Understanding Cassandra's internal architecture

Apache Cassandra:
Core Concepts, Skills, and Tools

Leo Schuman, Joe Chu

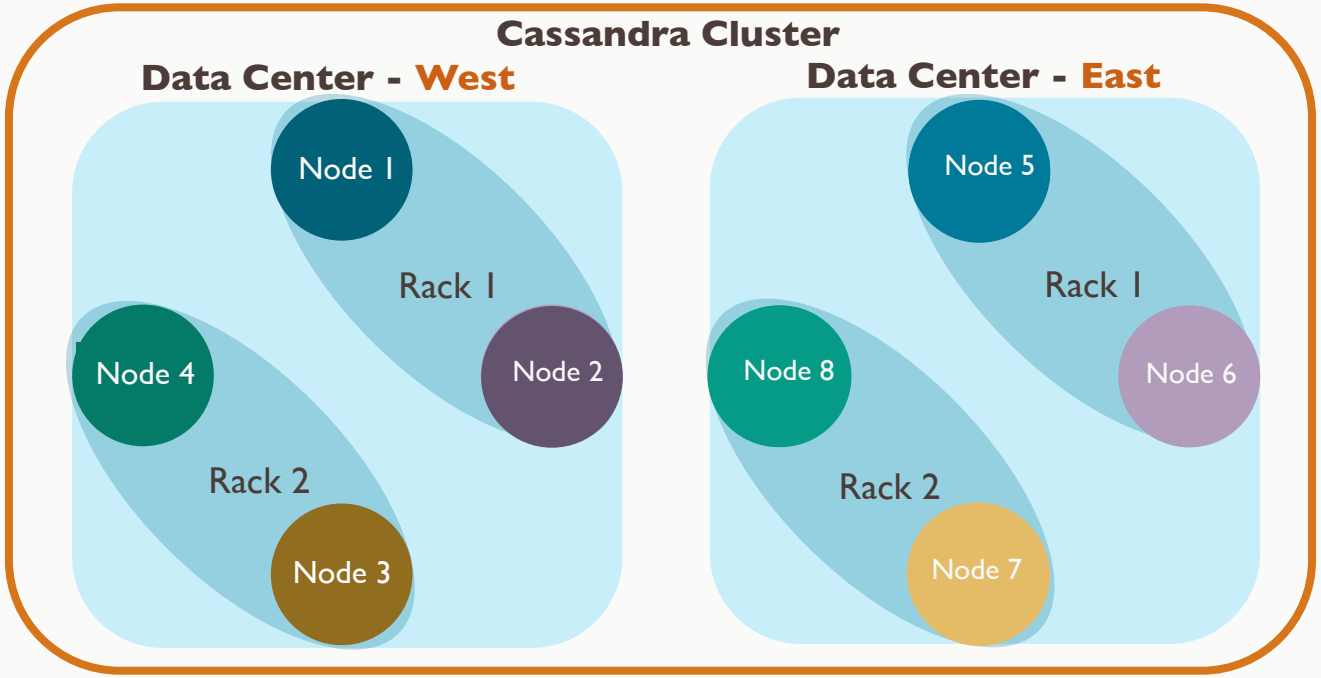
October, 2014

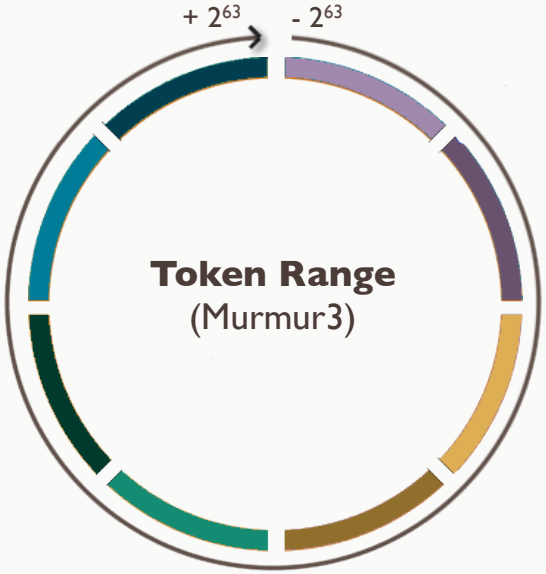
Learning Objectives

- **Understand how requests are coordinated**
- Understand replication
- Understand and tune consistency
- Introduce anti-entropy operations
- Understand how nodes communicate
- Understand the *System* keyspace

What is a cluster?

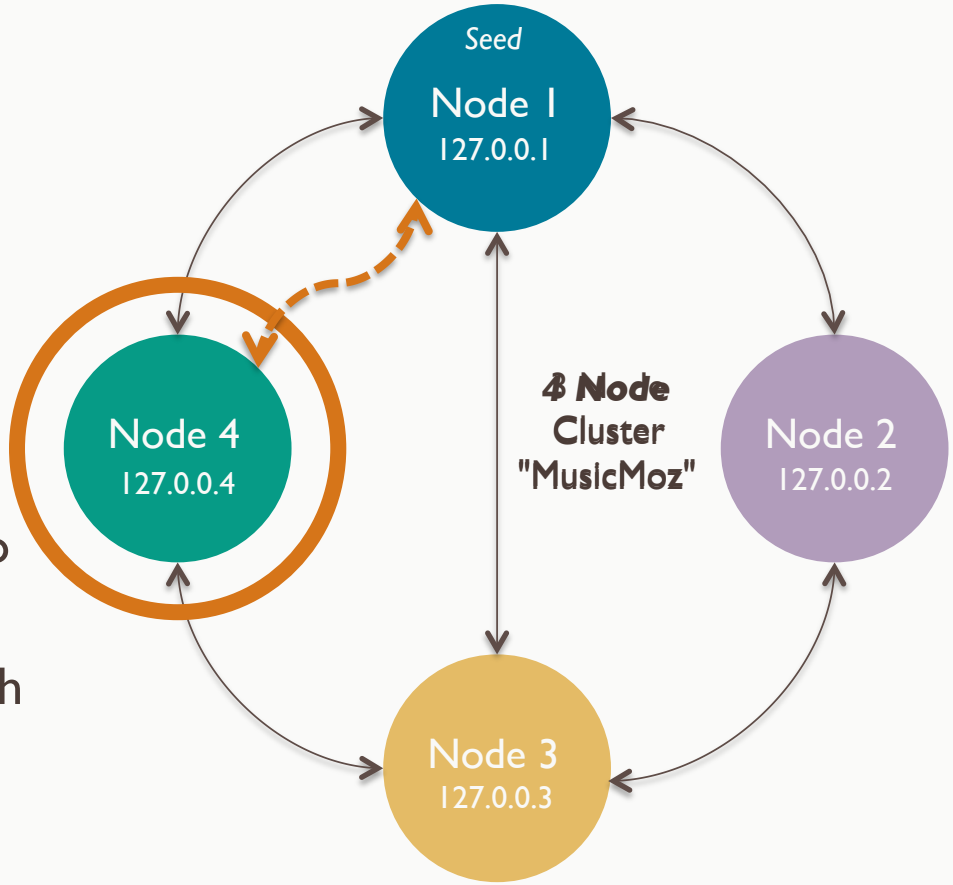
- A peer to peer set of nodes
  - Node – one Cassandra instance
  - Rack – a logical set of nodes
  - Data Center – a logical set of racks
  - Cluster – the full set of nodes which map to a single complete token ring





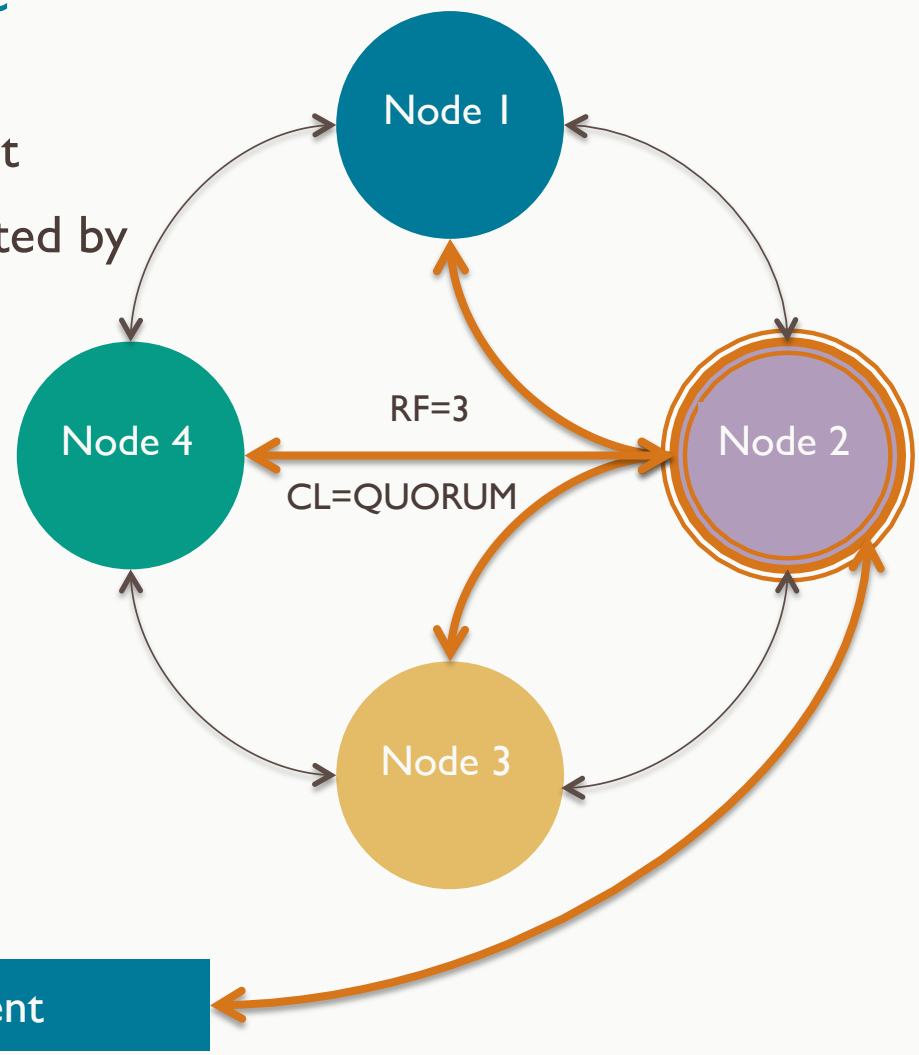
What is a cluster?

- 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 through which this particular node communicates



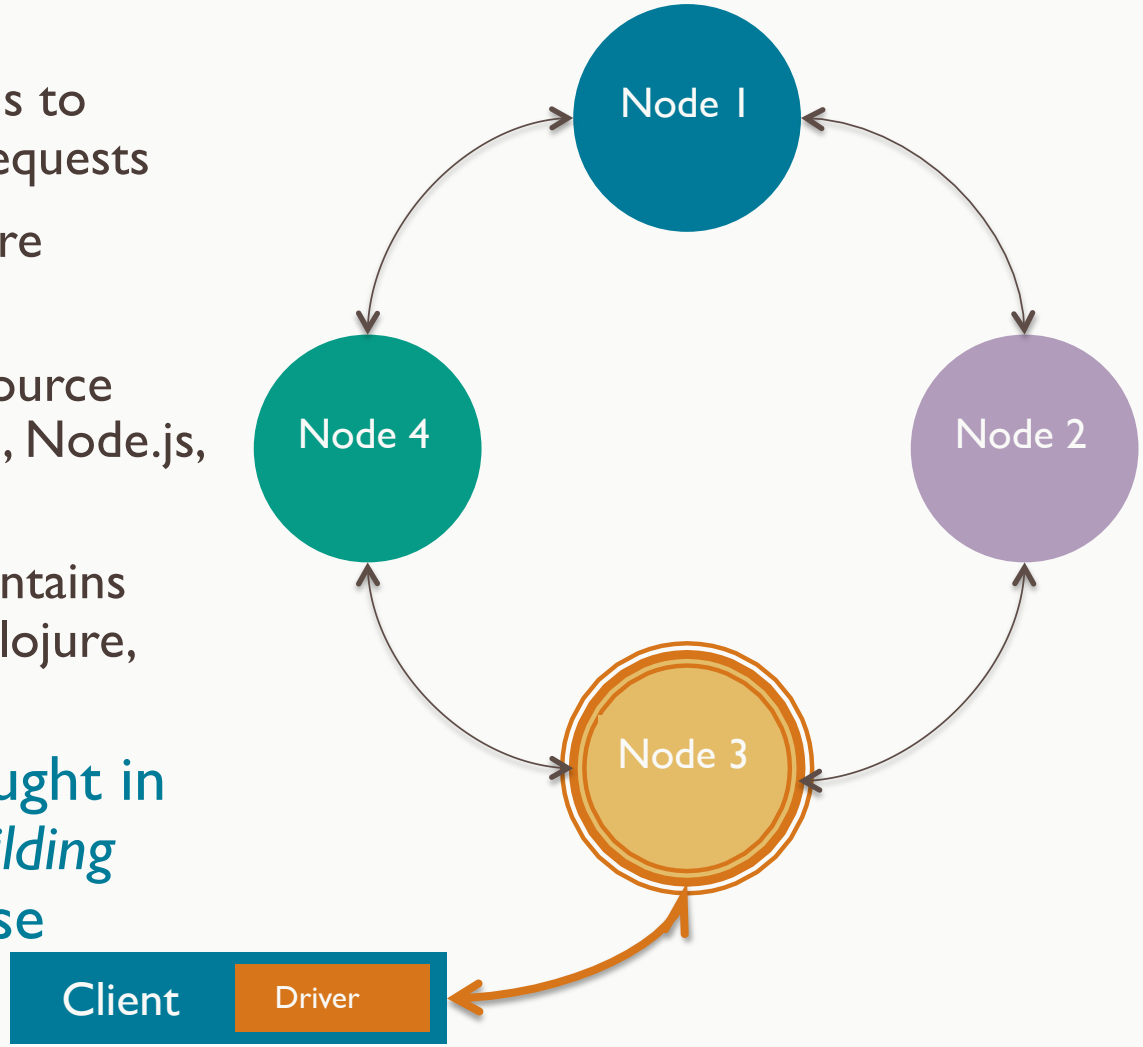
What is a coordinator?

- 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



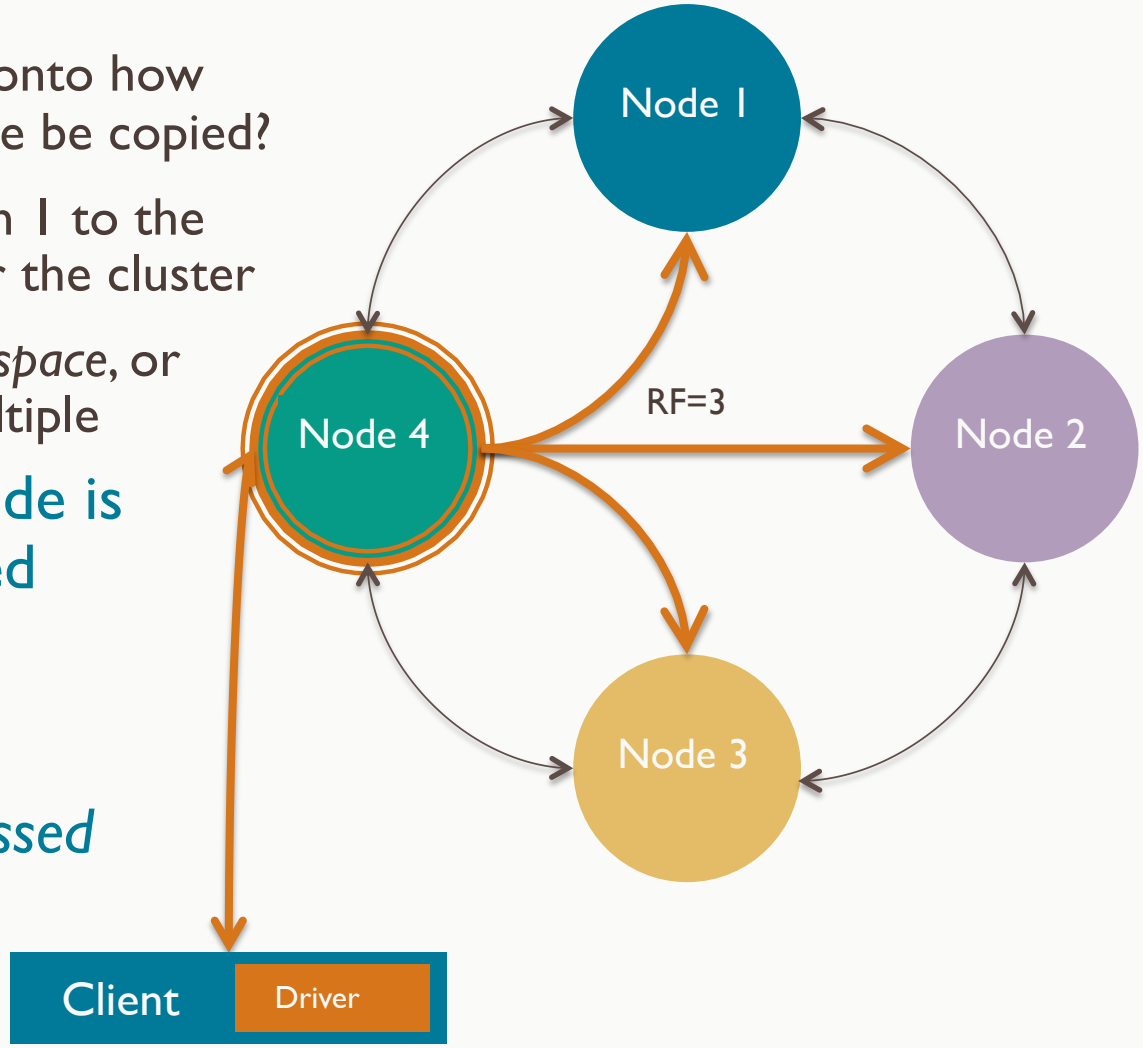
How are client requests coordinated?

- 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
  - Default policy is TokenAware DCAWARE
  - DataStax maintains open source drivers for Java, C#, Python, Node.js, Ruby, C/C++ (beta)
  - Cassandra Community maintains drivers for PHP, Perl, Go, Clojure, Haskell, R, Scala
- Client development is taught in the *Apache Cassandra: Building Scalable Applications* course



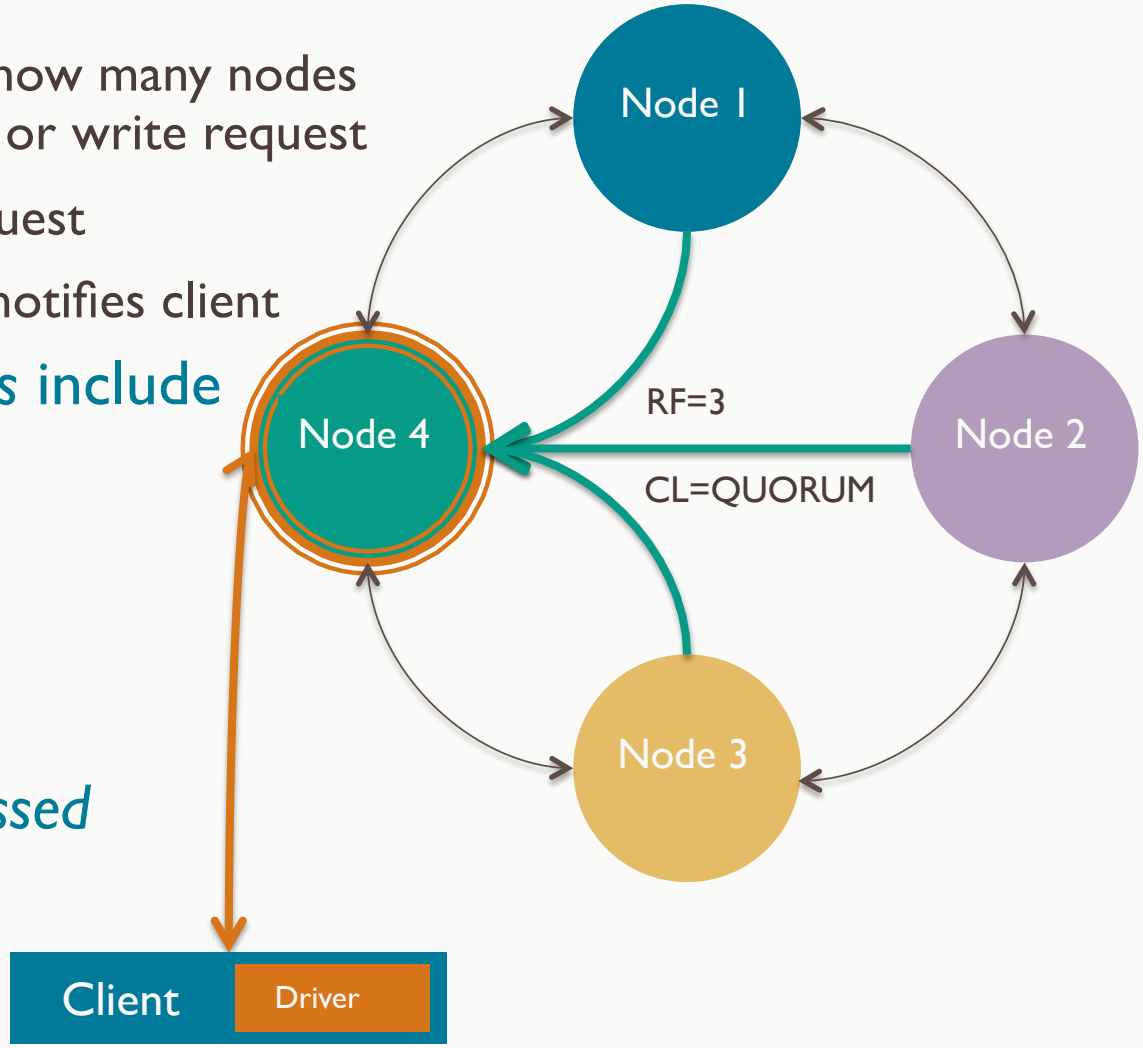
How are client requests coordinated?

- 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
- *Replication factor is discussed further ahead*



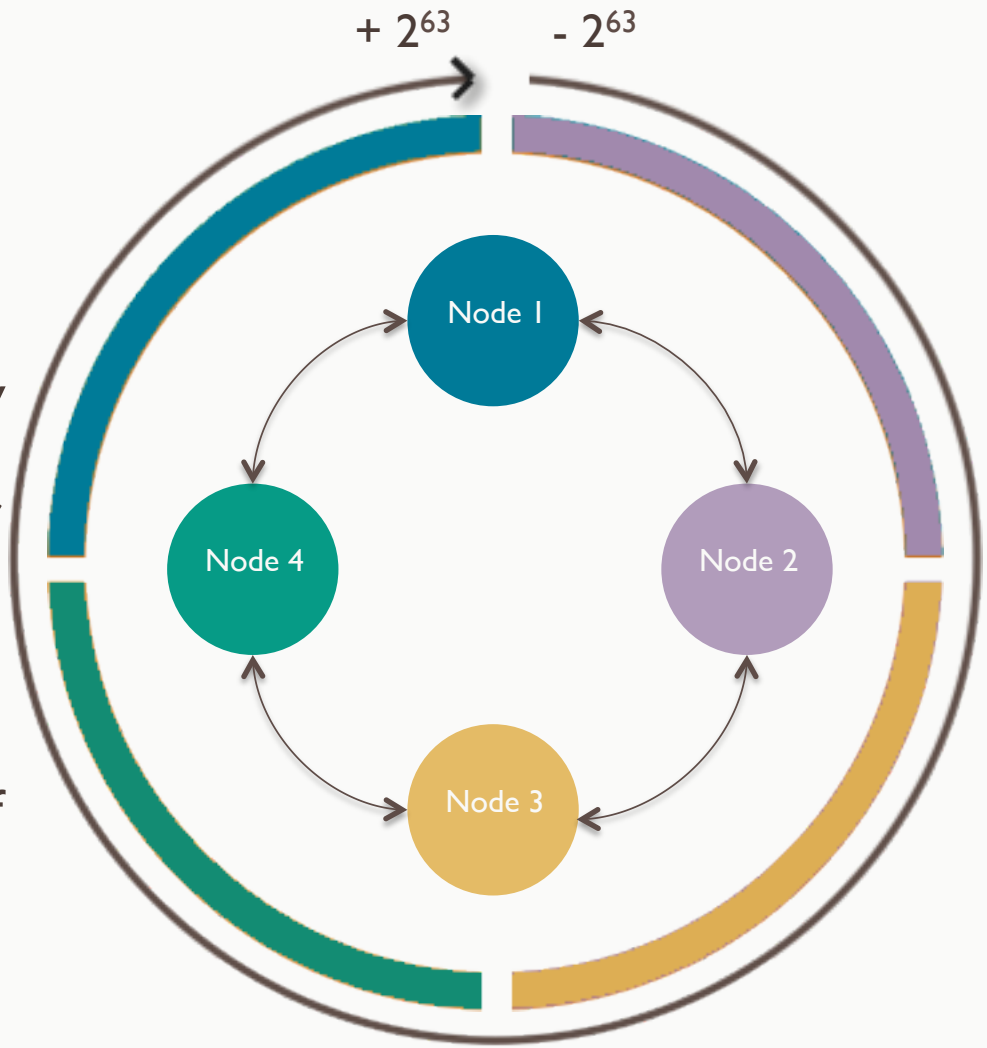
How are clients requests coordinated?

- 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
  - ANY
  - ONE
  - QUORUM (RF / 2) + 1
  - ALL
- *Consistency Level is discussed further ahead*



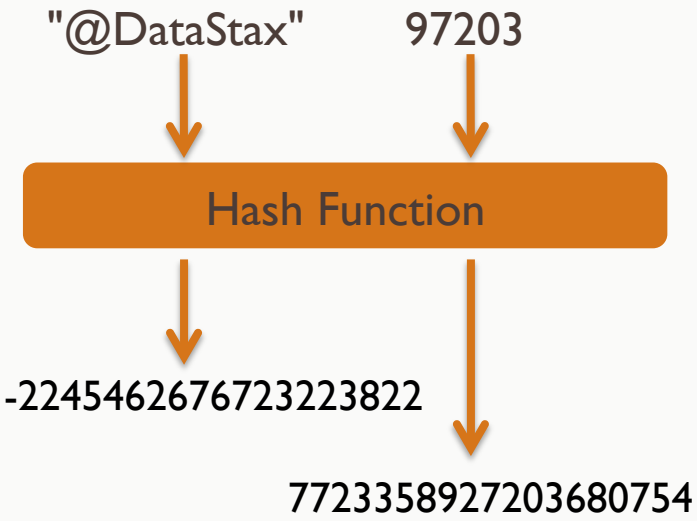
What is consistent hashing?

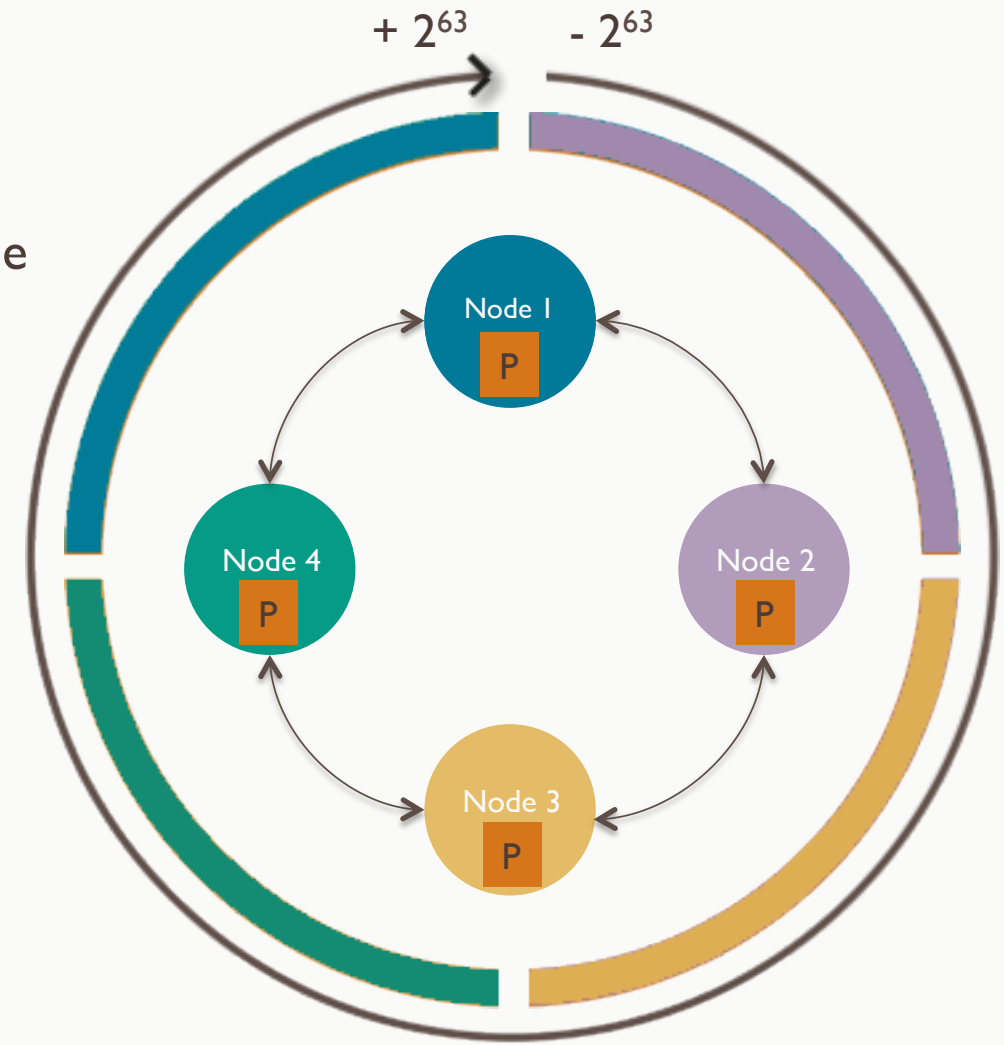
- Data is stored on *nodes* in *partitions*, each identified by a unique *token*
  - Partition – a storage location on a node (analogous to a "table row")
  - Token – integer value generated by a hashing algorithm, identifying a partition's location within a cluster
- The 2^{64} value *token range* for 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*
  - Partitioner options discussed ahead



What is the partitioner?

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

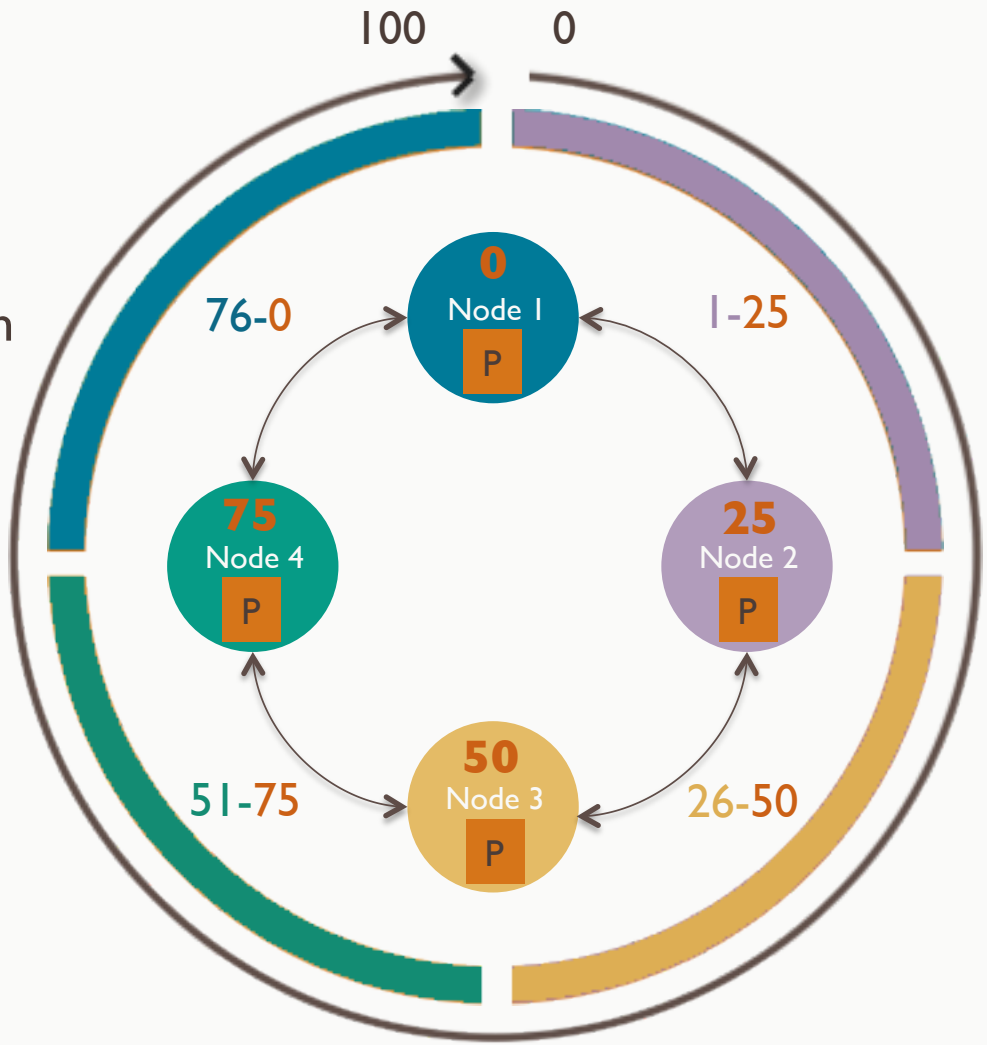




- Various partitioners available

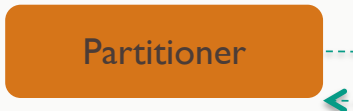
What is the partitioner?

- Imagine a 0 to 100 token range (instead of -2^{63} to $+2^{63}$)
  - Each node is assigned a token, just like each of its partitions
  - *Node tokens* are the highest value in the segment owned by that node
- This segment is the *primary token range* of replicas owned by this node
  - Nodes also store *replicas* keyed to tokens outside this range ("secondary range")



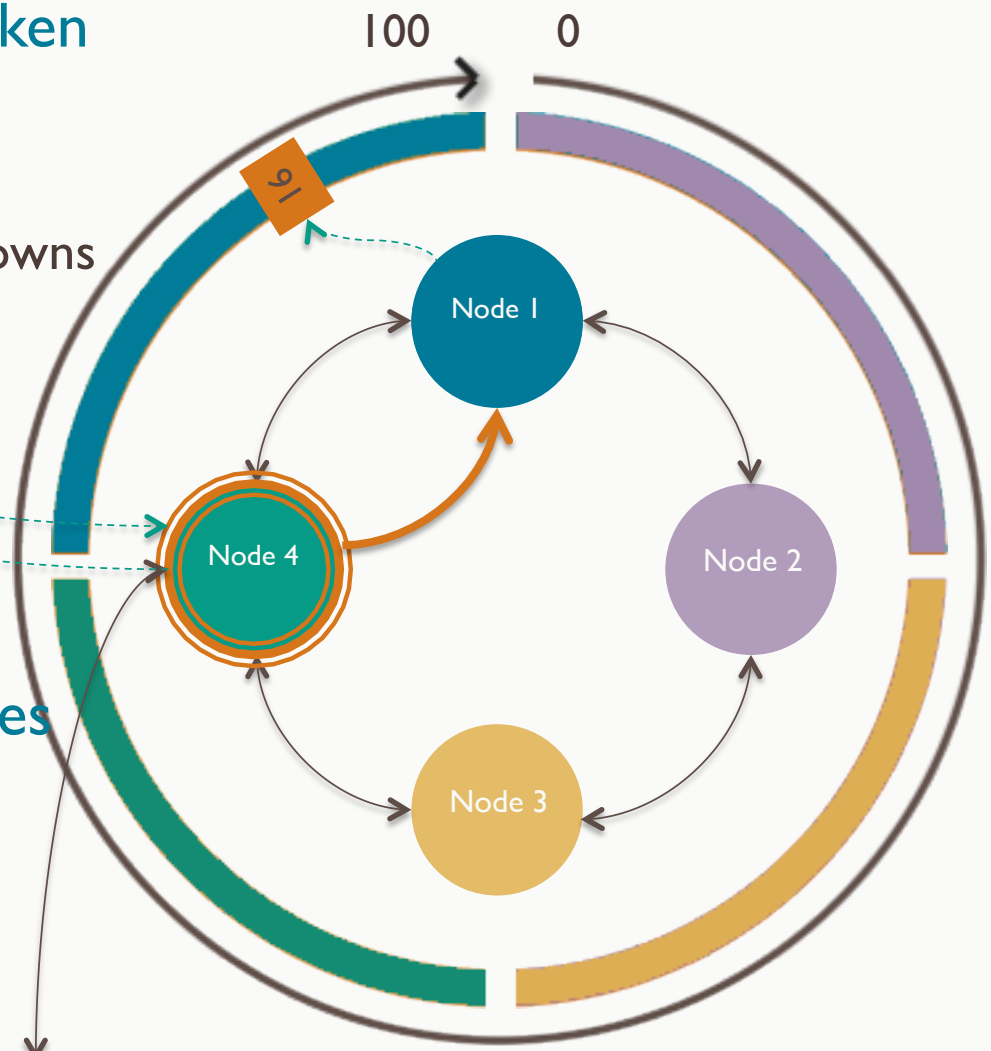
How does a partitioner work?

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



Token 91

'Orange:Oscar'



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

```
CREATE TABLE Users (
firstname text, lastname text, level text,
PRIMARY KEY ((lastname, firstname))
);
```

```
INSERT INTO Users (firstname, lastname, level)
VALUES ('Oscar', 'Orange', 42);
```

What partitioners does Cassandra offer?

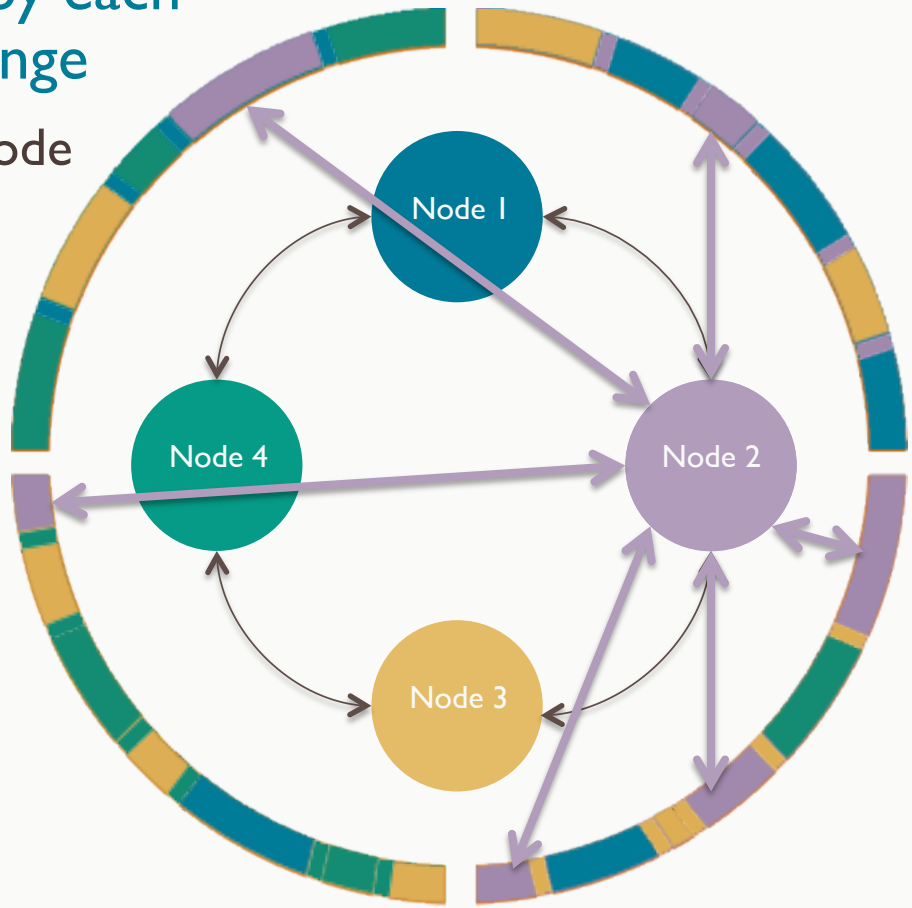
- Cassandra offers three partitioners
  - Murmur3Partitioner (default) – uniform distribution based on Murmur3 hash
  - RandomPartitioner – uniform distribution based on MD5 hash
  - ByteOrderedPartitioner (legacy only) – lexical distribution based on key bytes
- Murmur3Partitioner is the default and best practice
- The partitioner is configured in the *cassandra.yaml* file
  - Must be the same across all nodes in the cluster

cassandra.yaml

```
# The partitioner is responsible for distributing rows (by key) across
# nodes in the cluster.  Any IPartitioner may be used, including your
# own as long as it is on the classpath.  Out of the box, Cassandra
# provides org.apache.cassandra.dht.{Murmur3Partitioner, RandomPartitioner
# ByteOrderedPartitioner, OrderPreservingPartitioner (deprecated)}.
#
# See http://wiki.apache.org/cassandra/Operations for more on
# partitioners and token selection.
partitioner: org.apache.cassandra.dht.Murmur3Partitioner
```

What are virtual nodes?

- Multiple smaller primary range segments – virtual nodes – can be owned by each machine, instead of one larger range
  - virtual nodes behave like a regular node
  - available in Cassandra 1.2+
  - default is 256 per machine
  - not available on nodes combining Cassandra with Solr or Hadoop
- How are virtual nodes helpful?
  - token ranges are distributed, so machines bootstrap faster
  - impact of virtual node failure is spread across entire cluster
  - token range assignment automated



Note, for visual clarity, these slides illustrate concepts like replication with regular nodes not virtual nodes.

What are virtual nodes?

- Virtual nodes are enabled in *cassandra.yaml*
  - *partitions*, *regular nodes*, and *virtual nodes* are each identified by a *token*
  - regular or virtual node tokens are the highest value in one segment of the total token range for a cluster, which is the *primary range* of that node

*cassandra.yaml

```
# Cassandra storage config YAML

# This defines the number of tokens randomly assigned to this node on the ring
# The more tokens, relative to other nodes, the larger the proportion of data
# that this node will store. You probably want all nodes to have the same number
# of tokens assuming they have equal hardware capability.
#
# If you leave this unspecified, Cassandra will use the default of 1 token
# for legacy compatibility, and will use the initial_token as described below.
#
# Specifying initial_token will override this setting.
#
# If you already have a cluster with 1 token per node, and wish to migrate to
# multiple tokens per node, see http://wiki.apache.org/cassandra/Operations
num_tokens: 256
```

Learning Objectives

- Understand how requests are coordinated
- **Understand replication**
- Understand and tune consistency
- Introduce anti-entropy operations
- Understand how nodes communicate
- Understand the *System* keyspace

How are nodes organized as racks and data centers?

- A *cluster* of *nodes* can be logically grouped as *racks* and *data centers*
  - Node – the virtual or physical host of a single Cassandra instance
  - Rack – a logical grouping of physically related nodes
  - Data Center – a logical grouping of a set of racks
- Enables geographically aware read and write request routing
  - Cluster topology is communicated by the *Snitch* and *Gossip* (discussed ahead)
- Each *node* belongs to one *rack* in one *data center*
  - A default Cassandra node belongs to *rack1* in *datacenter1*
- The identity of each node's rack and data center may be configured in a property file

*cassandra-rackdc.properties

```
# These properties are used with GossipingPropertyFileSnitch and will
# indicate the rack and dc for this node
dc=DC1
rack=RAC1
```

How does the keyspace impact replication?

- Replication factor is configured when a keyspace is created
  - SimpleStrategy (learning use only) – one factor for entire cluster
    - assigned as "replication_factor"

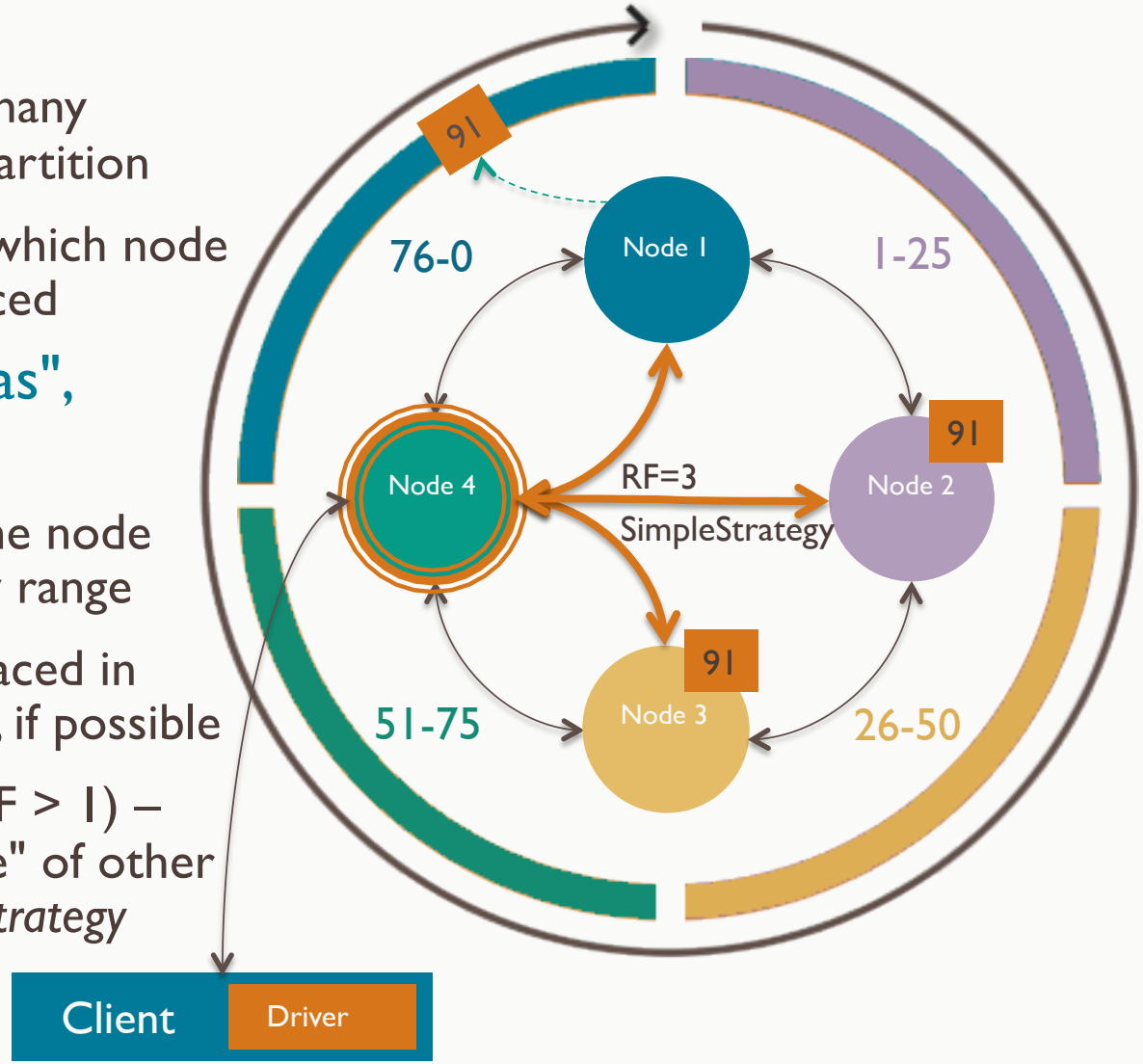
```
CREATE KEYSPACE simple-demo
WITH REPLICATION =
{'class':'SimpleStrategy',
 'replication_factor':2}
```

  - NetworkTopologyStrategy – separate factor for each data center in cluster
    - assigned by data center id (as also used in *cassandra-rackdc.properties*)

```
CREATE KEYSPACE simple-demo
WITH REPLICATION =
{'class':'NetworkTopologyStrategy',
 'dc-east':2, 'dc-west':3}
```

How does a coordinator forward write requests?

- The target table's *keyspace* determines
  - Replication factor – how many replicas to make of each partition
  - Replication strategy – on which node should each replica be placed
- All partitions are "replicas", there are no "originals"
  - First replica – placed on the node owning its token's primary range
  - Closest node – replicas placed in same rack and data center, if possible
  - (Subsequent) replicas (if RF > 1) – placed in "secondary range" of other nodes, per the *replication strategy*

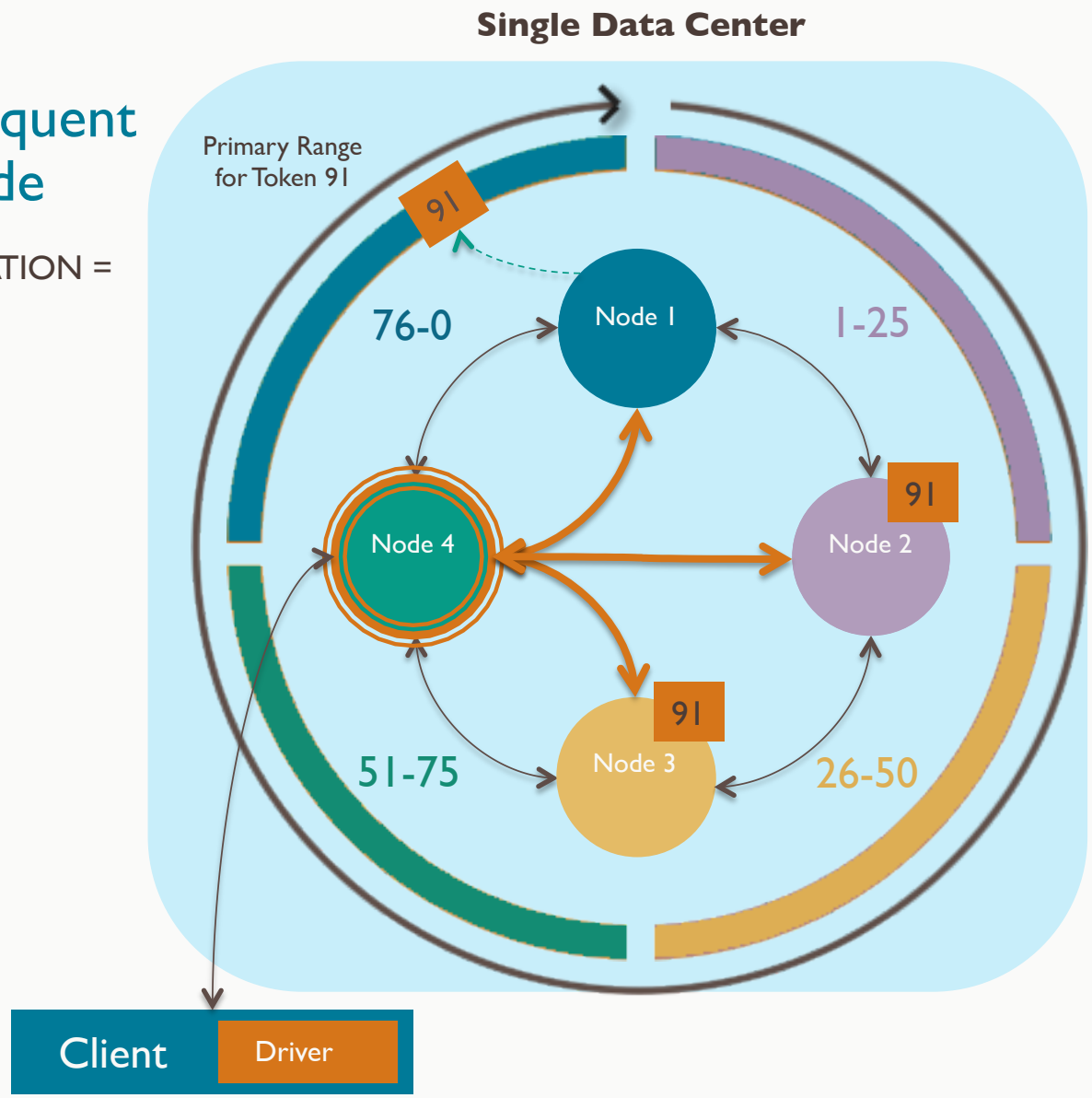


How is data replicated among nodes?

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

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

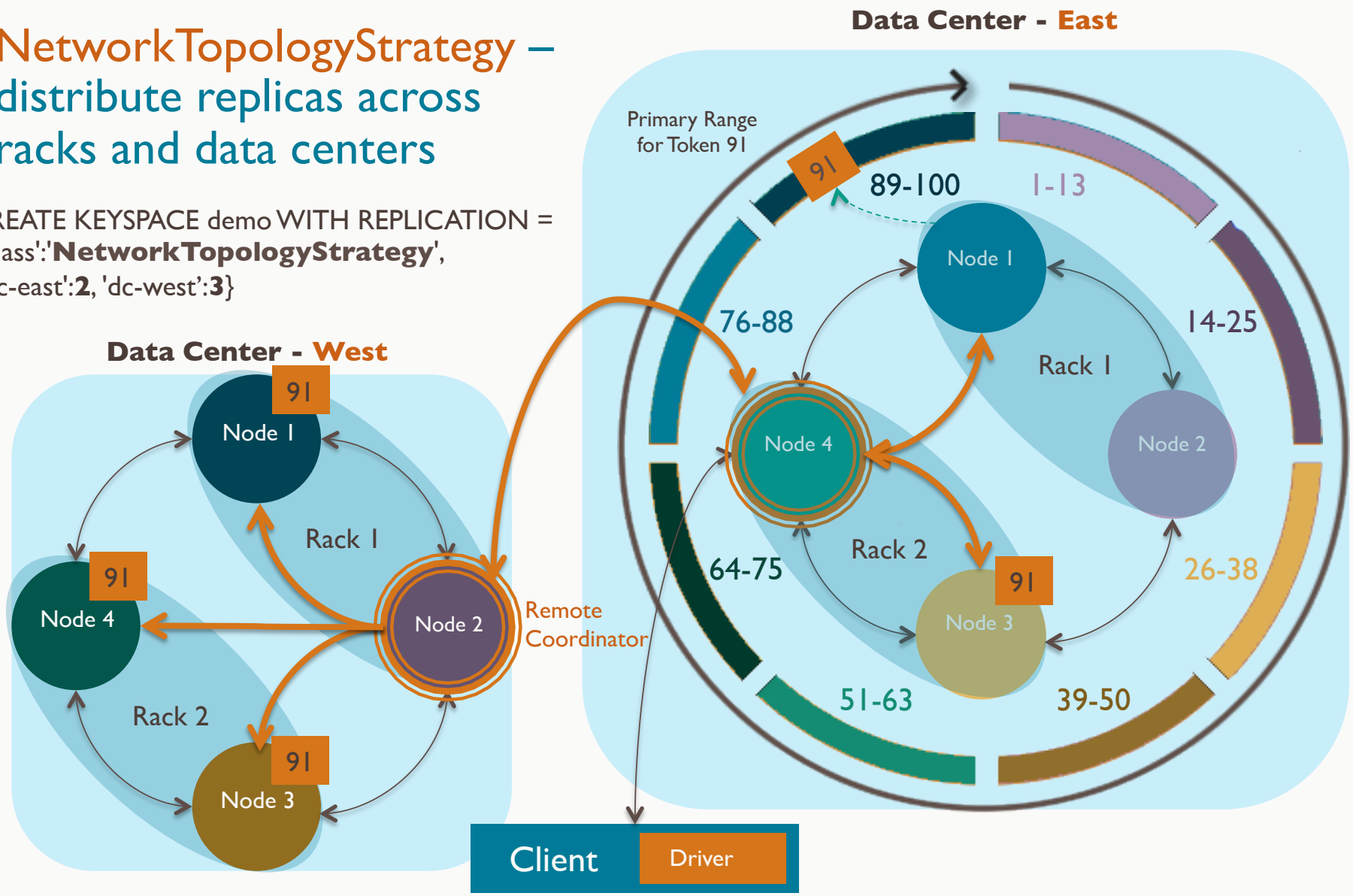
  - *replication factor* of 3 is a recommended minimum



How is data replicated between data centers?

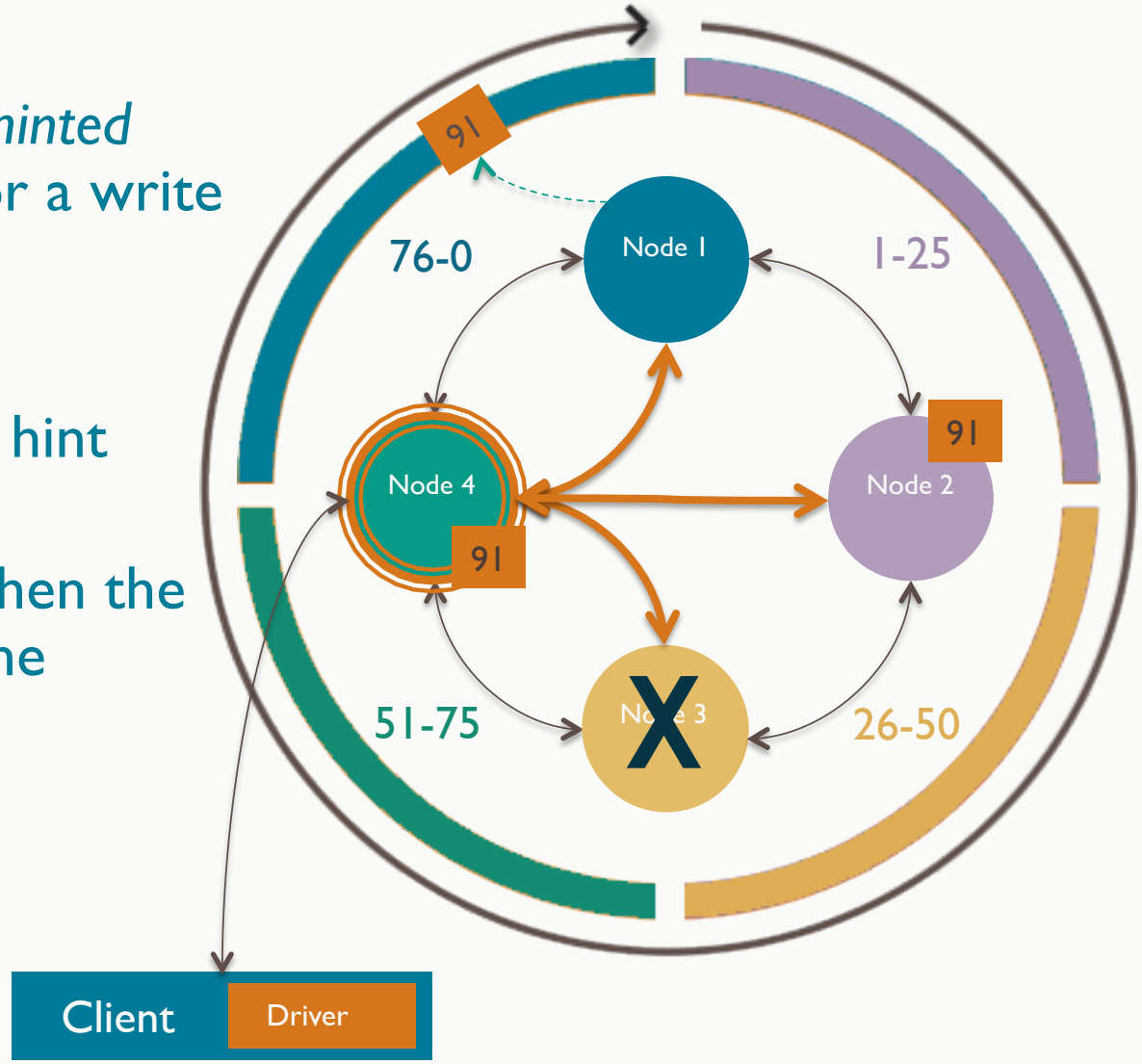
- NetworkTopologyStrategy – distribute replicas across racks and data centers

CREATE KEYSPACE demo WITH REPLICATION = {'class':'**NetworkTopologyStrategy**', 'dc-east':**2**, 'dc-west':**3**}



What is a hinted handoff?

- A recovery mechanism for writes targeting offline nodes
- *Coordinator* can store a *hinted handoff* if target node for a write
  - is known to be down, or
  - fails to acknowledge
- Coordinator stores the hint in its *system.hints* table
- The write is replayed when the target node comes online



What is a hinted handoff?

- The *hinted handoff* is comprised of
  - the target node location which is down
  - the partition which requires a replay
  - the data to be written
- Configurations in the *cassandra.yaml* file include
  - hinted_handoff_enabled (default: true) – HH enabled per DC or disabled
  - max_hint_window_in_ms (default: 3 hours) – after this consecutive outage period hints are no longer generated until target node comes back online
    - nodes offline longer are made consistent using *repair* or other operations

cassandra.yaml

```
# See http://wiki.apache.org/cassandra/HintedHandoff
# May either be "true" or "false" to enable globally, or contain a list
# of data centers to enable per-datacenter.
# hinted_handoff_enabled: DC1,DC2
hinted_handoff_enabled: true
# this defines the maximum amount of time a dead host will have hints
# generated.  After it has been dead this long, new hints for it will not be
# created until it has been seen alive and gone down again.
max_hint_window_in_ms: 10800000 # 3 hours
```

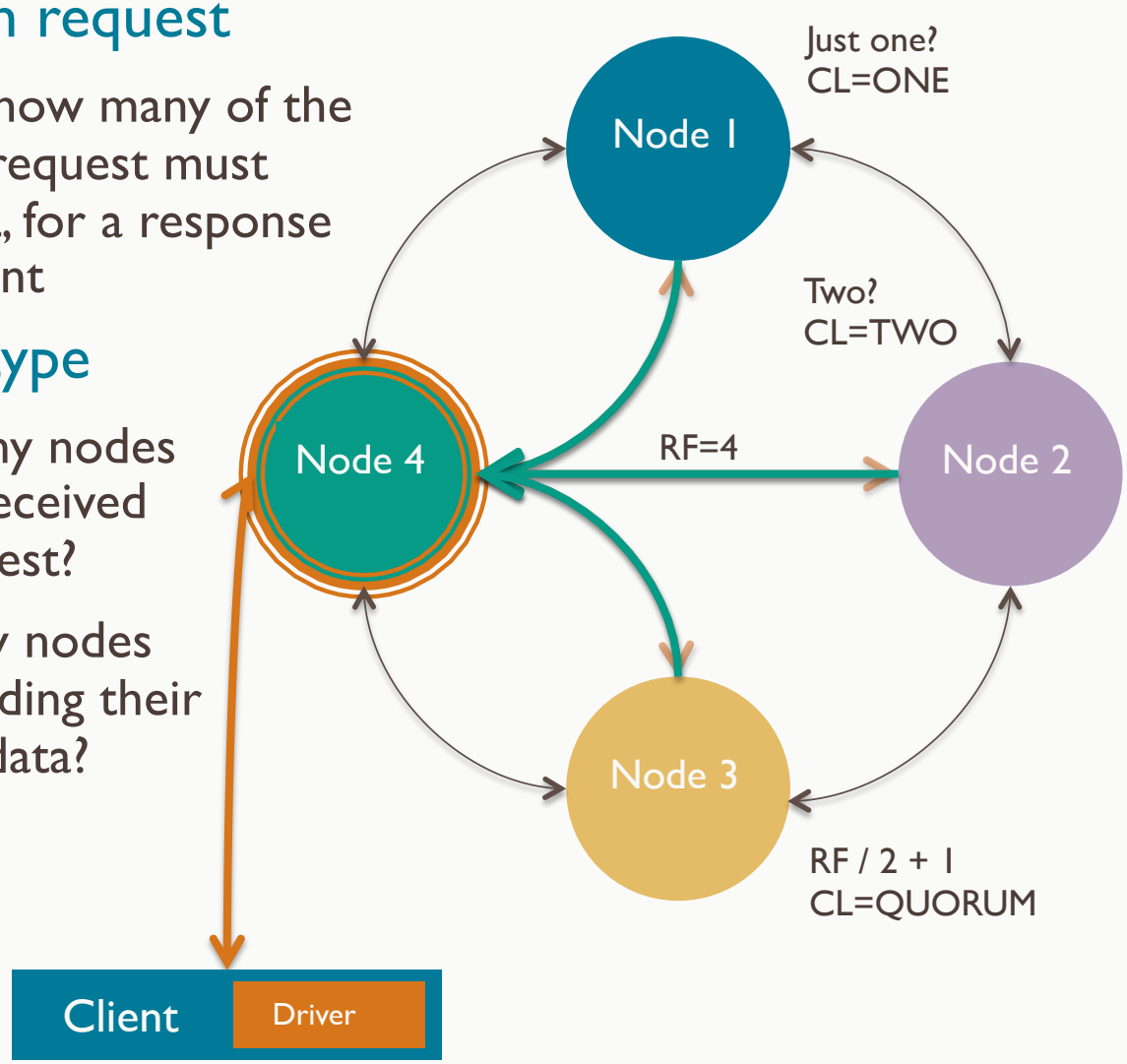
Exercise 1: Create a keyspace

Learning Objectives

- Understand how requests are coordinated
- Understand replication
- **Understand and tune consistency**
- Introduce anti-entropy operations
- Understand how nodes communicate
- Understand the *System* keyspace

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 lowest consistency (writes)
ALL	Check all nodes. Fail if any is down.	Highest consistency and lowest availability
ONE (TWO, THREE)	Check closest node to coordinator.	Highest availability and lowest consistency (reads)
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

How do you set consistency per request?

- The default *consistency level* for all requests is ONE
  - In *cqlsh*, the *CONSISTENCY* command modifies this value for all subsequent requests during the same *cqlsh* session
  - In *client drivers*, a *ConsistencyLevel* constant is passed as part of each request

Cassandra | dstraining@DST: /home/dsc-c

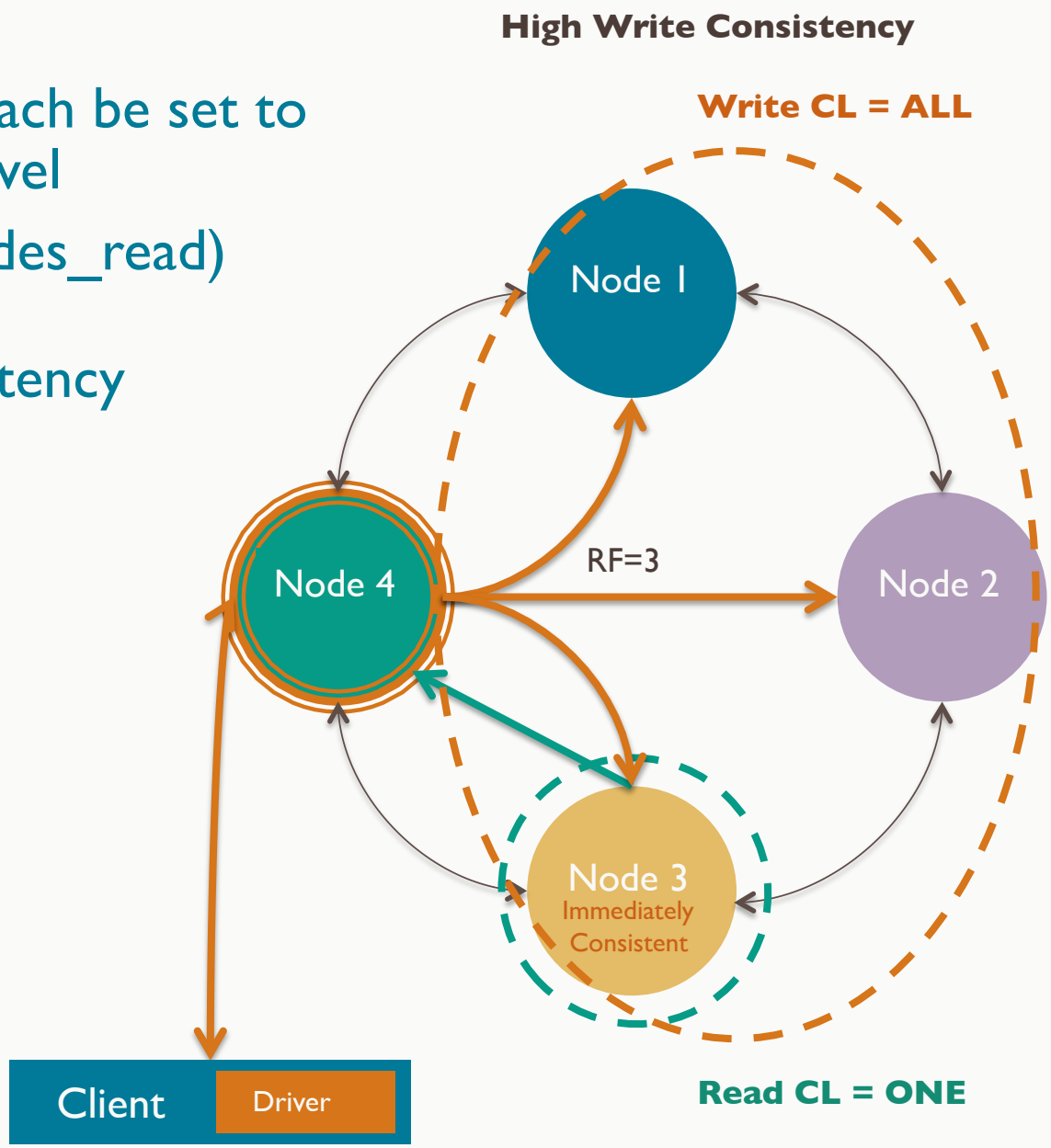
```
dstraining@DST:/home/dsc-cassandra-2.0.5/bin$ ./cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.5 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
Use HELP for help.
cqlsh> USE demo;
cqlsh:demo> CONSISTENCY;
Current consistency level is ONE.
cqlsh:demo> CONSISTENCY QUORUM;
Consistency level set to QUORUM.
cqlsh:demo> CONSISTENCY ANY;
Consistency level set to ANY.
cqlsh:demo> CONSISTENCY ALL;
Consistency level set to ALL.
cqlsh:demo>
```

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 only one replica node is checked before a result is returned
  - *Lowest latency* because the response from one replica is immediately returned

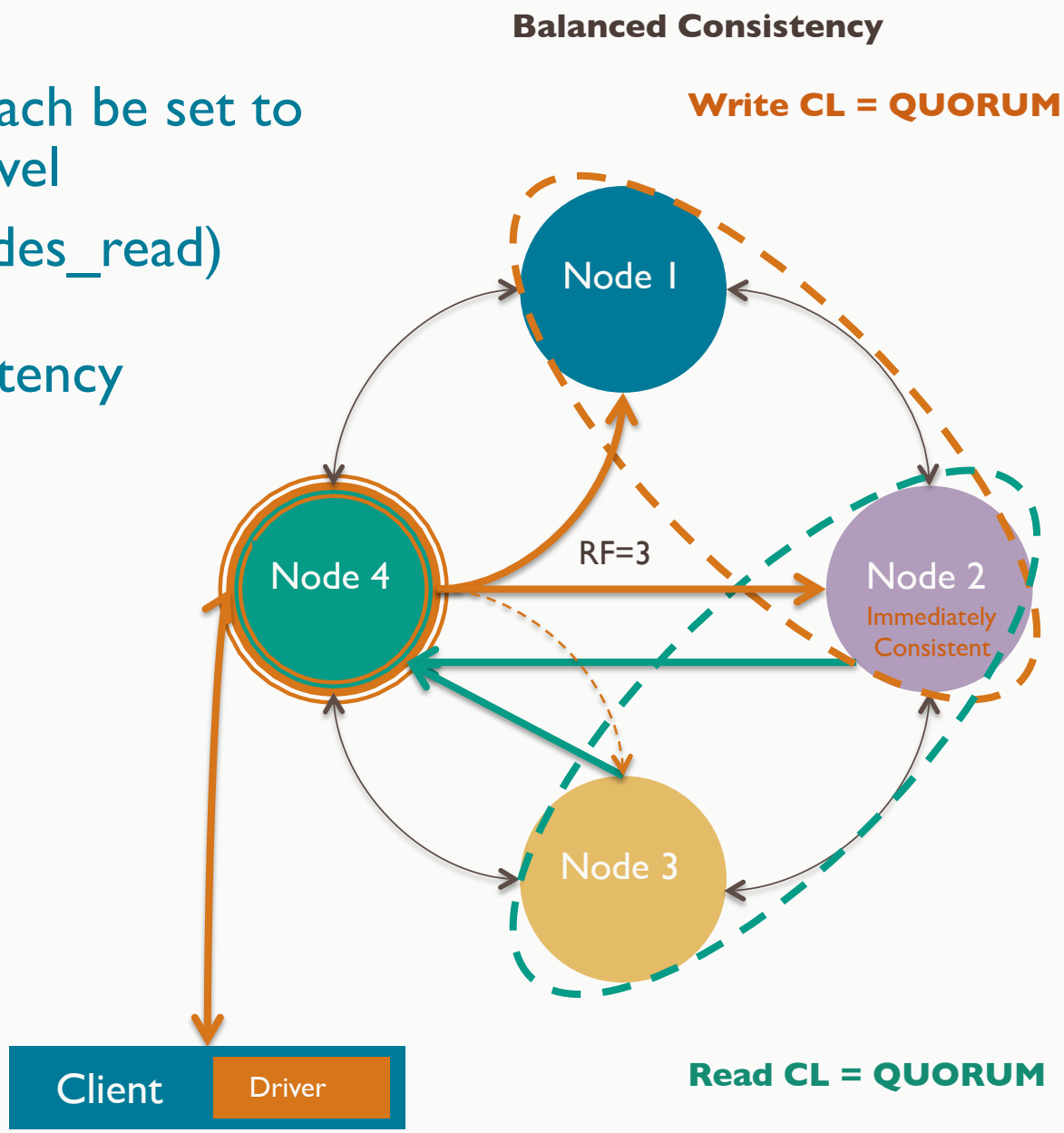
What does it mean to tune consistency?

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



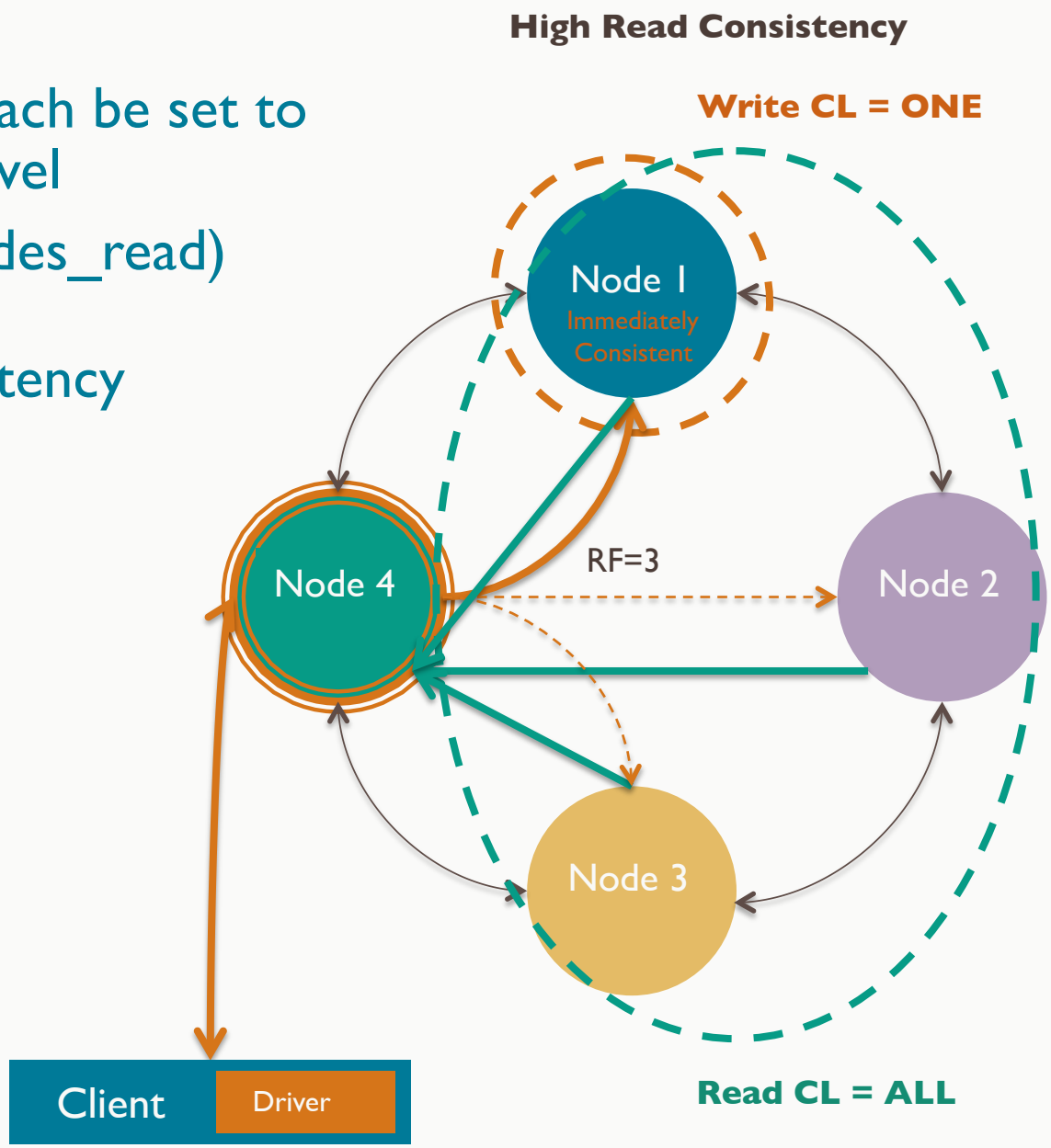
What does it mean to tune consistency?

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



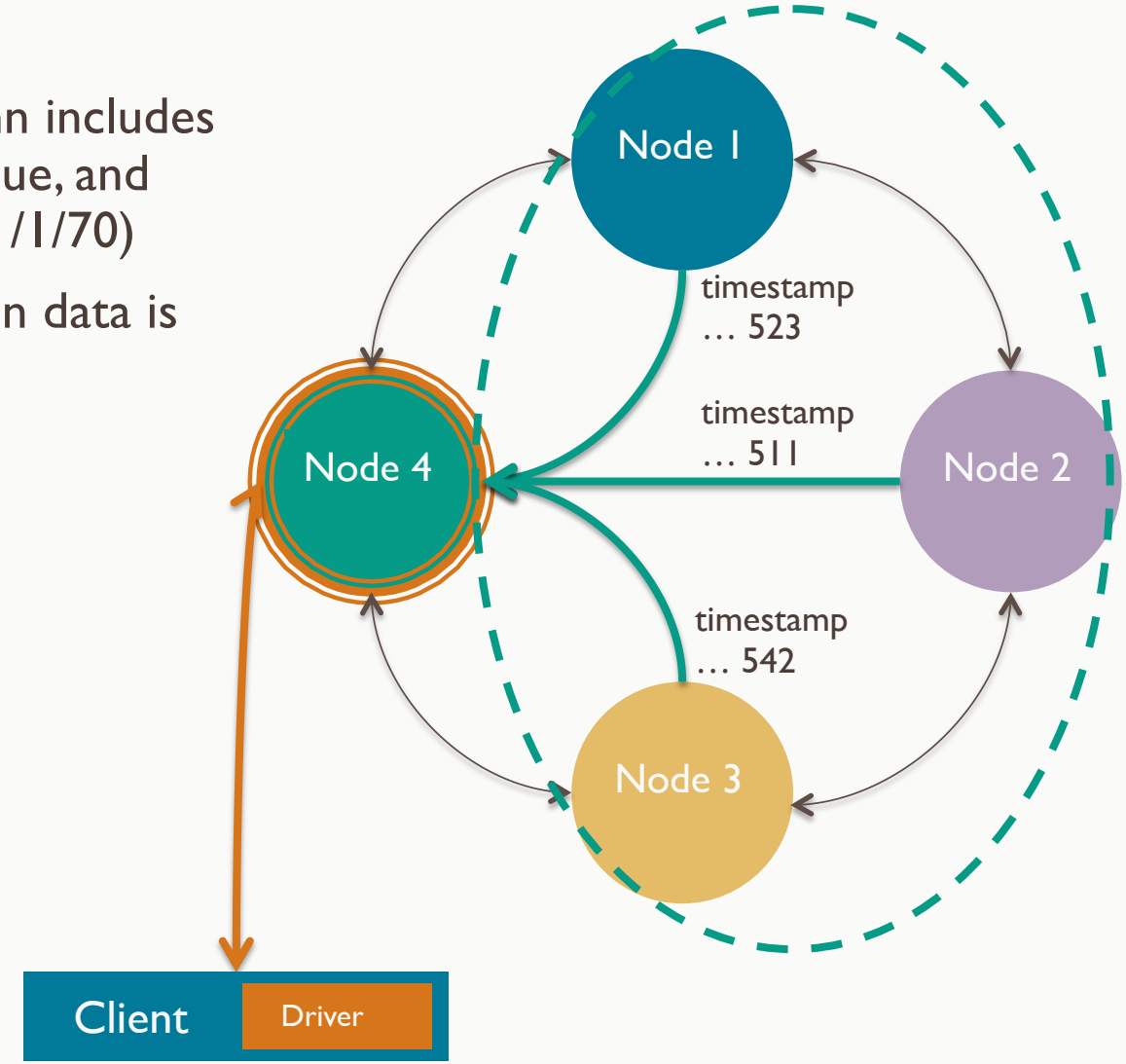
What does it mean to tune consistency?

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



Why is server clock synchronization significant?

- Clock synchronization across nodes is critical because
  - Every write to any column includes column name, column value, and timestamp since epoch (1/1/70)
  - The most recently written data is returned to the client



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 (if read CL + write CL < RF)	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?

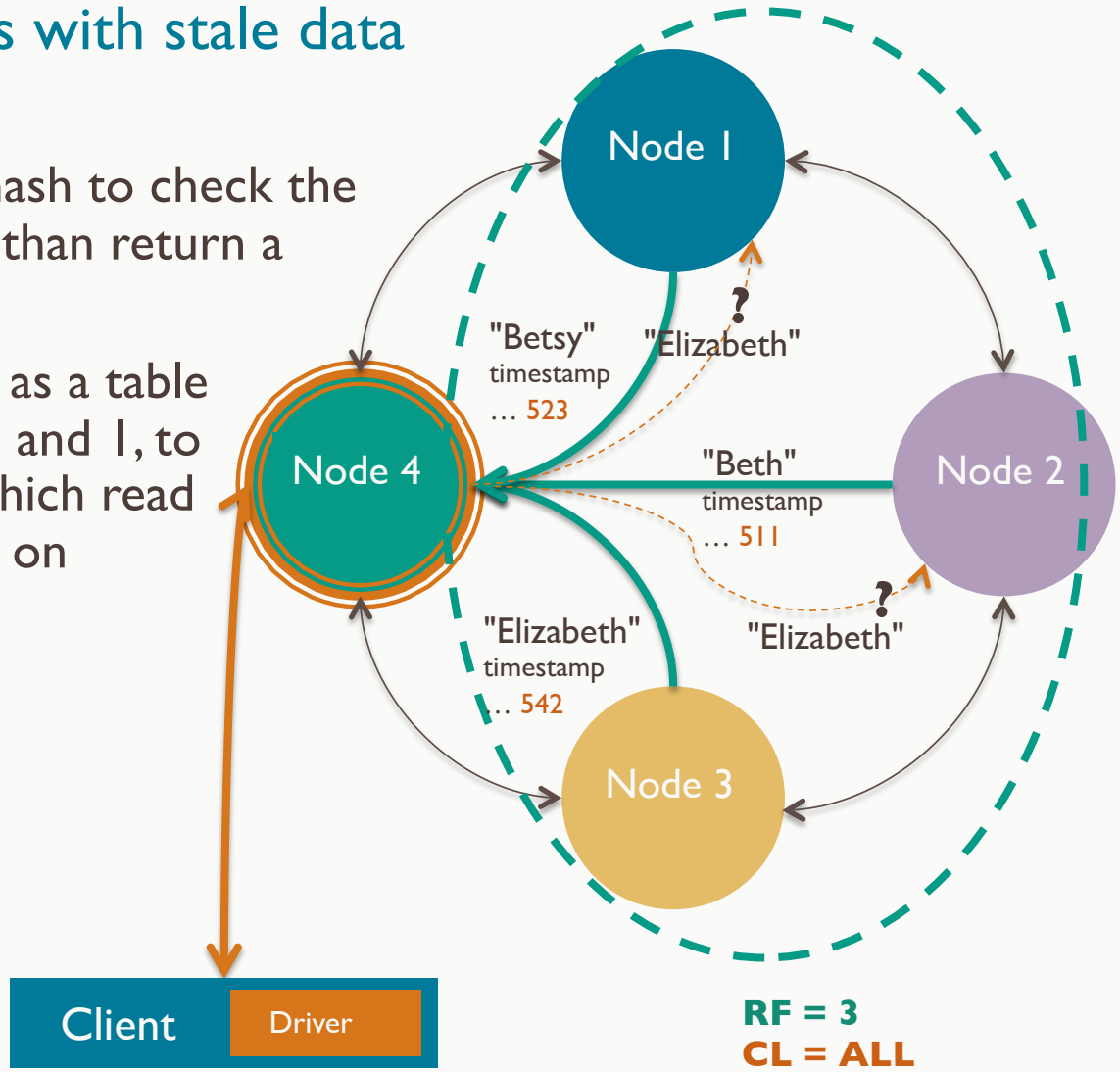
Exercise 2: Working with consistency level

Learning Objectives

- Understand how requests are coordinated
- Understand replication
- Understand and tune consistency
- **Introduce anti-entropy operations**
- Understand how nodes communicate
- Understand the *System* keyspace

What is read repair?

- As part of a read, a *digest query* is sent to replica nodes, and nodes with stale data are updated
  - digest query – returns a hash to check the current data state, rather than return a complete query result
  - read_repair_chance – set as a table property value between 0 and 1, to set the probability with which read repairs should be invoked on non-quorum reads



How are node repairs initiated using nodetool?

- The *nodetool repair* command makes all data on a node consistent with the most current replicas in the cluster
  - clusters with high writes/deletes at CL < ALL require periodic repair

```
bin/nodetool -h [host] -p [port] repair [options]
```

- Options help mitigate heavy disk use from this operation
  - --partitioner-range – option restricts repair to node's primary range only
  - --start-token [uuid] --end-token [uuid] – restrict repair to this token range
  - -dc [name] or -local – repair named data center, or local center only

How are node repairs initiated using nodetool?

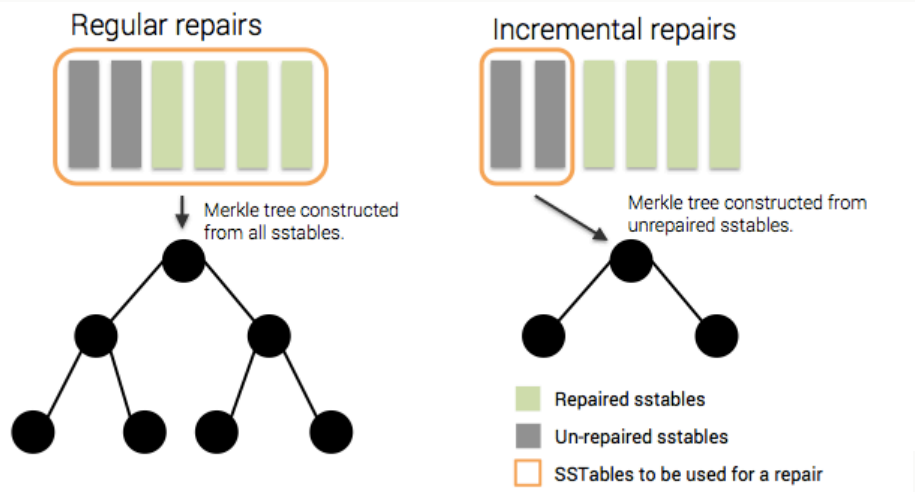
- When to run *nodetool repair*
  - recovering a failed node
  - bringing a downed node back online
  - periodically on nodes with infrequently read data
  - periodically on nodes with write or delete activity
- If run periodically, do so at least every gc_grace_seconds
  - gc_grace_seconds – tombstone garbage collection period (default: 864000)
  - tombstone – marker placed on a deleted column within a partition
  - failure to repair within this period can lead to deleted column resurrection

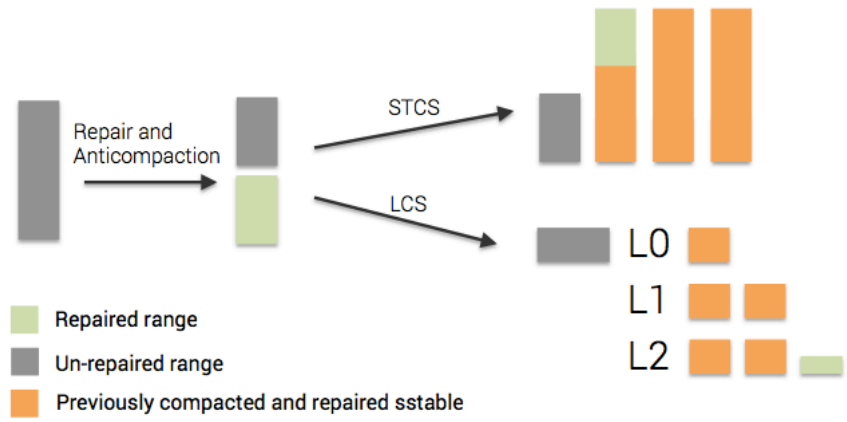
```
ALTER TABLE performer
WITH gc_grace_seconds = 432000;
```

Note, gc_grace_seconds and repair are discussed further, later in context of compaction

What is incremental repair?

- Repairs data that has not been previously repaired
  - Merkle trees are only generated and compared for unrepaired SSTables
  - Unrepaired SSTables are determined by a *repairedAt* property in its metadata
  - An *anticompaction* occurs after the incremental repair to separate the repaired and unrepaired ranges



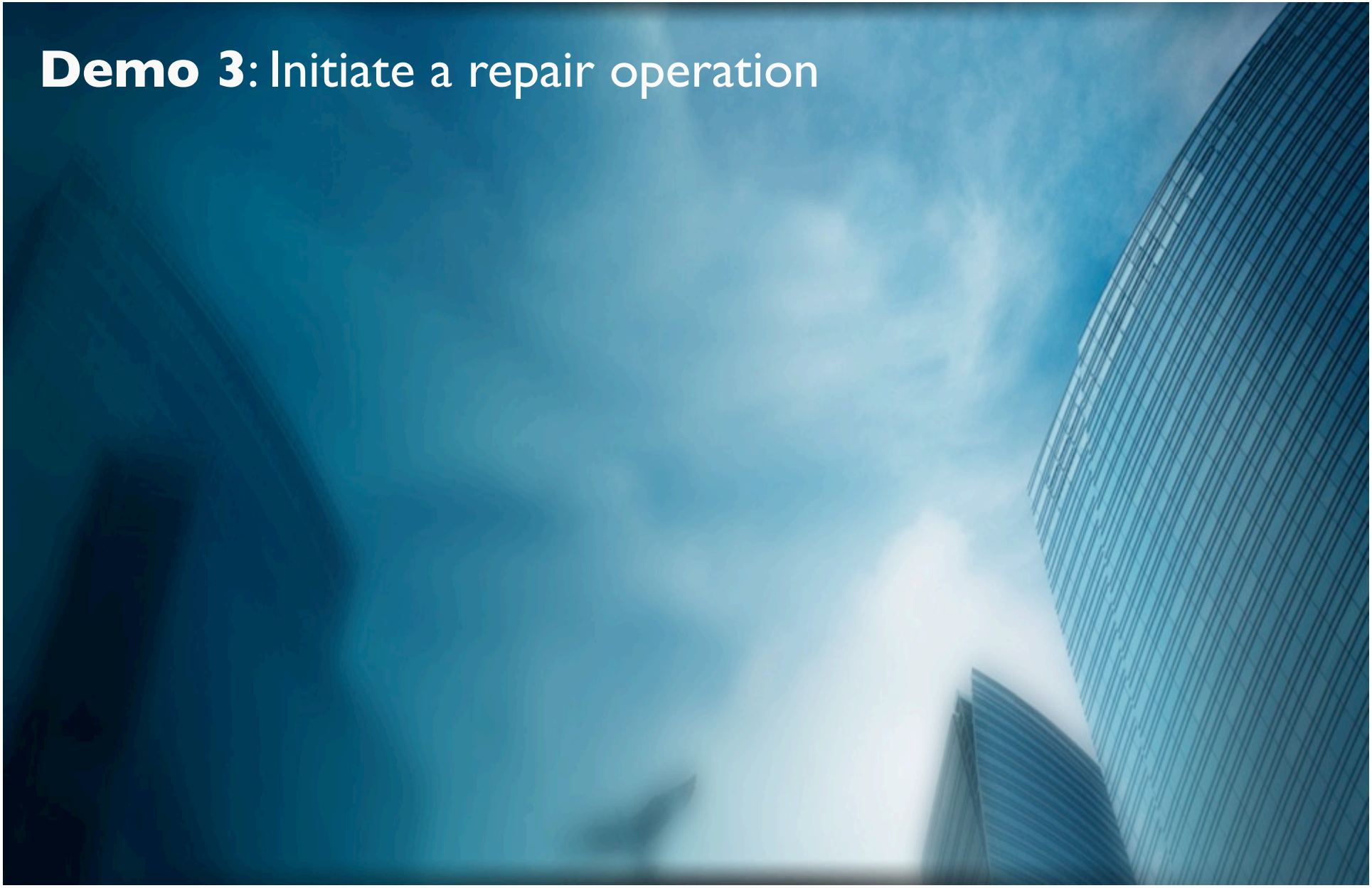



```
bin/nodetool -h [host] -p [port] repair --incremental [options]
```

What is incremental repair?

- Tools provided for SSTables when incremental repair is used
  - tools/bin/sstablemetadata – display the *repairedAt* property for a SSTable
  - tools/bin/sstablerepairedset – manually set a SSTable as being repaired
- Guidelines for incremental repair
  - Recommended if using leveled compaction
  - Run incremental repairs daily
  - Avoid anticompaction by not using a partitioner range or subrange

Demo 3: Initiate a repair operation

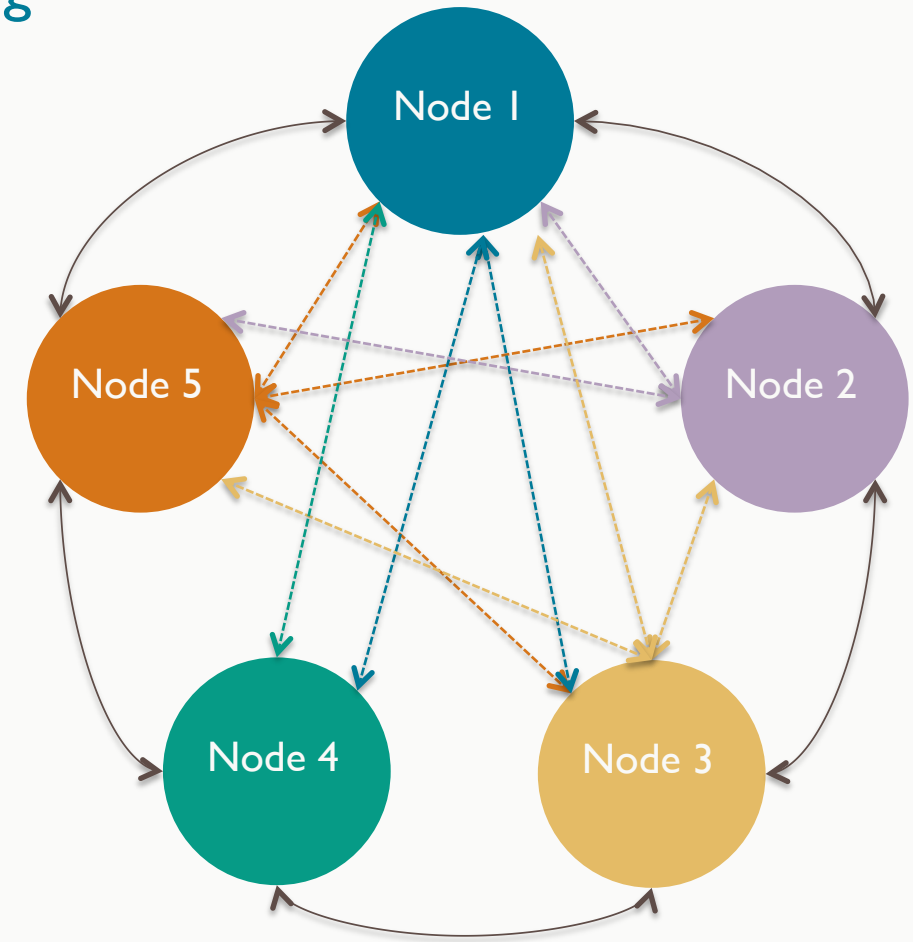


Learning Objectives

- Understand how requests are coordinated
- Understand replication
- Understand and tune consistency
- Introduce anti-entropy operations
- **Understand how nodes communicate**
- Understand the *System* keyspace

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



What is the Gossip protocol?

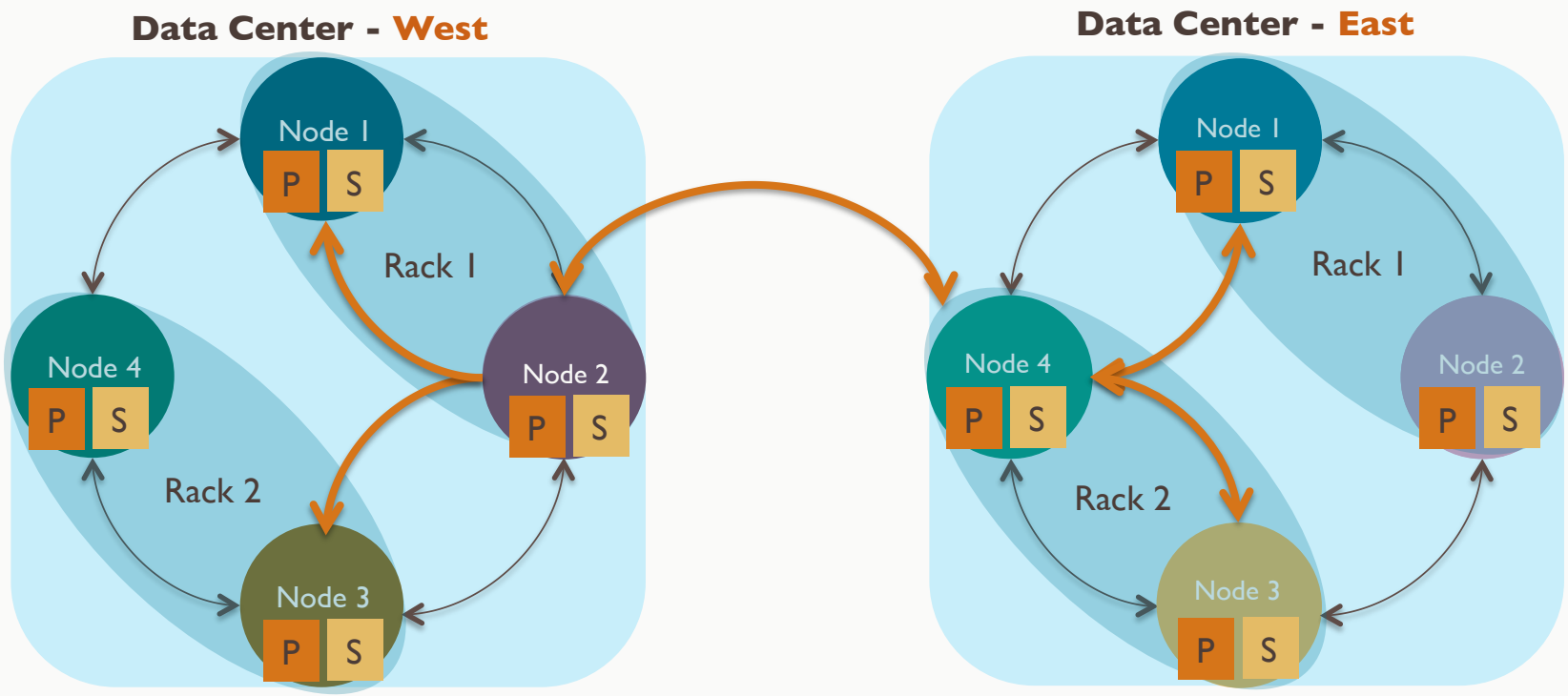
- As a node joins a cluster, it gossips with the *seed nodes* set in its *cassandra.yaml* to learn its cluster's topology
  - Assign the same seed nodes to each node in each data center
  - If more than one data center, include a seed node from each

cassandra.yaml

```
# any class that implements the SeedProvider interface and has a
# constructor that takes a Map<String, String> of parameters will do.
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other and learn
    # the topology of the ring.  You must change this if you are running
    # multiple nodes!
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "127.0.0.1"
```

What is the Snitch and how is it configured?

- The Snitch
  - informs its *partitioner* of their node's *rack* and *data center* topology
  - enables replication which avoids duplication within a rack
    - *duplicate replicas within a rack risk data loss if that rack fails*



What is the Snitch and how is it configured?

- The *endpoint_snitch* is a Java class implementing the *IEndpointSnitch* interface, which you assign in *cassandra.yaml*

cassandra.yaml

```
# endpoint_snitch -- Set this to a class that implements
# IEndpointSnitch.  The snitch has two functions:
# You can use a custom Snitch by setting this to the full class name
# of the snitch, which will be assumed to be on your classpath.
endpoint_snitch: SimpleSnitch
```

How does the Snitch aid with partition placement?

- Nine Snitch options are distributed with Cassandra
  - SimpleSnitch – node proximity determined by the strategy declared for the keyspace, single data center only
  - PropertyFileSnitch – node proximity determined by rack and data center configuration in cassandra-topology.properties
  - GossipingPropertyFileSnitch – node proximity determined by this node's rack and data center in cassandra-rackdc.properties, and propagated by Gossip
  - YamlFileNetworkTopologySnitch – node proximity determined by rack and data center configuration in cassandra-topology.yaml
  - RackInferringSnitch – sample for writing a custom Snitch
  - Ec2Snitch – Amazon EC2 aware, treating an EC2 Region as the data center, and EC2 Availability Zone as the racks; uses private IPs, so single region only
  - Ec2MultiRegionSnitch – as with Ec2Snitch, but uses public IPs for each node's broadcast_address, enabling multiple EC2 Regions / data centers
  - GoogleCloudSnitch – used with the Google Cloud Platform
  - CloudstackSnitch – for Apache Cloudstack environments

Demo 4: Review the snitch setting in cassandra.yaml

Learning Objectives

- Understand how requests are coordinated
- Understand replication
- Understand and tune consistency
- Introduce anti-entropy operations
- Understand how nodes communicate
- **Understand the *System* keyspace**

What is the System keyspace?

- Cassandra stores its state in *System* keyspace tables
  - Examining *System* keyspace tables is useful towards understanding Cassandra
  - But, directly editing any *System* keyspace table is an anti-pattern, so don't

dstraining@DST: /home/cassandra | dstraining@DST: /home/dsc-cassandra-2.0.5

```
dstraining@DST:/home/cassandra$ bin/cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.5 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
Use HELP for help.
cqlsh> USE SYSTEM;
cqlsh:system> SELECT * FROM system.schema_keyspaces;

 keyspace_name | durable_writes | strategy_class                              | strategy_options
---------------+----------------+---------------------------------------------+----------------------------
          test |           True | org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"1"}
        system |           True |  org.apache.cassandra.locator.LocalStrategy |                         {}
 system_traces |           True | org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"2"}
          demo |           True | org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"1"}

(4 rows)

cqlsh:system>
```

  - Use the CQL *DESCRIBE KEYSPACE* command to list all System table schema

What tables are in the System keyspace, and why?

- *System* keyspace tables include

Table	Purpose
schema_keyspaces	Keyspaces available within this cluster, along with their assigned replication strategy and replication factor
schema_columns	Details of cluster compound primary key columns
schema_columnfamilies	Details of cluster tables and their configuration
peers	Local tracking of cluster-wide gossip for this node
local	Details of the local node's own state
hints	Stores information for hinted handoffs

- Other *System* keyspace tables

IndexInfo	schema_usertypes	batchlog	compaction_history
compactions_in_progress	paxos	peer_events	range_xfers
schema_triggers	sstable_activity		

Demo 5: Query and examine System tables

Summary

- A Cassandra *cluster* is comprised of peer-to-peer *nodes* logically organized into *racks* within *data centers*
- Any node may *coordinate* any *request* issued by a Cassandra *client*
- Data is organized into *partitions* ("rows") identified by *tokens* in an integer range
- The total *token range* is treated internally as a ring whose segments are owned by nodes
- Nodes are identified by the highest token in their segment of the total range
- A node's *partitioner* hashes a token from the *partition key* of a value being written
- The *first replica* ("copy") of a *partition* is written to the *node* owning *the primary range* containing its *token*
- The *Murmur3Partitioner* is the default and best practice
- *Virtual nodes* are multiple smaller token ring segments managed by a single machine
- Virtual nodes improve performance as nodes are *bootstrapped*, added, and removed
- A *keyspace* defines a cluster's *replication strategy* and *replication factor*
- *Replication factor* (RF) determines how many replicas ("copies") are made of each partition

Summary

- *Replication strategy* determines how replicas are distributed across the cluster
- A per-request *consistency level* (CL) determines how many nodes must acknowledge
- A *hinted handoff* temporarily stores requests issued to unavailable nodes
- *Consistency levels* include *ANY*, *ONE*, *QUORUM*, *LOCAL_QUORUM*, *EACH_QUORUM*, and *ALL*
- *Immediately consistent* data is guaranteed to be current
- Nodes with stale data are updated during each read request through *read repair*
- The *nodetool repair* command makes stale data consistent for a node or set of nodes
- IF *nodes_written* + *nodes_read* > *replication_factor* THEN results *are immediately consistent*
- *Eventually consistent* data may be a few milliseconds stale
- Node clocks must be in sync as the most recently timestamped data returns to the client
- Nodes continually exchange state and location information via the *Gossip* protocol
- Each node includes a *Snitch* which tracks and reports on the current cluster topology
- Cassandra tracks its internal state and structure in *System* keyspace tables

Review Questions

- Describe the relationship of nodes, racks, clusters, and data centers
- What is the function of the partitioner?
- Can a node hold a partition with a token outside its primary range?
- In a 3 node cluster with RF=2, how much total data volume does each node own?
- What is the function of the *nodetool repair* operation?
- What is a remote coordinator?
- How could RF and CL be tuned to ensure immediate consistency?

DATASTAX