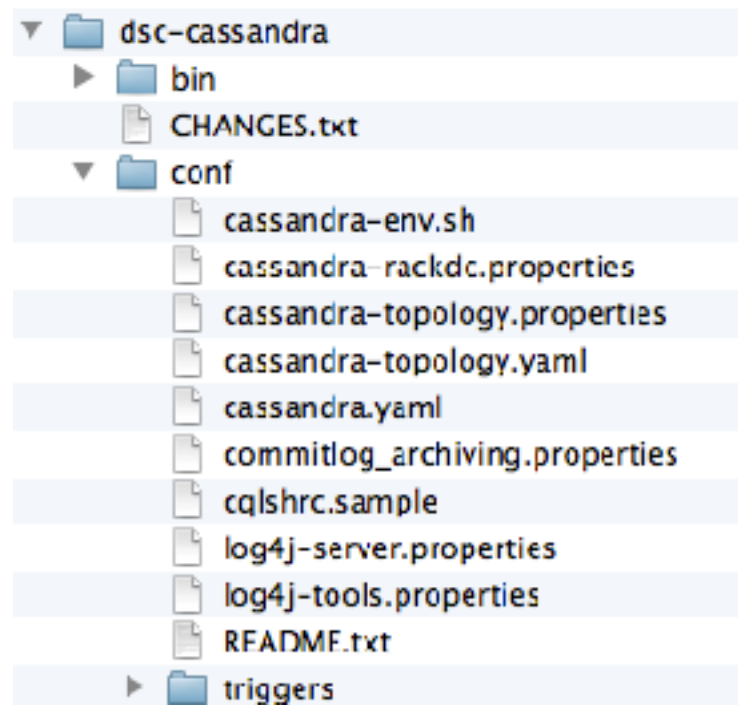
- A peer to peer set of nodes
  - Node – one Cassandra instance
  - Rack – a logical set of nodes
  - Data Center – a logical set of racks
  - Cluster – a ring of nodes





Configuration files

- cassandra.yaml
  - One file per node, must agree with other node’s files
  - Parameters defined throughout
- cassandra-env.sh
  - Memory settings
  - JMX settings
- log4j-server.properties
  - Error log settings



What key properties are set in *cassandra.yaml?*

- cluster_name (*default:'Test Cluster'*)
  - All nodes in a cluster must have the same value.
- listen_address (*default:localhost*)
  - Defines the network interface for gossip connections
- rpc_address
  - Network interface for client connections (0.0.0.0 means all interfaces)
- rpc_port (*default:9160*)
  - port for thrift client connections
- native_transport_port (*default:9042*)
  - port on which CQL native transport listens for clients

Key properties in *cassandra.yaml*

- commitlog_directory (*default*: /var/lib/cassandra/commitlog)
  - Best practice to mount on a separate disk in production (unless SSD)
- data_file_directories (*default*:/var/lib/cassandra/data)
  - List of storage directories for data tables SSTables)
- saved_caches_directory (*default*:/var/lib/cassandra/saved_caches)
  - Storage directory for key and row caches

What key properties are set in *cassandra-env.sh?*

- JVM Heap Size settings
  - MAX_HEAP_SIZE="*value*"
    - Maximum recommended i n production is currently 8G due to current limitations in Java garbage collection

System Memory	Heap Size
Less than 2GB	1/2 of system memory
2GB to 4GB	1GB
Greater than 4GB	1/4 system memory, but not more than 8GB

  - HEAP_NEWSIZE="value"
    - Generally set to ¼ of MAX_HEAP_SIZE
- This file computes the default values, but you can override them as necessary.

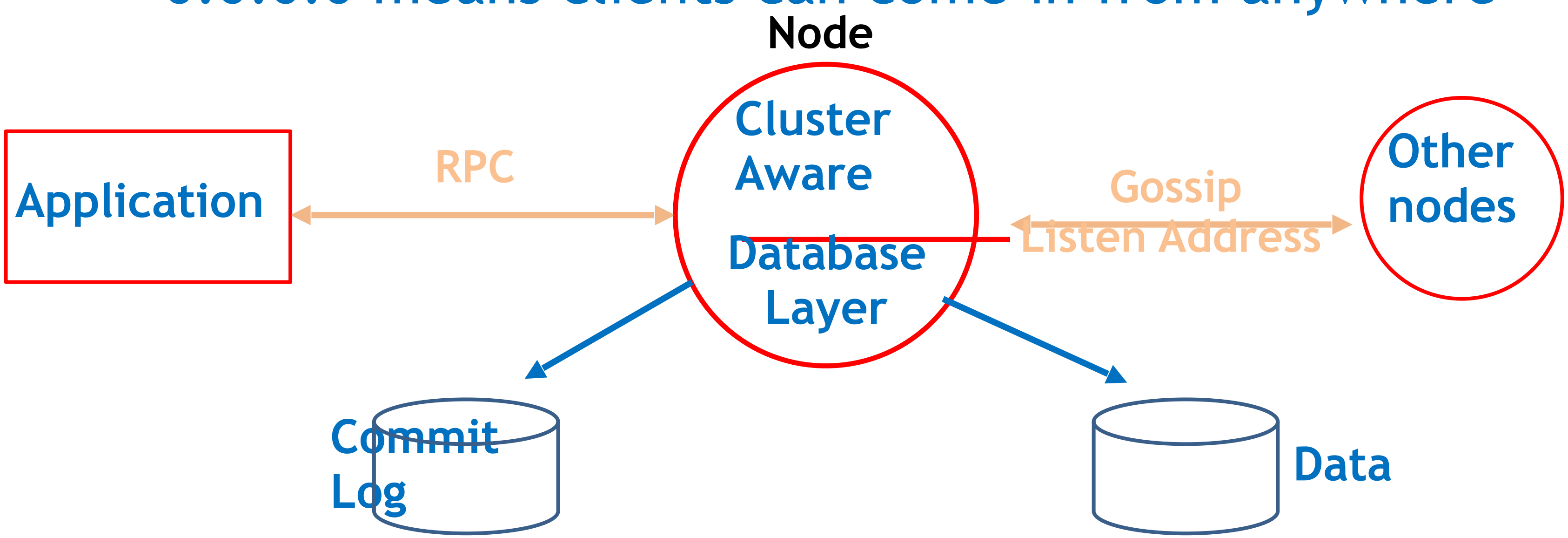
What key properties are set in *log4j-server.properties?*

- Cassandra system.log location
  - Default location is */var/log/cassandra/system.log*
  - system.log is numerically renamed as it grows over time.
- Cassandra logging level
  - Default logging level is *INFO*

```
# output messages into a rolling log file as well as stdout
log4j.rootLogger=INFO,stdout,R
```

What is a node?

- Single node database
- rpc_address—used to setup how clients come into a cluster
- rpc_port (9160)—thrift, how node talks to the application
- native_transport_port (9042)—native connections
- 0.0.0.0 means clients can come in from anywhere



What is a cluster?

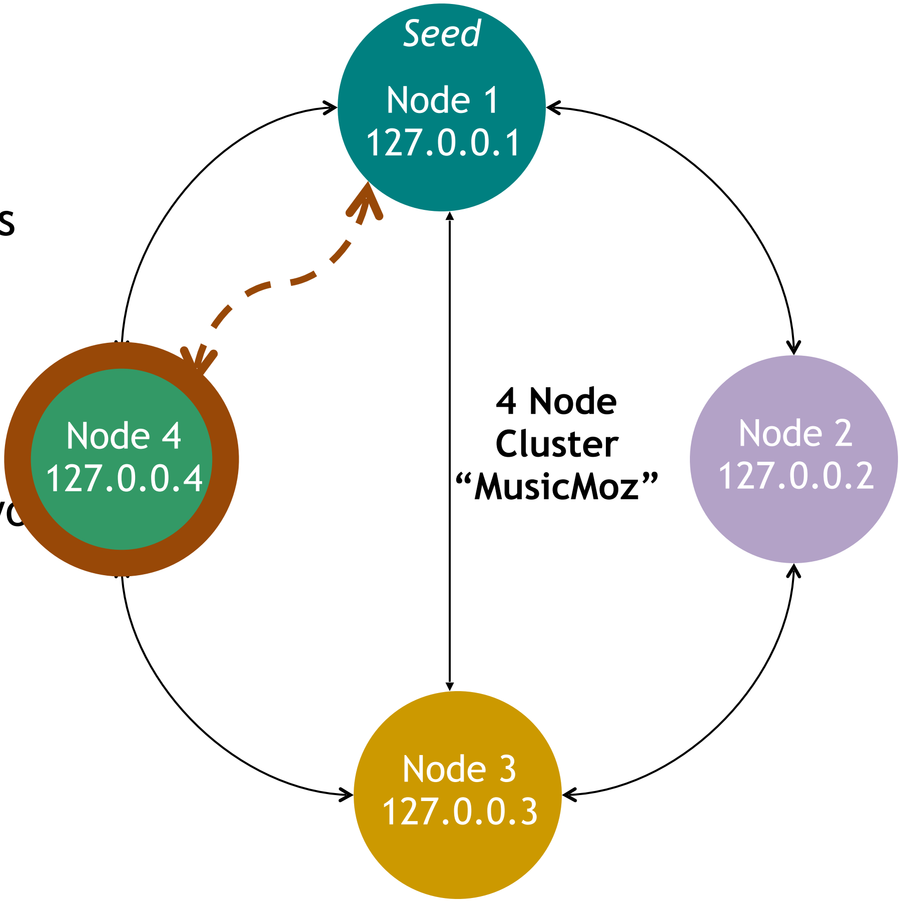
- Nodes join a cluster based on the configuration of their own *conf/cassandra.yaml* file

- Key settings include

  - cluster_name – shared name to logically distinguish a set of nodes

  - seeds – IP addresses of initial nodes for a new node to contact and discover the cluster topology (best practice to use the same two per data center)

  - listen_address – IP address to determine adaptor through which this particular node communicates to other nodes

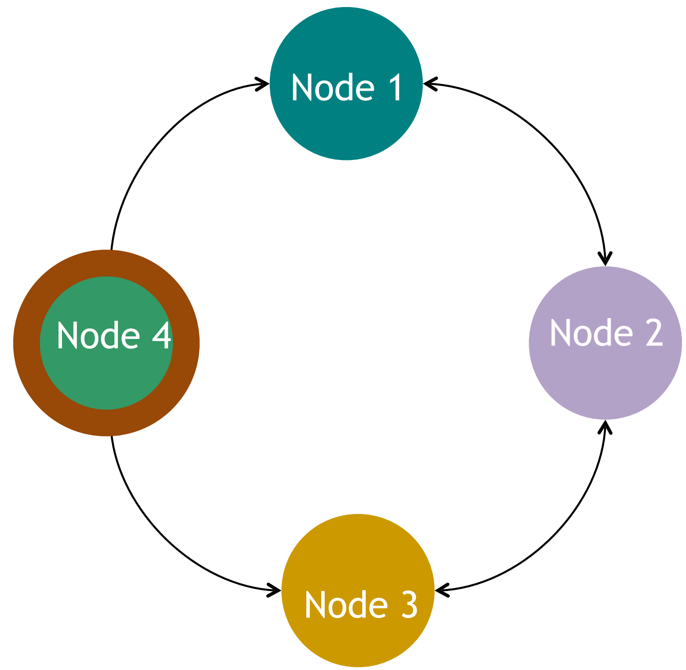


Where does my data go?

- Cassandra automatically shards your data.
- It will put one or more copies of your data on your nodes.

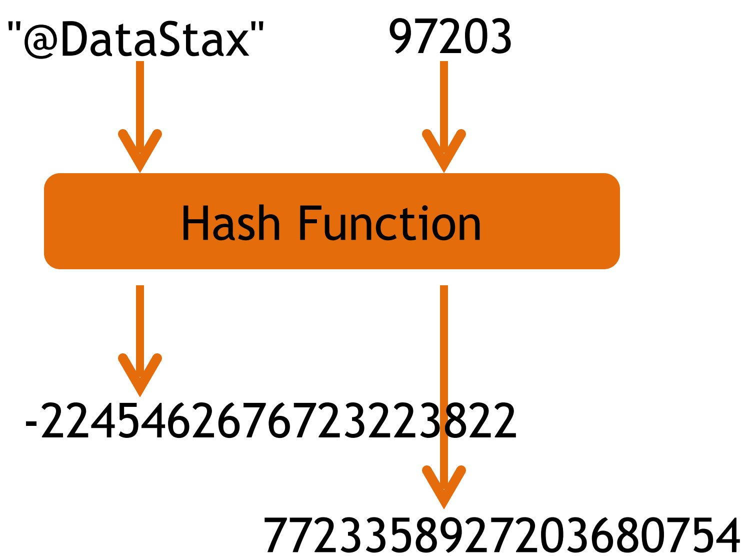
What is consistent hashing?

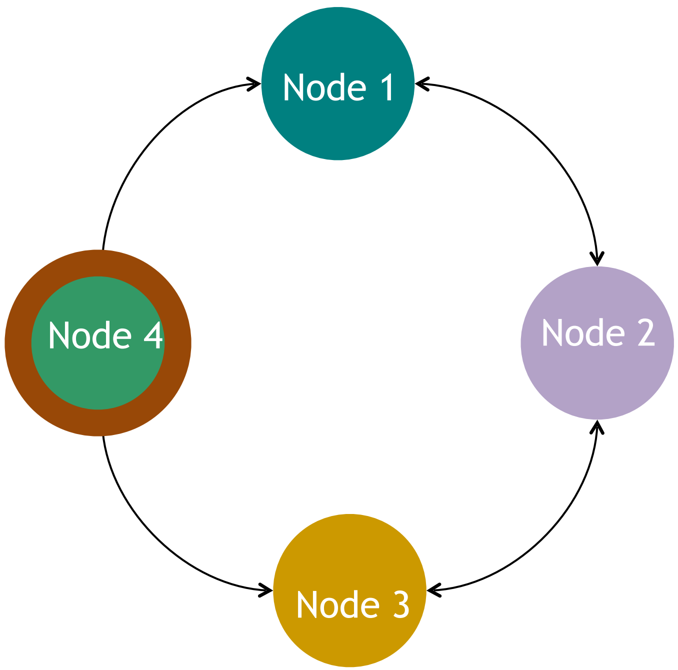
- Data is store in *nodes* in *partitions*, each identified by a *partition key.*
  - **Partition** - a storage location on a node (analogous to a "table row")
  - **Token** – 64 bit integer, generated by a hashing algorithm, identifying a partition's location within a cluster
- The 264 value *token range for a* cluster is used as a single ring
  - So, any partition in a cluster is locatable from one *consistent set of* hash values, regardless of its node
  - Specific token range varies by choice of *partitioner*



What is the partitioner?

- A system on each node which has hashes keys to create a token from designated values in rows being added
  - **Hash function** – converts a variable length value to a corresponding fixed length value

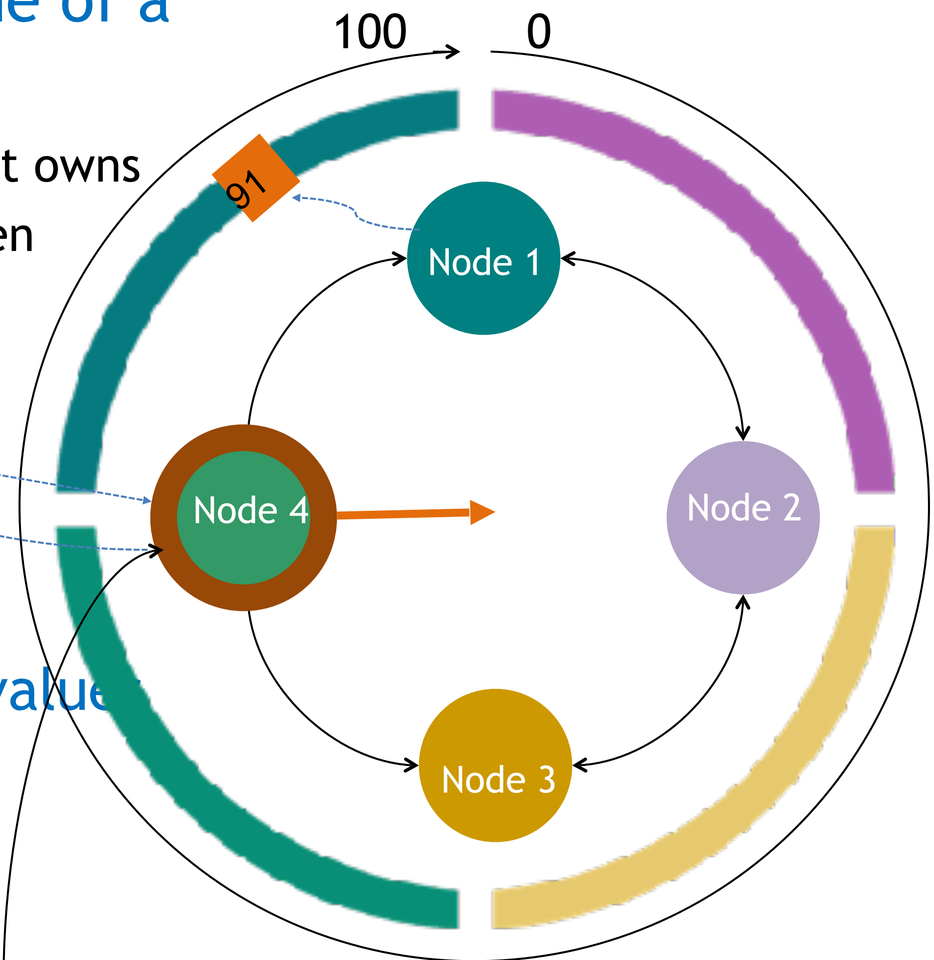




- Various partitioners available

How does a partitioner work?

- A node’s *partitioner* hashes a token from the *partition key* value of a write request
  - <u>First</u> replica written to node that owns the primary range for this token



- The *primary key* of a table determines its *partition key* value

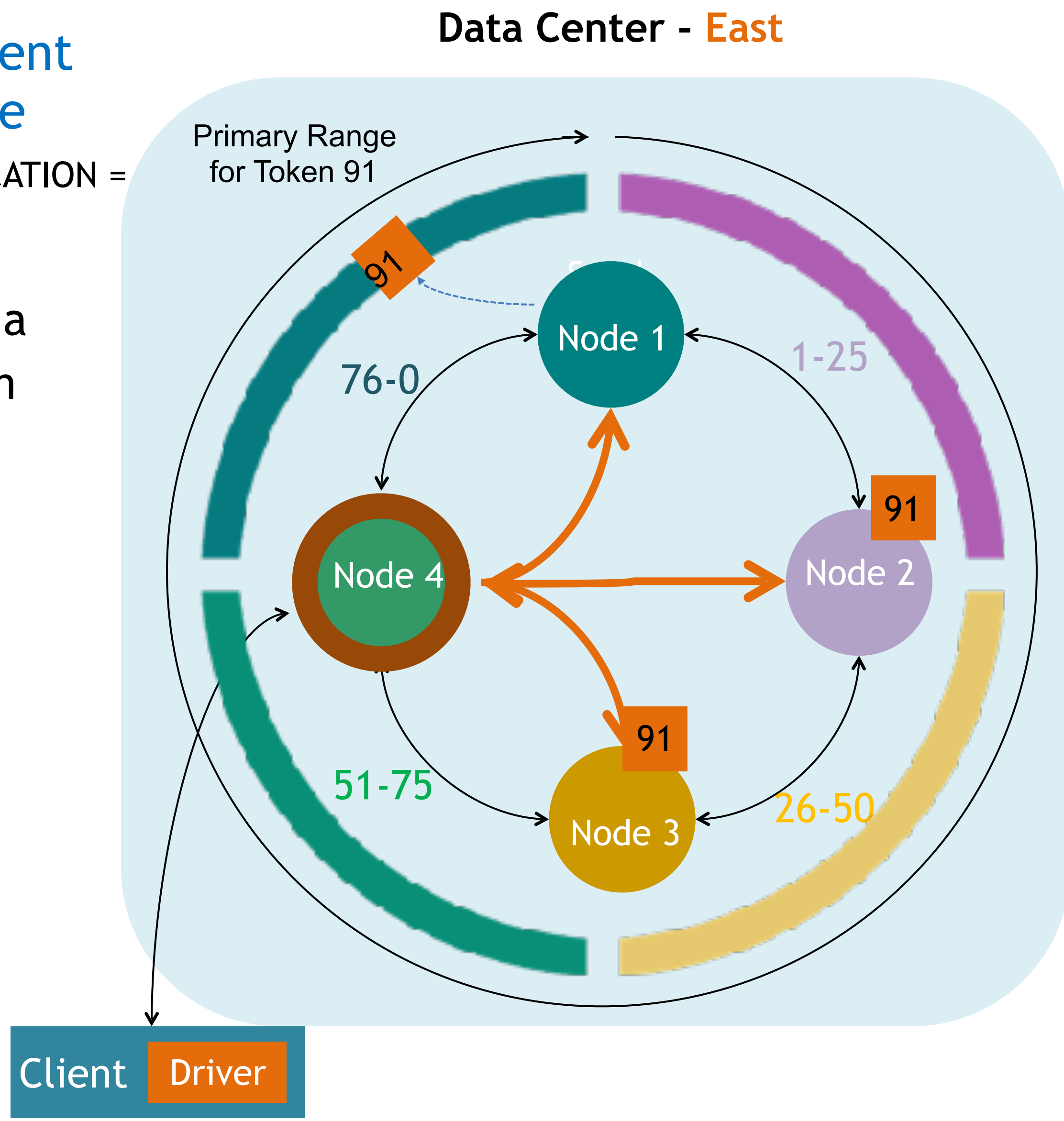
CREATE TABLE Users (
Firstname text, lastname text, level text,
PRIMARY KEY ((lastname, firstname))
);
INSERT INTO Users (firstname, lastname, level)
VALUES (‘Oscar’, Orange’, 42);

How is data replicated among nodes?

- SimpleStrategy – create replicas on nodes subsequent to the *primary range* node

  CREATE KEYSPACE demo WITH REPLICATION = {‘class’:’**SimpleStrategy**’, ‘replication_factor’:**3**}

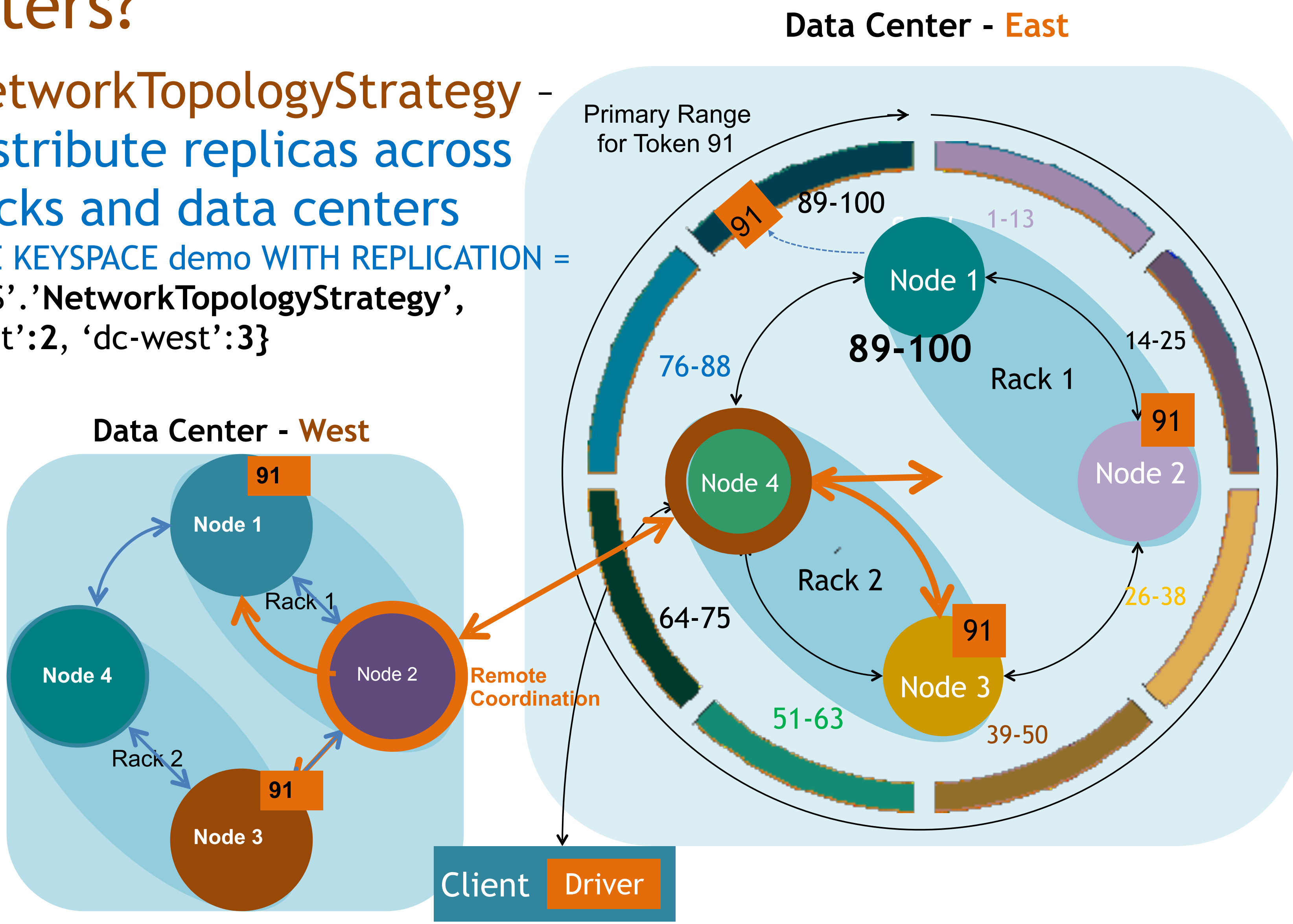
  - *replication factor* of 3 is a recommended minimum



How is data replicated between data centers?

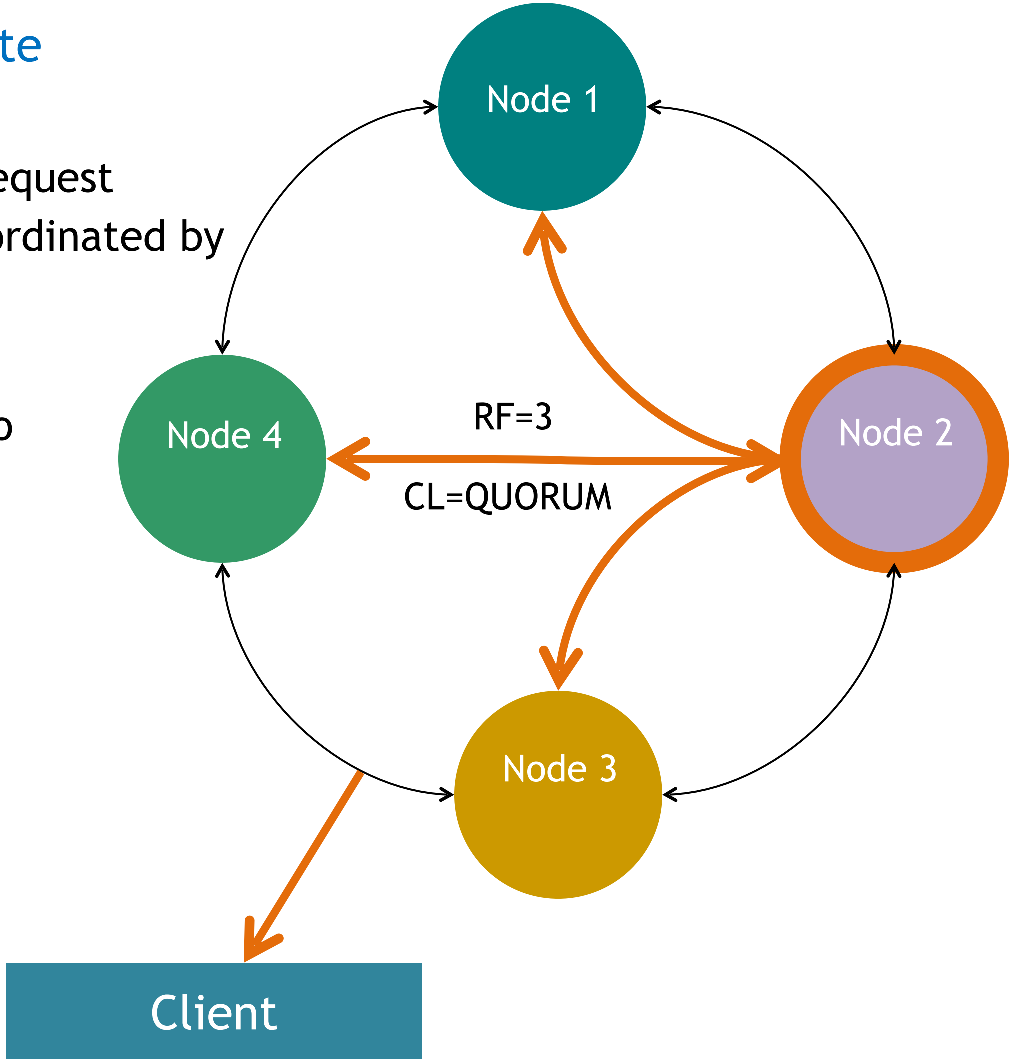
- NetworkTopologyStrategy – distribute replicas across racks and data centers

CREATE KEYSPACE demo WITH REPLICATION = {‘CLASS’.’**NetworkTopologyStrategy**’, ‘dc-east’:**2**, ‘dc-west’:**3**}



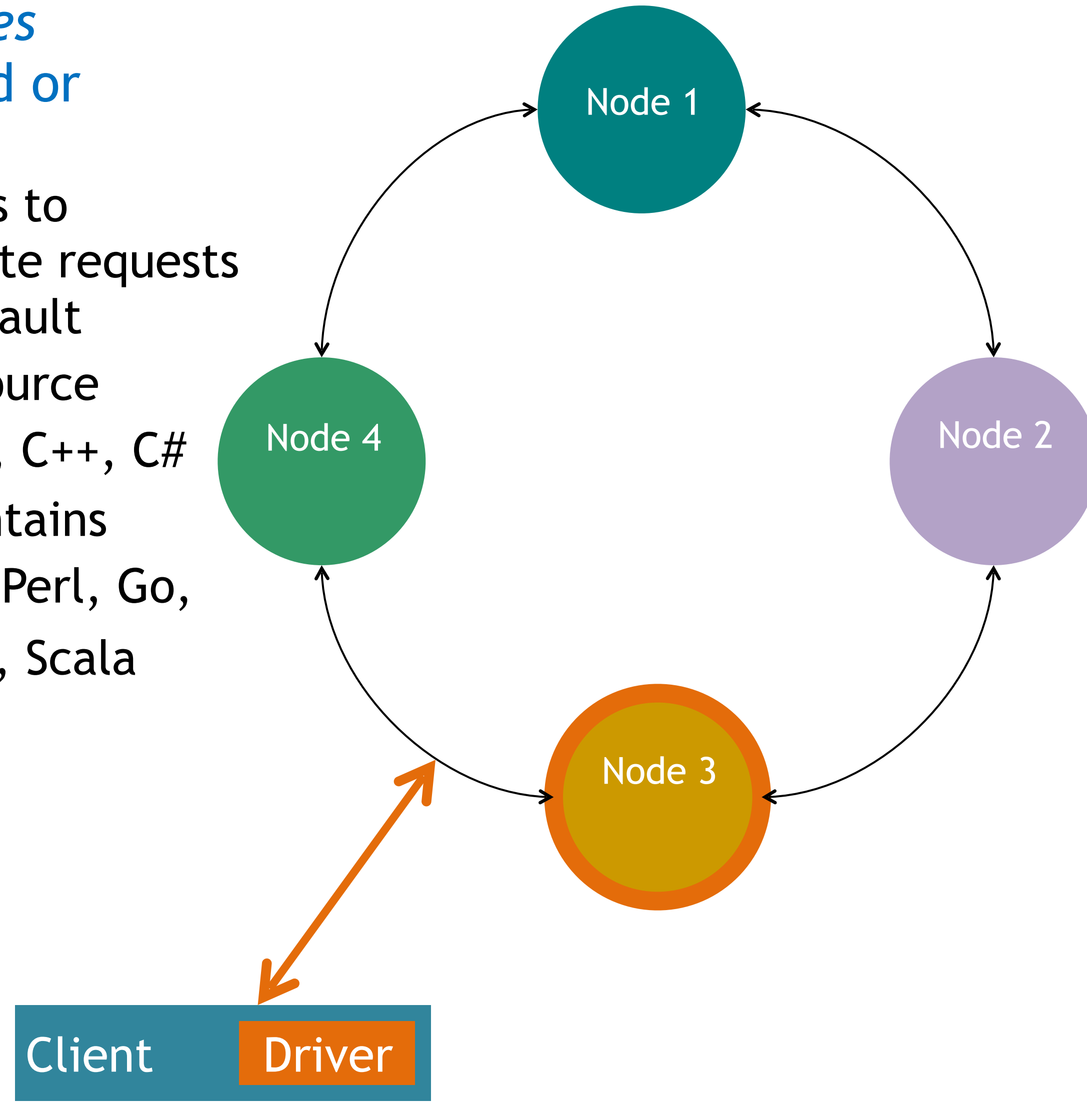
What is a coordinator?

- The *node* chosen by the client to receive a particular read or write request to its *cluster*
  - Any node can coordinate any request
  - Each client request may be coordinated by a different node
- No single point of failure
  - This principle is fundamental to Cassandra's architecture



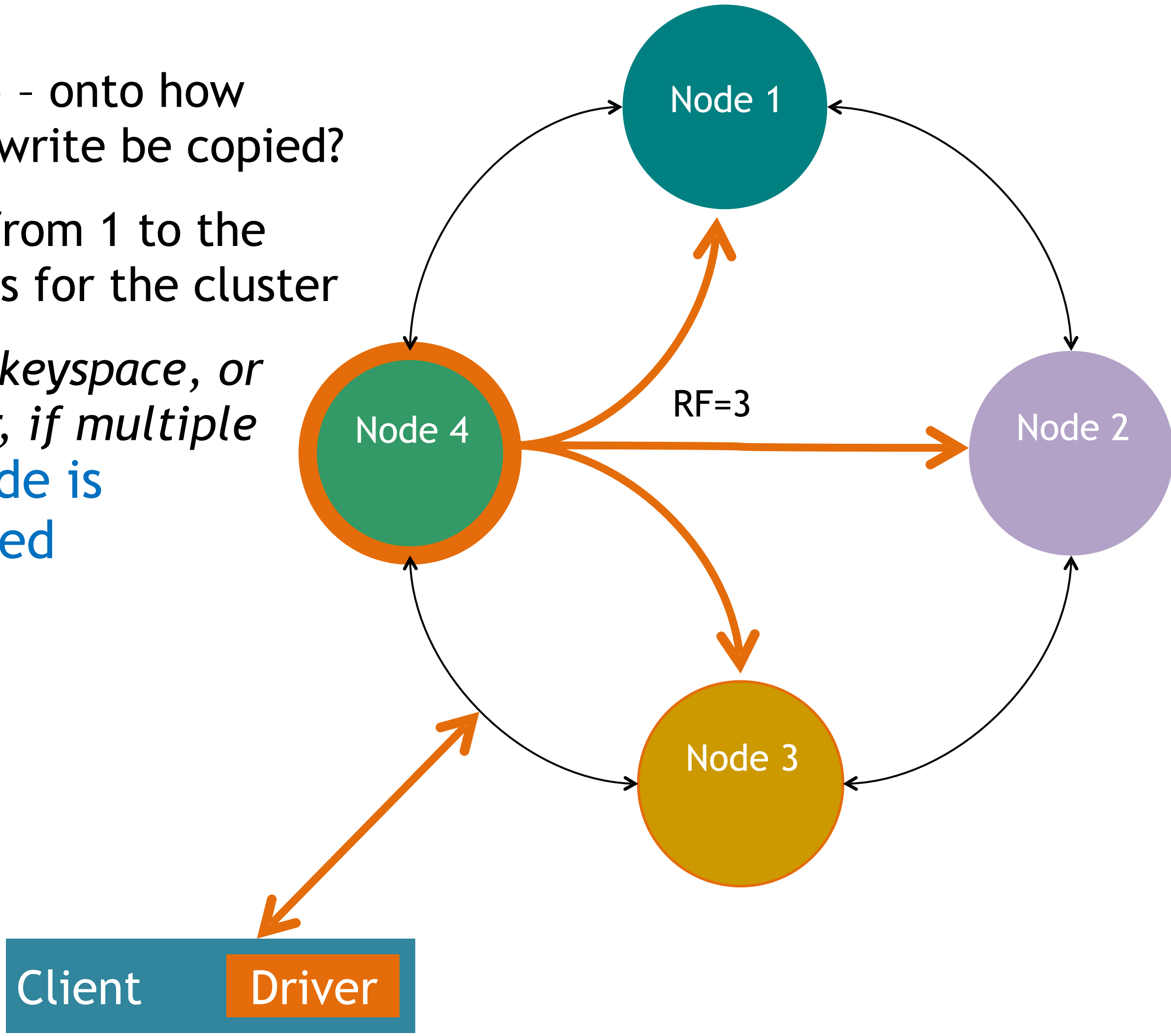
How are client requests coordinated?

- The *Cassandra driver chooses* the node to which each read or write request is sent
  - Client library providing APIs to manage client read/write requests
  - Round-robin pattern by default
  - DataStax maintains open source drivers for Java, Python, C++, C#
  - Cassandra Community maintains drivers for Node.js, PHP, Perl, Go, Clojure, Haskell, R, Ruby, Scala



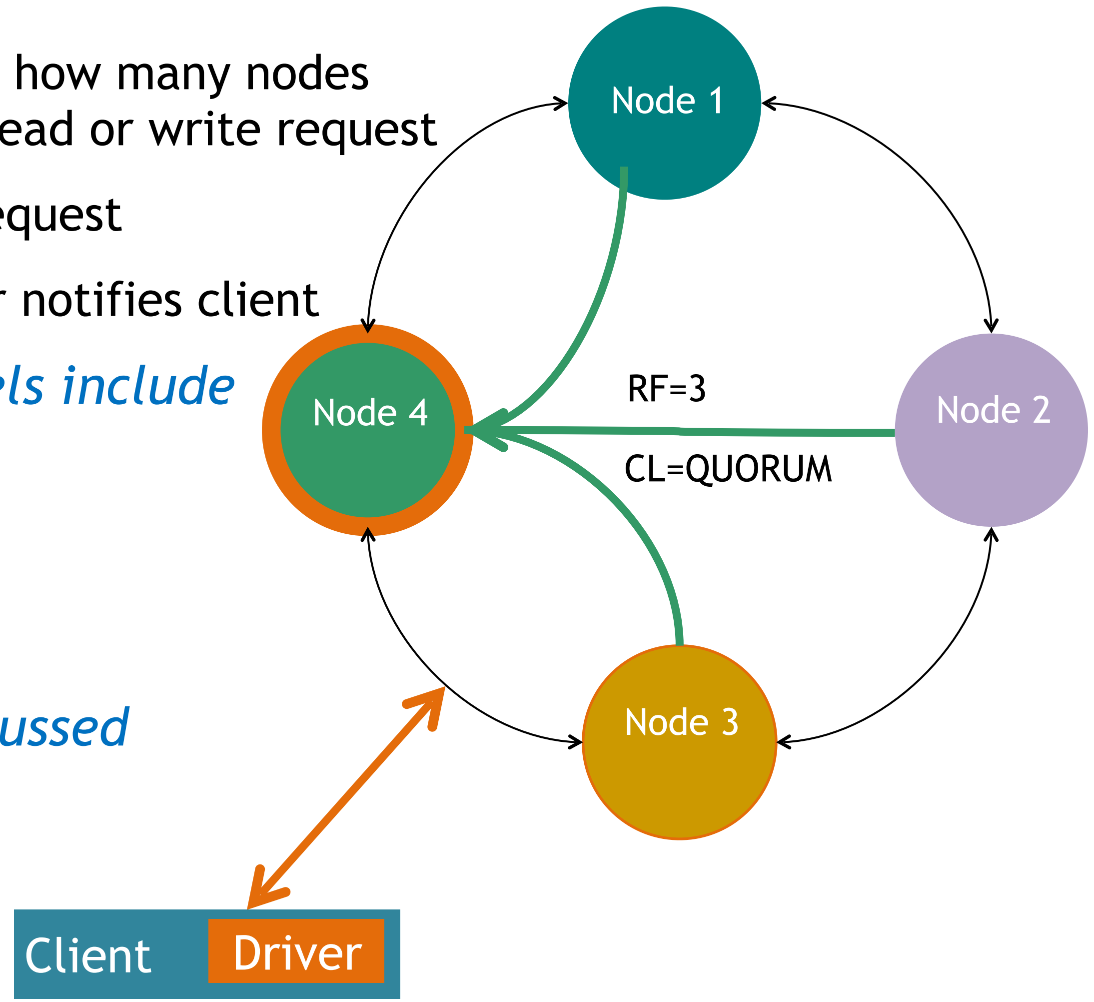
How are client requests coordinated?

- The coordinator manages the Replication Factor (RF)
  - Replication factor (RF) – onto how many nodes should a write be copied?
  - Possible values range from 1 to the total of planned nodes for the cluster
  - RF is set for an entire *keyspace*, or for each *data center, if multiple*
- Every write to every node is individually time-stamped



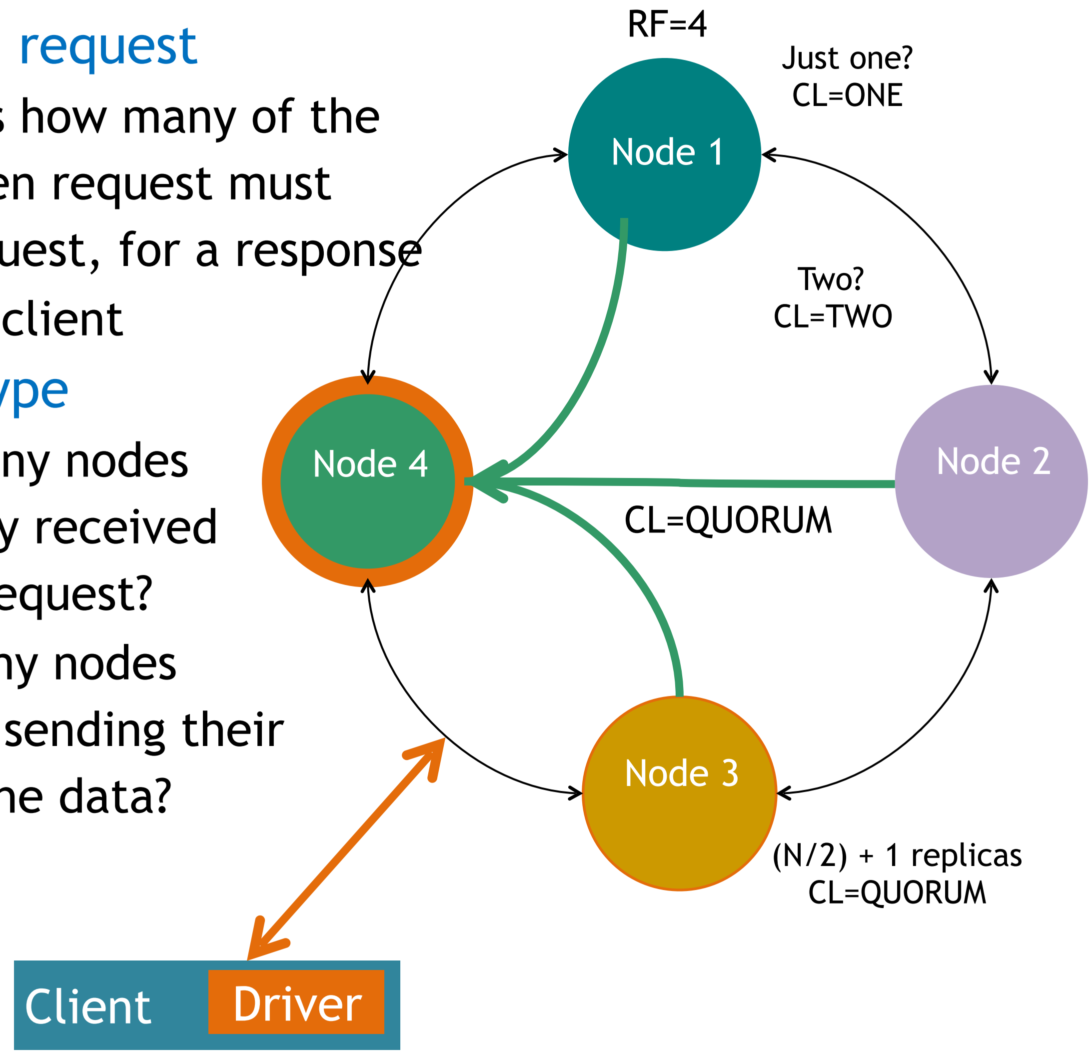
How are client requests coordinated?

- The coordinator also applies the Consistency Level (CL)
  - Consistency level (CL) – how many nodes must acknowledge a read or write request
  - CL may vary for each request
  - On success, coordinator notifies client
- Possible *consistency levels include*
  - ONE
  - QUORUM (RF / 2) + 1
  - ALL
- *Consistency Level is discussed further ahead*



What is consistency?

- The *partition key determines which* nodes are sent any given request
  - Consistency Level – sets how many of the nodes to be sent a given request must acknowledge that request, for a response to be returned to the client
- The meaning varies by type
  - Write request – how many nodes must acknowledge they received and wrote the write request?
  - Read request – how many nodes must acknowledge by sending their most recent copy of the data?

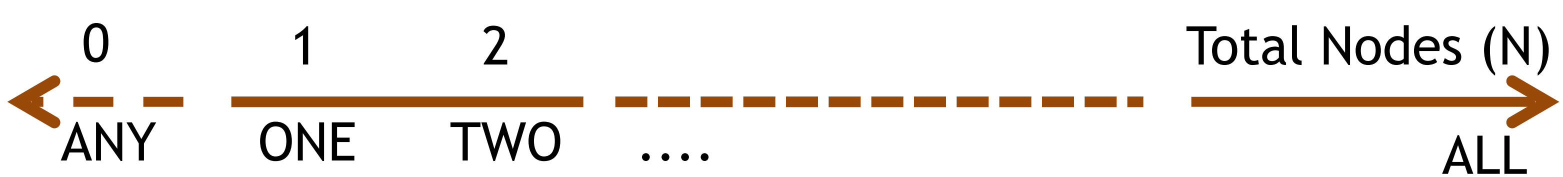


What consistency levels are available?

Name	Description	Usage
ANY (writes only)	Write to any node, and store hinted handoff if all nodes are down	Highest availability and
ALL	Check all nodes. Fail if any is down.	Highest consistency and
ONE (TWO, THREE)	Check closest node to coordinator	Highest availability and
QUORUM	Check quorum of available nodes.	Balanced consistency and availability
LOCAL_ONE	Check closest node to coordinator, in the local data center only.	Highest availability, lowest consistency, and no cross-data-center traffic
LOCAL_QUORUM	Check quorum of available nodes, in the local data center only.	Balanced consistency and availability, with no cross-data-center traffic
EACH_QUORUM	Only valid for writes. Check quorum of available nodes, in each data center of the cluster	Balanced consistency and availability, with cross-data-center consistency
SERIAL	Conditional write to quorum of nodes. Read current state with no change.	Used to support linearizable consistency for lightweight transactions
LOCAL_SERIAL	Conditional write to quorum of nodes in local data center.	Used to support linearizable consistency for lightweight transactions

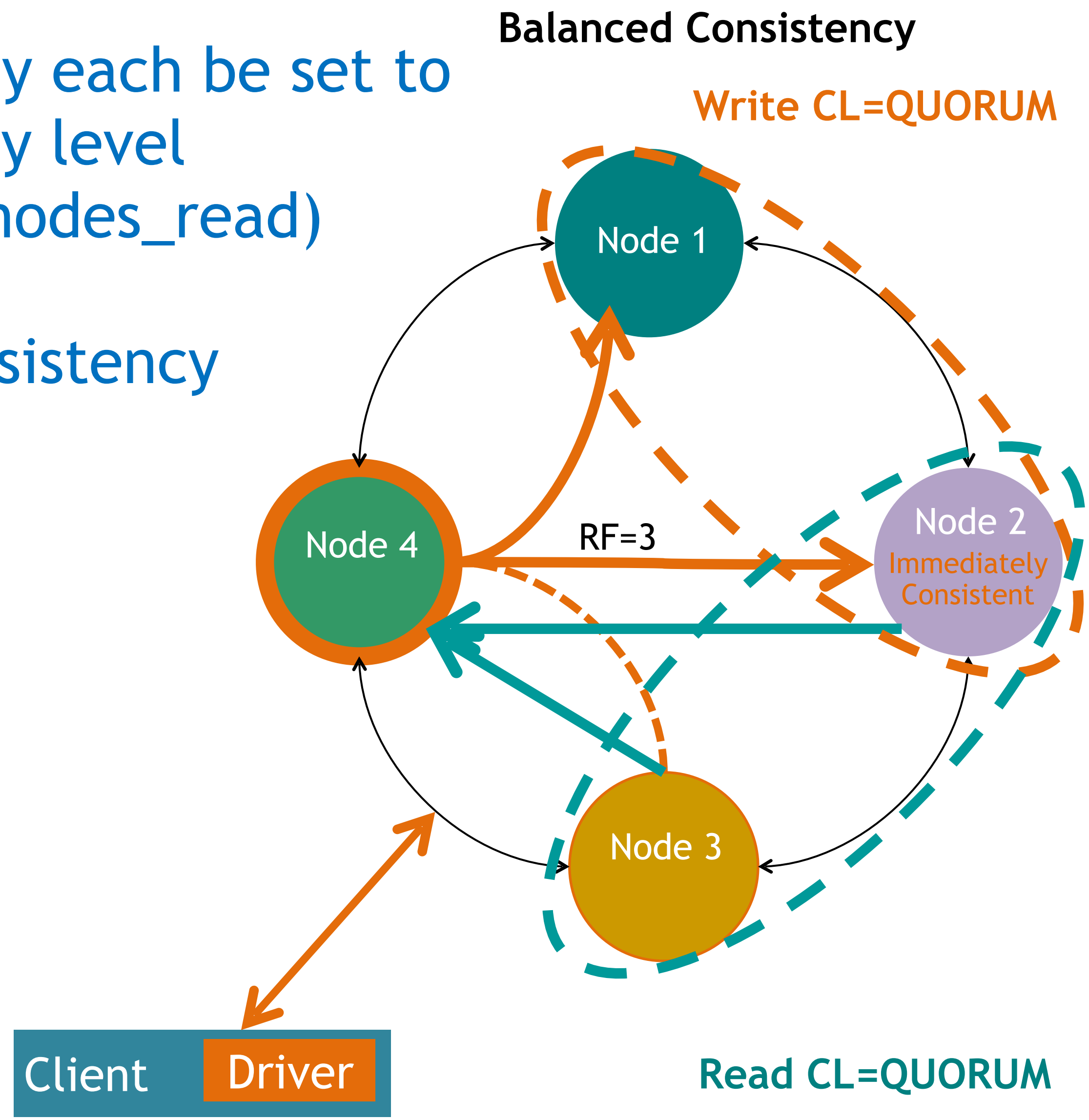
What is immediate vs. eventual consistency?

- For any given read, how likely is it the data may be stale?
- Immediate Consistency – reads always return the most recent data
    - Consistency Level ALL guarantees immediate consistency, because all replica nodes are checked and compared before a result is returned
    - *Highest latency because all replicas are checked and compared*
- Eventual Consistency – reads may return stale data
    - Consistency Level ONE carries the highest risk of stale data, because the replica from the first node to respond is immediately returned
    - *Lowest latency because the first replica is immediately returned*



Tuning consistency?

- Reads and writes may each be set to a specific consistency level
- **if** (nodes_written + nodes_read) > replication_factor **then** immediate consistency



How do you choose a consistency level?

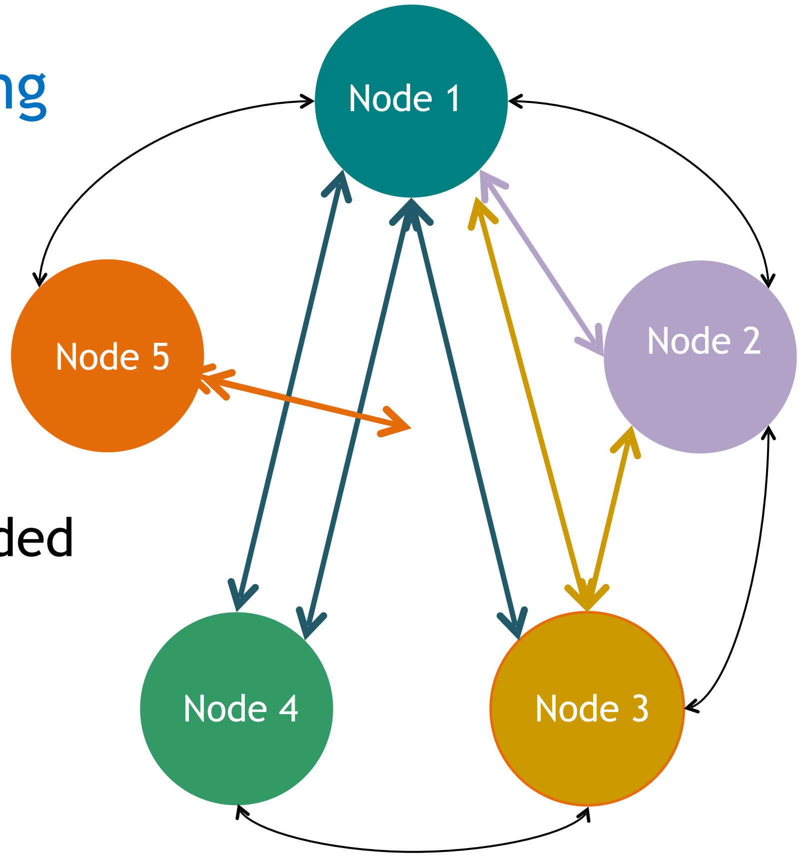
- In any given scenario, is the value of immediate consistency worth the latency cost?
  - Netflix uses CL ONE and measures its "eventual" consistency in milliseconds
  - Consistency Level ONE is your friend …

Consistency Level ONE	Consistency Level QUORUM	Consistency Level ALL
Lowest latency	Higher latency (than ONE)	Highest latency
Highest throughput	Lower throughput	Lowest throughput
Highest availability	Higher availability (than ALL)	Lowest availability
Stale read possible	No stale reads (if read and write at quorum)	No stale reads (if either read or write at ALL)

- If "stale" is measured in milliseconds, how much are those milliseconds worth?

What is the Gossip protocol?

- Once per second, each node contacts 1 to 3 others, requesting and sharing updates about
  - Known node states ("heartbeats")
  - Known node locations
  - Requests and acknowledgments are timestamped, so information is continually updated and discarded



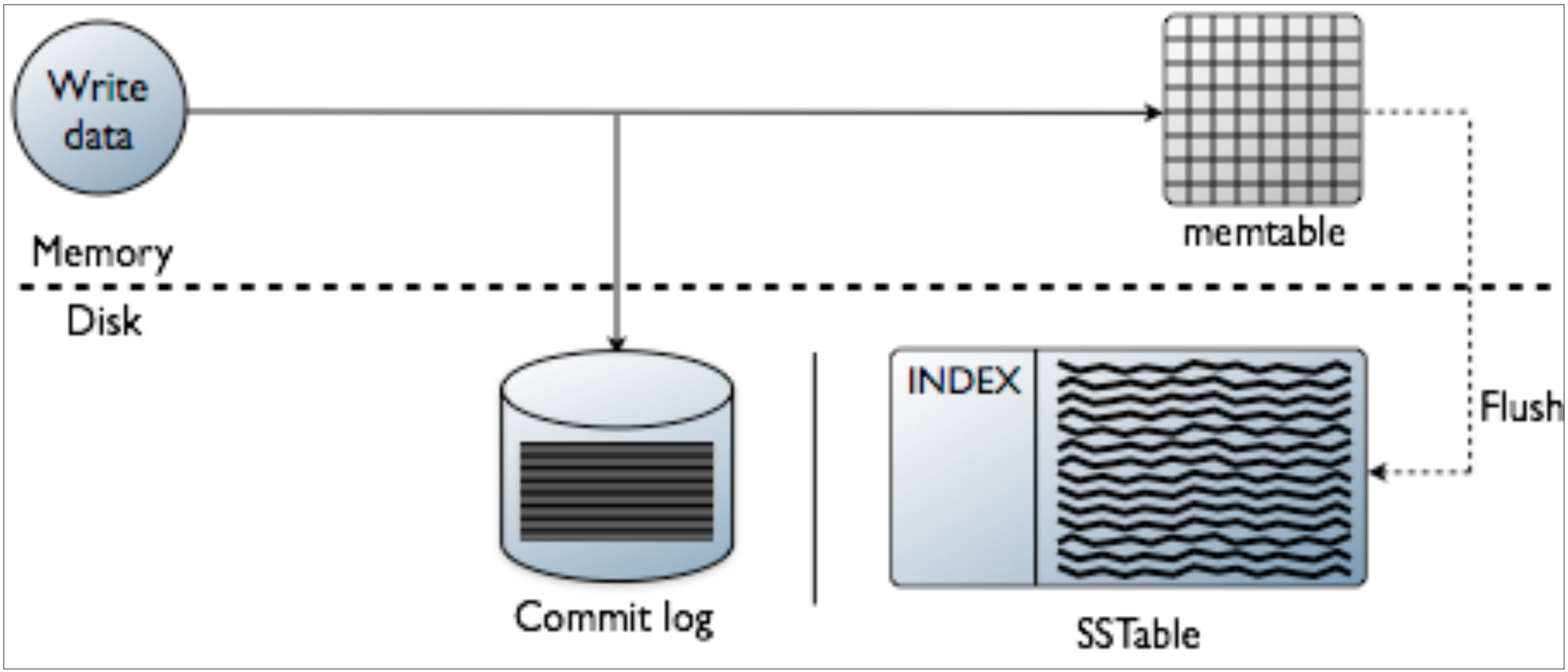
The write path

- On a write, Cassandra first appends writes to the commit log on disk
  - The write is durable once the data is in the commit log
    - Actually - once fsync is called and the OS flushes its own cache to disk
  - The commit log is append only, so there is no seek necessary for the append
    - Assuming a dedicated disk for the commit log (a recommended practice)
- A write also stores the data in memory
  - In a structure called the memtable
- A write is successful once written to the commit log and memory
- Very fast - Very little disk I/O at time of write

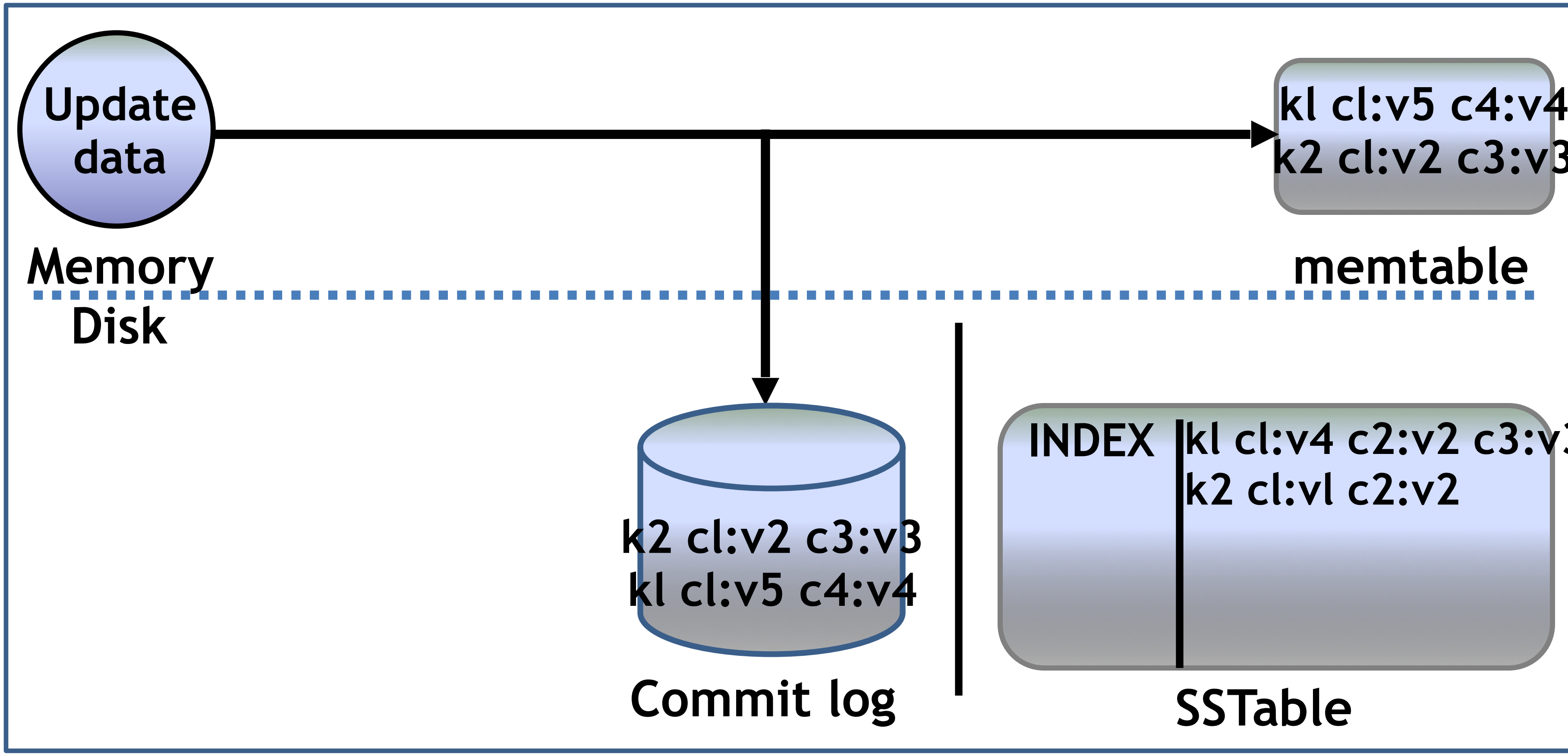
Memtables and SSTables

- Memtables are organized in sorted order by partition key
  - There is one memtable per table per node
  - Updates to column values in the memtable overwrites the existing column values
  - Accessing them is very fast since they are in-memory
  - Updates are also merged in-memory for a given partition key
- Memtables are eventually flushed to SSTables (Sorted String Tables) on disk
- So they don't grow too large in memory
  - Flushed using sequential I/O - no random seeking so it's fast
  - SSTables are immutable once they are written to disk
- Updates to data already in an SSTable go into a memtable, then eventually into a different SSTable
  - So data for a given partition key may be in several places

The write path illustrated



Memtable flushing illustrated

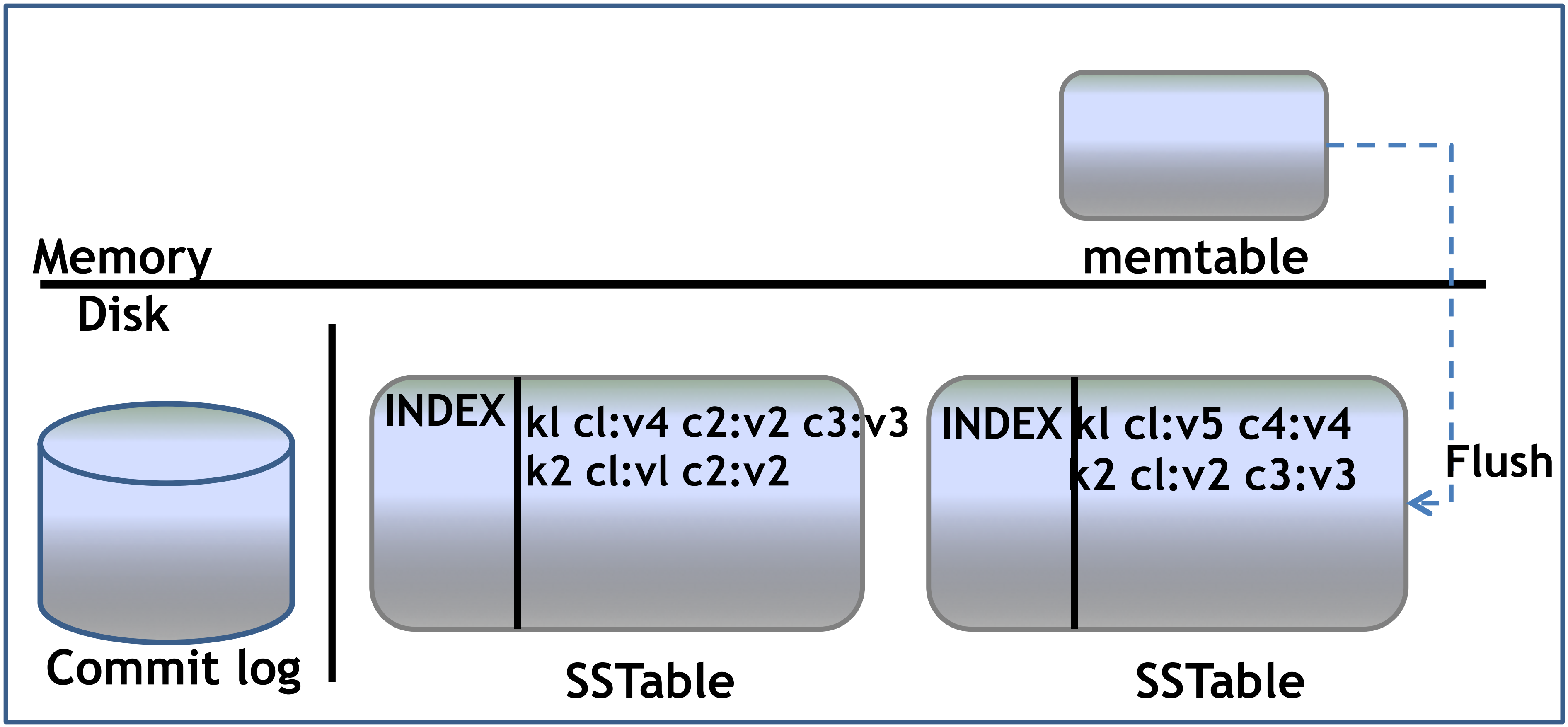


- After flushing, the memtable is emptied.

About SSTables

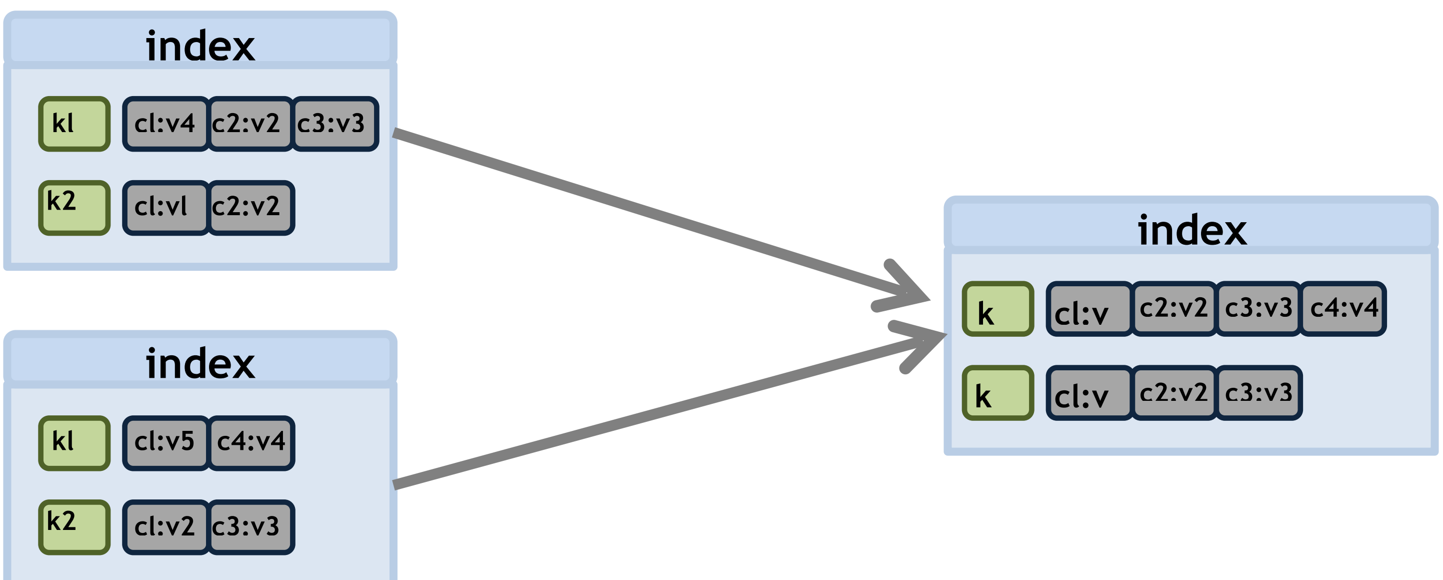
- SSTable is immutable once it is written
  - Mutations to keys already in an SSTable eventually end up in another SSTable
  - Never any updates to existing data in SSTable
  - Meaning no disk seeks on write - speeds up writes
- Reads may need to go to multiple SSTables for a given key
  - Because multiple writes may have created fragments in multiple SSTables for a given key
- SSTables contain structures to speed up reads
  - Bloom filters and indexes

SSTables illustrated



Compaction

- Compaction: Merges SSTables for a data table into one SSTable - eliminating fragments
  - Reduces number of SSTables to be accessed for a read request
  - Runs asynchronously in the background
  - Uses sequential I/O – fast
  - When merging multiple column values, latest timestamped value is used

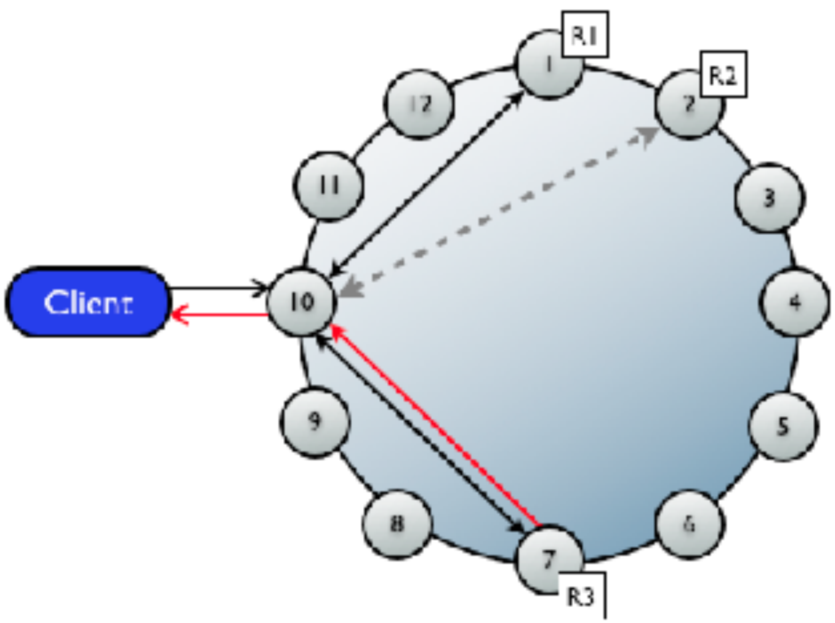


Read overview

- Cassandra must access multiple locations to read data
- Across replicas
  - Data can be replicated across multiple replicas
  - Replicas may not be consistent at any given time (eventual consistency)
- Within a replica, data may be
  - In an unflushed memtable
  - In multiple SSTables
  - In a cache

Client read requests

- Client requests are made to a coordinator
  - The coordinator contacts replicas based on the request CL (1)
- In the request below, we have RF=3
  - Assume CL=QUORUM - so the coordinator contacts two replicas - in this case R1 and R3



Merging replica data

- The coordinator may read data from multiple replicas
  - The data may not be consistent (a write may not have propagated)

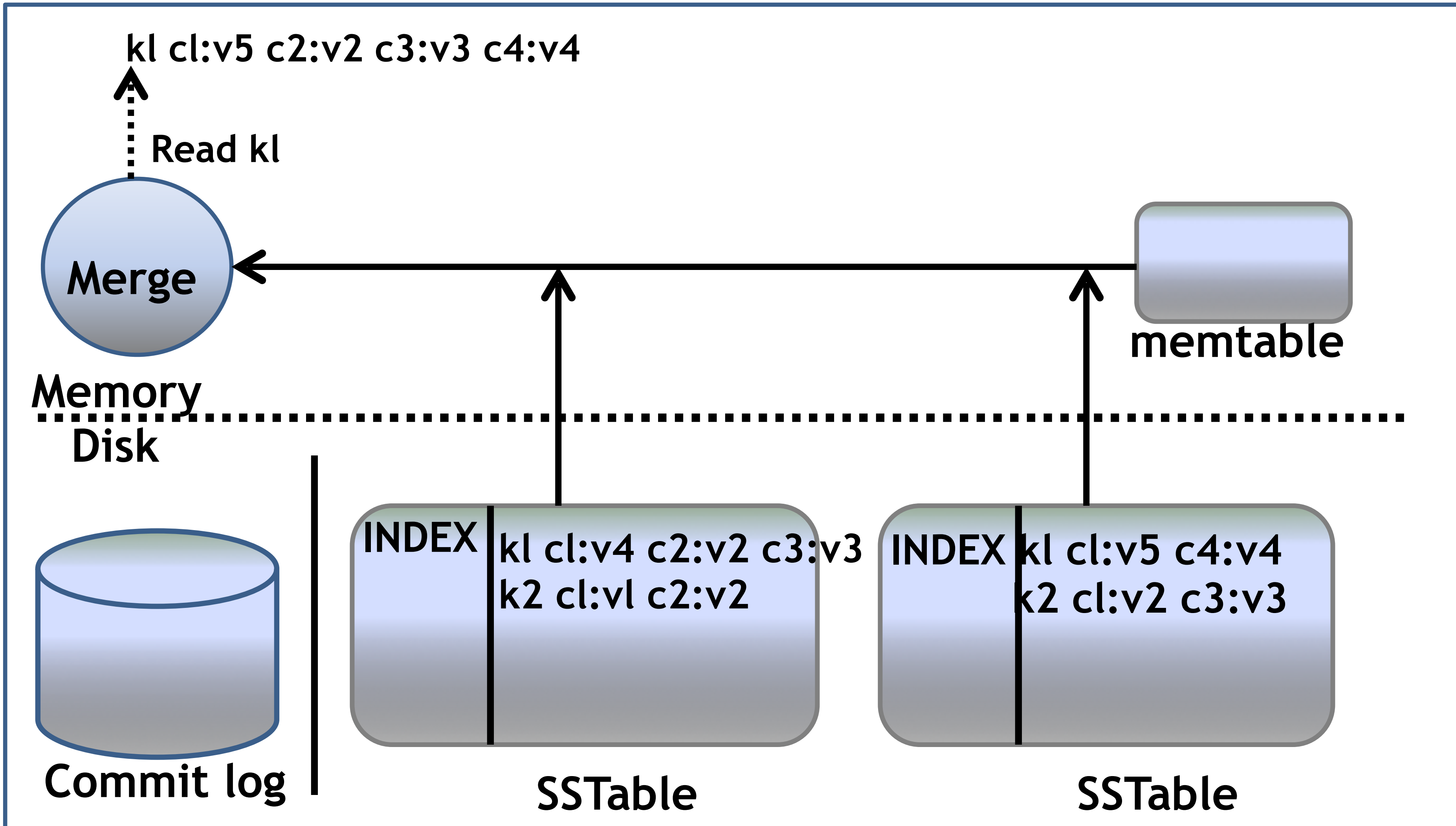
- If multiple replicas are contacted, the rows from each replica are compared for consistency
  - If consistent, then the data is just returned
  - If not consistent, then the most recent data is used
    - Based on the timestamp value that is contained in the internal storage cell

- Cassandra uses a mechanism, **read repair**, to ensure that all replicas are updated to the latest version of data

Read processing in a node

- To satisfy a read request for a given partition key, a node combines data from
  - Any unflushed memtables
  - All SSTables on the node that contain data for that partition key
- For a write, C* only stores the column values that are updated

Read processing in a node illustrated



Optimizing reads—bloom filters

- Cassandra uses Bloom filters to minimize SSTable reads
  - Each SSTable read is a disk I/O, so we want to minimize them

- Bloom filters are used to check if an SSTable has data for a particular partition key
  - One per SSTable
  - Saved on disk, but kept in memory (off heap)
  - On a read, the node checks the Bloom filter for each SSTable
  - The SSTable is only read from disk if the Bloom filter indicates there is data for the key
  - This helps make Cassandra very performant on reads

Full read path

- Row Cache (off heap): If found here, just return the data, otherwise continue with steps below
- Memtable (on heap): Read current memtable, and memtables awaiting flush. Get row fragments for the given row.
- Bloom Filter (off heap): Check for each SSTable to build list of candidate SSTables
- Key Cache (on heap): (If enabled) For each SSTable from above, probe the key cache to get position in data file. This may miss.
- SSTable Index summary (on heap): Probe here to find start of range in index file, seek to this position in the index file, then scan until you find the key
- SSTable (on disk): Seek to the row position in the SSTable and get the data
- Merge all row fragments, and reconcile duplicates via timestamp
- Update row cache
- Return results to client

Thanks!