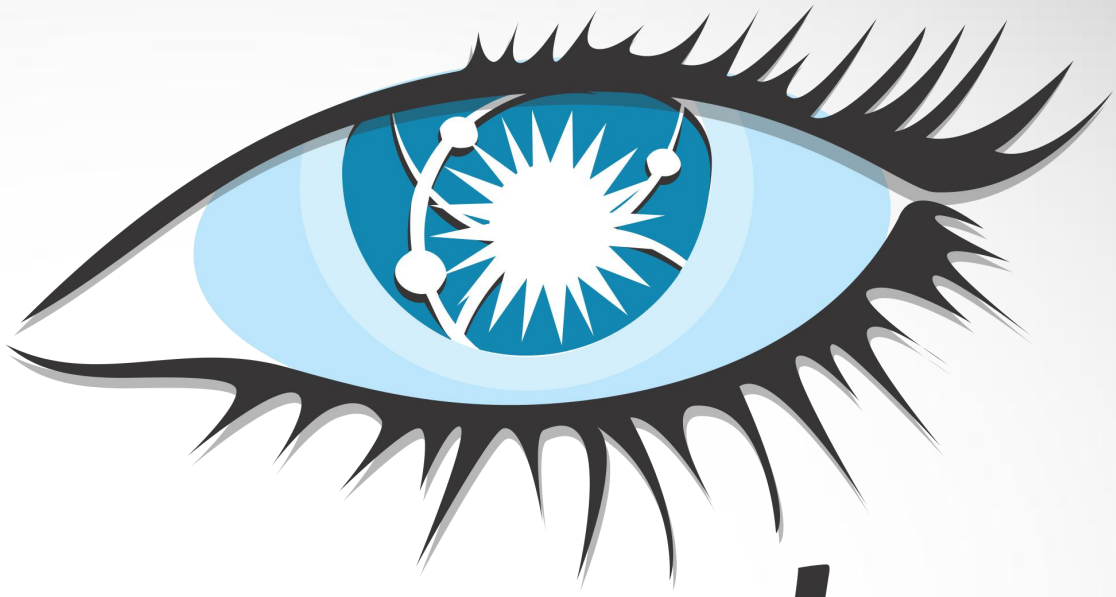


cassandra

By
Nirmallya Mukherjee

Certified Cassandra Developer
Certified Cassandra Administrator

Part 1

Warm up!

Types of NoSQL

- **Wide Row** - Also known as wide-column stores, these databases store data in rows and users are able to perform some query operations via column-based access. A wide-row store offers very high performance and a highly scalable architecture. Examples include: Cassandra, HBase, and Google BigTable.

- **Columnar** - Also known as column oriented store. Here the columns of all the rows are stored together on disk. A great fit for analytical queries because it reduces disk seek and encourages array like processing. Amazon Redshift, Google BigQuery, Teradata (with column partitioning).

- **Key/Value** - These NoSQL databases are some of the least complex as all of the data within consists of an indexed key and a value. Examples include Amazon DynamoDB, Riak, and Oracle NoSQL database

- **Document** - Expands on the basic idea of key-value stores where "documents" are more complex, in that they contain data and each document is assigned a unique key, which is used to retrieve the document. These are designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Examples include MongoDB and CouchDB

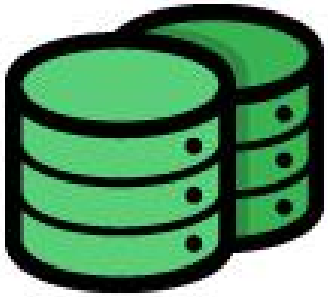
- **Graph** - Designed for data whose relationships are well represented as a graph structure and has elements that are interconnected; with an undetermined number of relationships between them. Examples include: Neo4J, OrientDB and TitanDB

Datomic

It can use any storage! Engine abstraction, you think?

Immutable data means strong consistency combined with horizontal **read** scalability, plus built-in caching.

Writes are not update-in-place; all data is retained by default. This provides built-in auditing and the ability to query history.



Datomic provides rich schema and query capabilities on top of modern scalable **storage** engines like DynamoDB, Cassandra, Riak and more.

SQL and NoSQL

1. Database, Relational, strict models
2. Data in rows, pre-defined schema, sql supports join
3. Vertically scalable
4. Random access pattern support
5. Good fit for online transactional systems
6. Master slave model
7. Periodic data replication as read only copies in slave
8. Availability model includes a slight downtime in case of outages

1. Datastore, distributed & Non relational
2. Data in key value pairs, flexible schema, no joins
3. Horizontally scalable
4. Designed for access patterns
5. Good for optimized read based system or high availability write systems
6. Seamless data replication
7. C* is masterless model
8. Masterless allows for no downtime

Applicability of SQL / NoSQL

- No one single rule and very usecase specific
  - High read oriented
  - High write oriented
  - Document based storage
  - KV based storage
- Important to know the access patterns upfront
  - Focus on modeling specific to the use case
- Very difficult to fix an improper model later on unlike a database

CAP Theorem

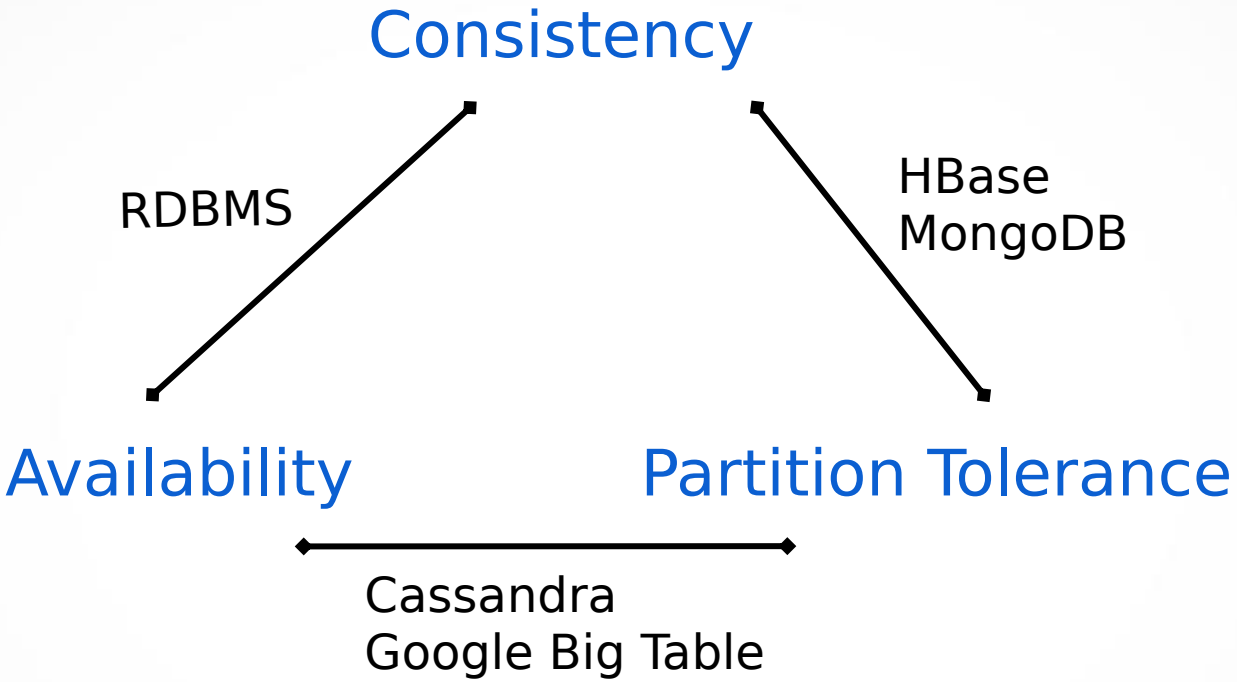
- ACID
  - **A**tomicity - Atomicity requires that each transaction be "all or nothing"
  - **C**onsistency - The consistency property ensures that any transaction will bring the database from one valid state to another
  - **I**solation - The isolation property ensures that the concurrent execution of transactions result in a system state that would be obtained if transactions were executed serially
  - **D**urability - Durability means that once a transaction has been committed, it will remain so under all circumstances

- What is it? Can I have all?
  - **C**onsistency - all replica nodes have the same data at all times
  - **A**vailability - Request must receive a response
  - **P**artition tolerance - Should run even if there is a part failure

- CAP theorem applies only to distributed systems only
  - https://en.wikipedia.org/wiki/CAP_theorem

- CAP leads to BASE theory
  - **B**asically **A**vailable, **S**oft state, **E**ventual consistency

AP systems, what about C?



How does this impact architecture?

1. Eventual consistency
2. Some application logic is needed

All this a bit later ...

Good fit use cases

- Always starts with V3
- Can handle data diversity
  - Sensor data
  - Online usage impressions
  - Playlists & collections
  - Personalization and recommendation engine
  - Orders, Bids, Inventory
  - Logs, Events
  - ...
- If there is data it can be put it in C* in an appropriate data model
- Scenario where the data can not be recreated, it is lost if not saved
- No need of rollups, aggregations (Another system needs to do this)
- Use case is !(ACID compliant with need for rollback)

http://www.datastax.com/resources/casestudies

Part 2

Concepts I

Database or Datastore

- Where did the name "Cassandra" come from?
  - Some interesting reference from Greek mythology
  - What's the significance of the logo?

- Symantics - no real definition, both are ok
- I would like to call it "Datastore"
- A bit of un-learning is required to get a hold of the "different" ways
- Inspiration from (think of it as object persistence)
  - Google BigTable
  - Dynamo DB from Amazon

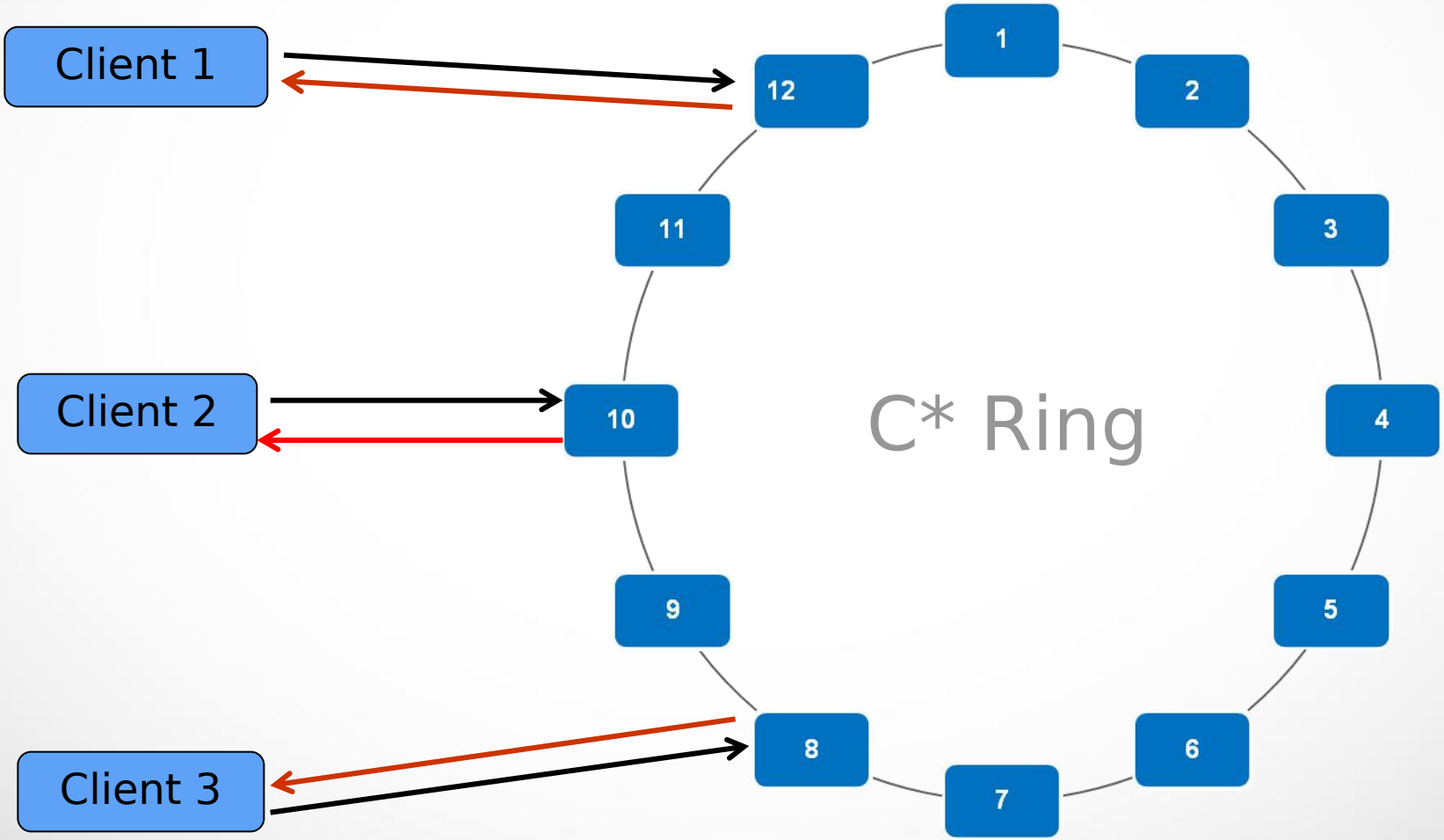
- A few interesting observations about datastore
  - Put = insert/update (Upsert?)
  - Get = select

- Think HashMaps, Sorted Maps ...
- Storage mechanism
  - Btree vs LSM Tree
- Row oriented datastore but grows horizontally. It is not "column oriented"!

- Biggest of all - No SPOF (Masterless), can you name another datastore that is masterless?

Masterless architecture

Why large malls have more than one entrance?



Seed node(s)

- It is like the recruitment department
- Helps a new node come on board (first step of the bootstrap process, comming up later...)
- Starts the gossip process in new nodes
- No other purpose
- Seed list can be specified in the c* yaml
- If you have more than one DC then include seed nodes from each DC
- This list need not be the same in all the nodes of the cluster especially across DC
- More than one seed is a very good practice
- Per rack based could be good, best practice is to have 3 nodes per DC

Gossip

- I think we all know what this means in English! ☺
- It is a bi-directional communication mechanism among nodes and a way any node learns about the cluster and other nodes
- At the time of boostrap 3 nodes are picked at random for the first time, this is done by the seed node because a new node does not know about the cluster yet
- Runs every second and talks to upto 1 to 3 nodes at random everytime
- Passing state information (it's own + others)
  - Available / down / bootstraping
  - Data "Load" it is under + "Severity" indicates I/O pressure
- Process
  - One node sends the digest to the other node about all the nodes gossip timestamp
  - Other node checks if the timestamp is latest (maybe the transmitting node has latest or itself may have the latest)
  - Receiving node sends the ack of the nodes for which it needs data and sends the data which it has as the latest
  - The sending node now accepts latest data from the receiving node and updates itself with any latest data from the receiving node
  - Transmit the other node data which the receiving node needs
- Communication among the nodes is not a guarantee, but that's not an issue because one node may learn about another node from more than 1 other node
- Once gossip has started on a node, the stats about other nodes are stored locally so that a re-start can be fast
- It is versioned, older states are erased
- Local gossip info can be cleaned if needed. Eg when an admin wants to reset the gossip state of a node (perhaps after bringing up a node that was unavailable). Start with -Dcassandra.load_ring_state=false option
- Helps in detecting failed "not responsive since" nodes (next ...)

Detecting a failed node

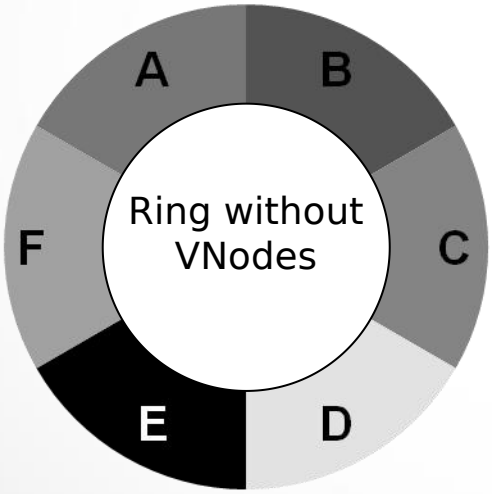
- C* uses a mechanism called "Accrual Failure Detector"
- The basic idea is that a node’s state is not necessarily up or down. What else? "Busy" maybe!
- It is an educated guess which takes multiple factors into account
- A server (node n1) suspects that a node is down because it hasn’t received the two last heartbeats from node n2
- n1 assigns a specific phi value to node n2 which denotes a level of "suspicion" that something could be wrong with n2
- Further lost heartbeats increases the phi value until it reaches a threshold
- When threshold > acceptable value the node is marked down
- The parameter is called "phi_convict_threshold" but it is always not required to be changed
- Higher the value the less chance of getting a false positive about a failure (Good range is between 5 and 12, default is 8)
- See "FailureDetector.java" class for details & JIRA CASSANDRA-2597

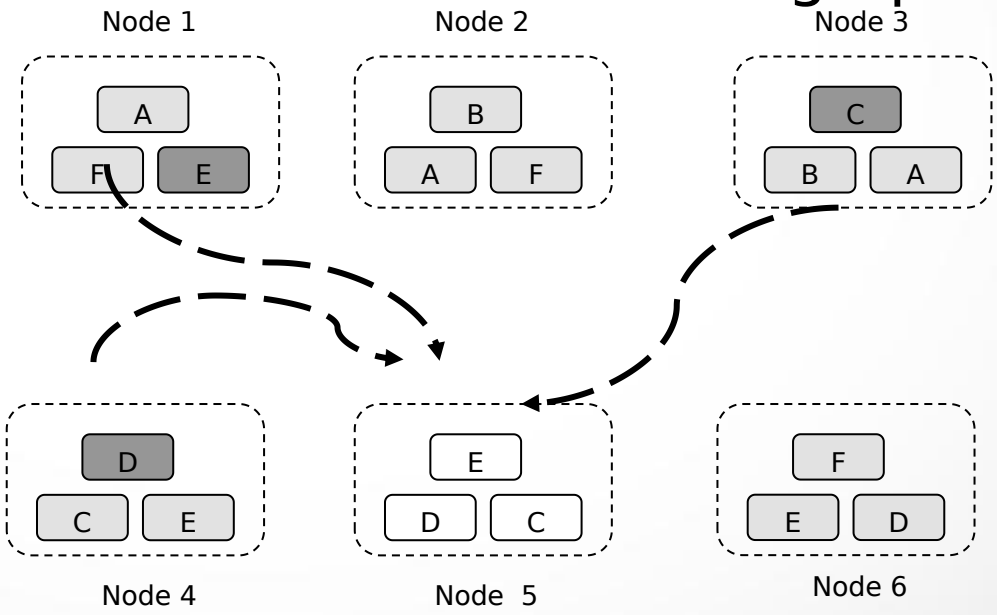
Partitioner

- How do you all organize your cubicle/office space? Where will your stuff be? Assume the floor you are on is about 25,000 sq ft.
- A partitioner determines how to distribute the data across the nodes in the cluster
- Murmur3Partitioner - recommended for most purposes (default strategy as well)
- Once a partitioner is set for a cluster, cannot be changed without data reload
- The partition/row key is hashed using the murmer hash to determine which node it needs to go
- There is nothing called as the "Master/Primary/Original/Gold replica", all replicas are identical - first/second/third ... replica
- The token range it can produce is -2^{63} to $2^{63} - 1$ (Range of Long, ROL)
- Wikipedia details
  - MurmurHash is a non-cryptographic hash function suitable for general hash-based lookup. It was created by Austin Appleby in 2008, and exists in a number of variants, all of which have been released into the public domain. When compared to other popular hash functions, MurmurHash performed well in a random distribution of regular keys.

Ring architecture - Nodes

- C* operates by dividing all data evenly around a cluster of nodes, which can be visualized as a ring
- In the early days the nodes had a range of tokens
- This had to be done at the time of setting up C*



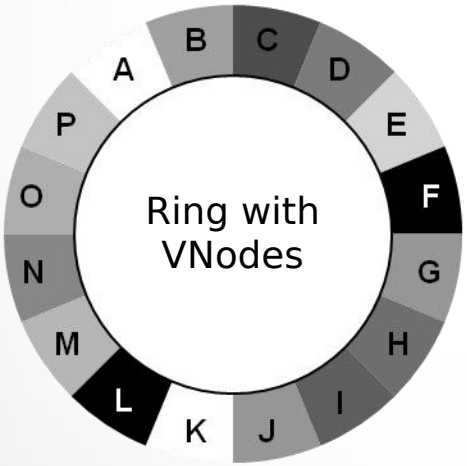


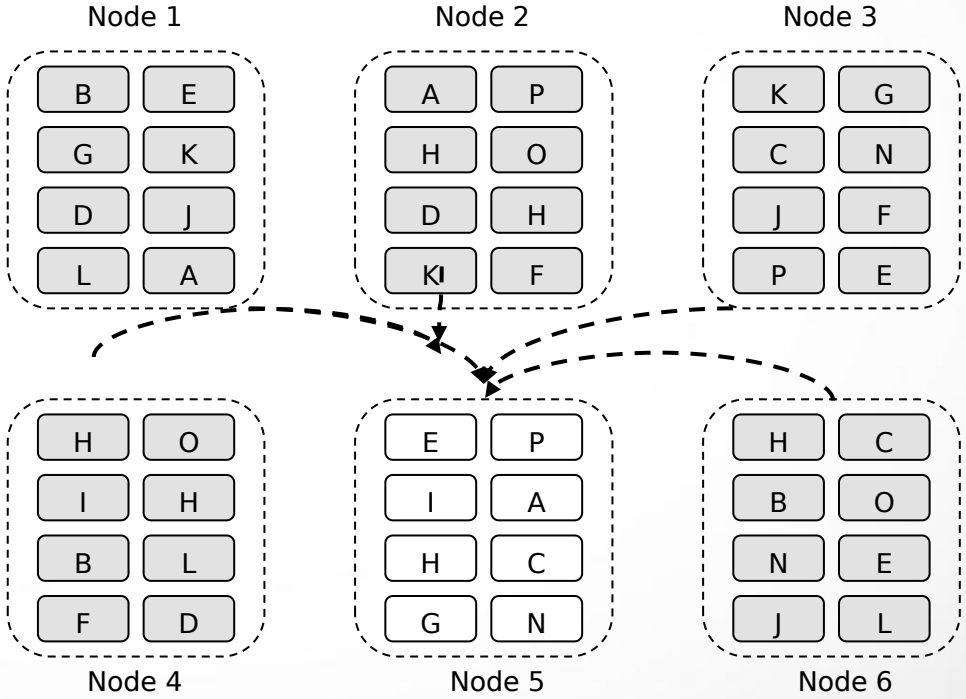
Few peers helping to bring a downed node up, Increased load on the selected peers

What will happen if a node with elivated utilization goes down?

Ring architecture - VNodes

- In the later versions vNode made things easy - one node with many ranges! (*One set of primary ranges*)
- Recovery from failure got better - small contributions from many nodes to help rebuilding
- Recovery got faster and load on other nodes reduced





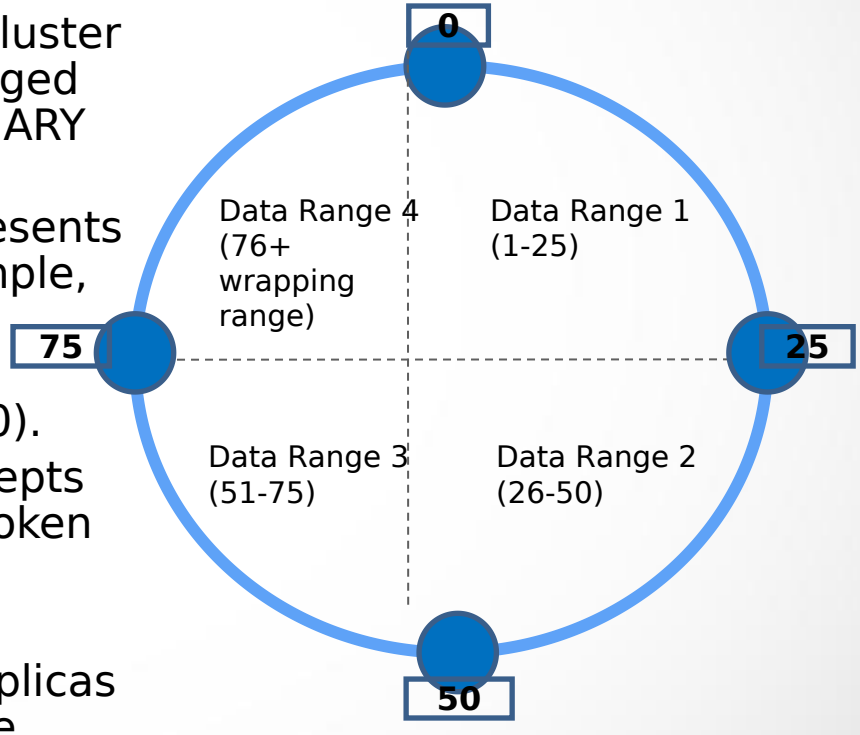
Many peers helping to bring a downed node up,
Marginal increase in load on the selected peers

Replication

- How many of you have multiple copies of the most critical files - pwd file, IT return confirmation etc on external drives?
- Replication determines how many copies of the data will be maintained in the cluster across nodes
- There is no single magic formula to determine the correct number
- The widely accepted number is 3 but your use case can be different
- This has an impact on the number of nodes in the cluster - cannot have a high replication with less nodes
  - Replication factor <= number of nodes
- Seamless synchronization of data across DC/DR
- In a DC/DR scenario using NetworkTopology strategy different replication can be specified per keyspace per DR/DC
  - strategy_options:{data-center-name}={rep-factor-value}
- Also has a performance impact during
  - "Insert/Update" using a particular consistency level
  - "Select" using a particular consistency level
- System tables are never replicated, they remain local to each node

Partitioner and Replication

- Position of the first copy of the data is determined by the partitioner and copies are placed by walking the cluster in a clockwise direction
- For example, consider a simple 4 node cluster where all of the partition/row keys managed by the cluster were numbers in the PRIMARY range of 0 to 100.
- Each node is assigned a token that represents a point in this range. In this simple example, the token values are 0, 25, 50, and 75.
- The first node, the one with token 0, is responsible for the wrapping range (76-0).
- The node with the lowest token also accepts partition/row keys less than the lowest token and more than the highest token.



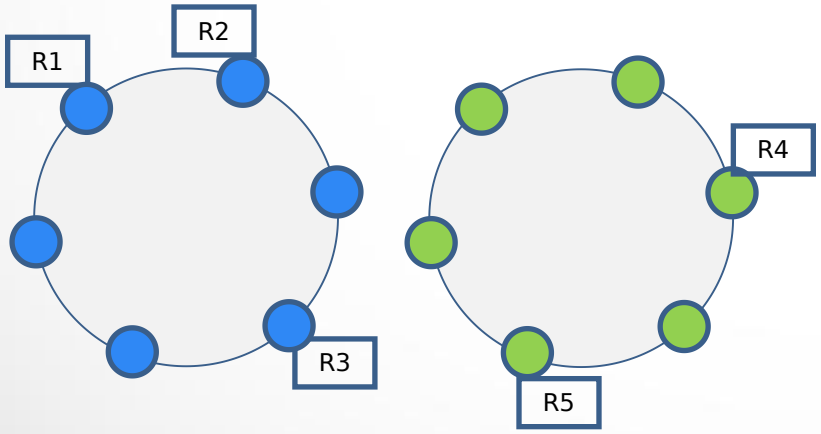
- Once the first node is determined the replicas will be placed in the nodes in a clockwise order

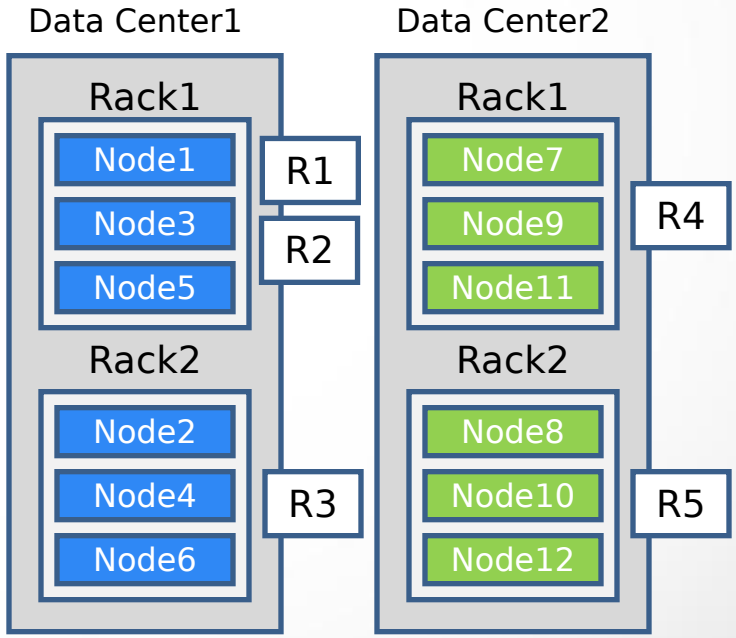
- The process is orchestrated by "Coordinator"
  .. a bit later

Multi DC replication

- When using NetworkToplogyStrategy, you set the number of replicas per data center
- For example if you set the replicas as 5 (3+2) then this is what you get ...

```
Create KEYSPACE stockdb WITH REPLICATION =
  {'class' : 'NetworkTopologyStrategy', 'DC1' : 3, 'DC2' : 2};

If altering RF, run nodetool repair on each affected node,
wait for it to come up and move on to the next node.
```





Snitch

- What can you typically find at the entrance of a very large mall? What's the need for "Can I help you?" desk?
- Informs the partitioner about the rack and DC locations - determines which nodes the replicas need to go
- Identify which DC and rack a node belongs to
- Think of this as the "Google map" for directions for the replication process
- "Tries" not to have more than one replica in the same rack (may not be a physical grouping)
- Routing requests efficiently
- Allows for a truly distributed fault tolerant cluster
- To be selected at the time of C* installation in C* yaml file
- Changing a snitch is a long drawn up process especially if data exists in the keyspace - you have to run a full repair in your cluster

Snitch

- 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
- 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
- YamlFileNetworkTopologySnitch (C* 2.1) – useful for mixed-cloud clusters, configured using *cassandra-topology.yaml*
- RackInferringSnitch – sample for writing a custom Snitch

In addition a "Google cloud snitch" is also available for GCE

Mixed cloud = cloud + local nodes

Snitch - property file example

```
# Data Center One
19.82.20.3=DC1:RAC1
19.83.123.233=DC1:RAC1
19.84.193.101=DC1:RAC1
19.85.13.6=DC1:RAC1

19.23.20.87=DC1:RAC2
19.15.16.200=DC1:RAC2
19.24.102.103=DC1:RAC2

# Data Center Two
53.25.29.124=DC2:RAC2
53.34.20.223=DC2:RAC2
53.14.14.209=DC2:RAC2

29.51.8.2=DC2:RAC1
29.50.10.21=DC2:RAC1
29.50.29.14=DC2:RAC1
```

The names like "DC1" and "DC2" are very important as we will see ...

Also, in production use GossipingPropertyFileSnitch (non cloud, for cloud use the appropriate snitch for that cloud) where you keep the entries for the rack+DC (cassandra-rackdc.properties) and C* will gossip this across the cluster automatically. C* topology properties is used as a fallback.

Snitch - property file

A bit more about property file snitch

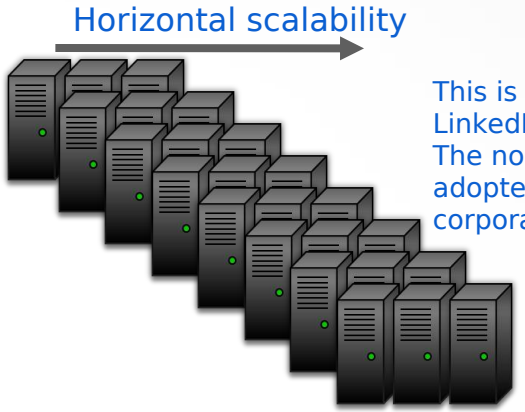
- All nodes in the system must have the same cassandra-topology.properties file
- It is very useful if you cannot ensure the IP octates (this prevents the use of Rack infering snitch where every octate means something) 10.<octat>.<>.<>
- Will give you full control of your cluster and the flexibilty of assigning any IP to a node (once assigned remains assigned)
- It can be hard to maintain information about a large cluster across multiple DC but it is worth given the benifits

Concepts - summary

- Seed node - like a recruitment department
- Gossip - share load information
- Partitioner - distribute data around the cluster based on Murmer3hash
- Replication - walk cluster clockwise, use snitch and try to avoid replicas on same rack
- Vnodes - split the token range (ROL) into smaller chunks. Auto calculation of the ranges
- Snitch - a yaml setting to help in positioning the replicas

Commodity vs Specialized hardware

vs



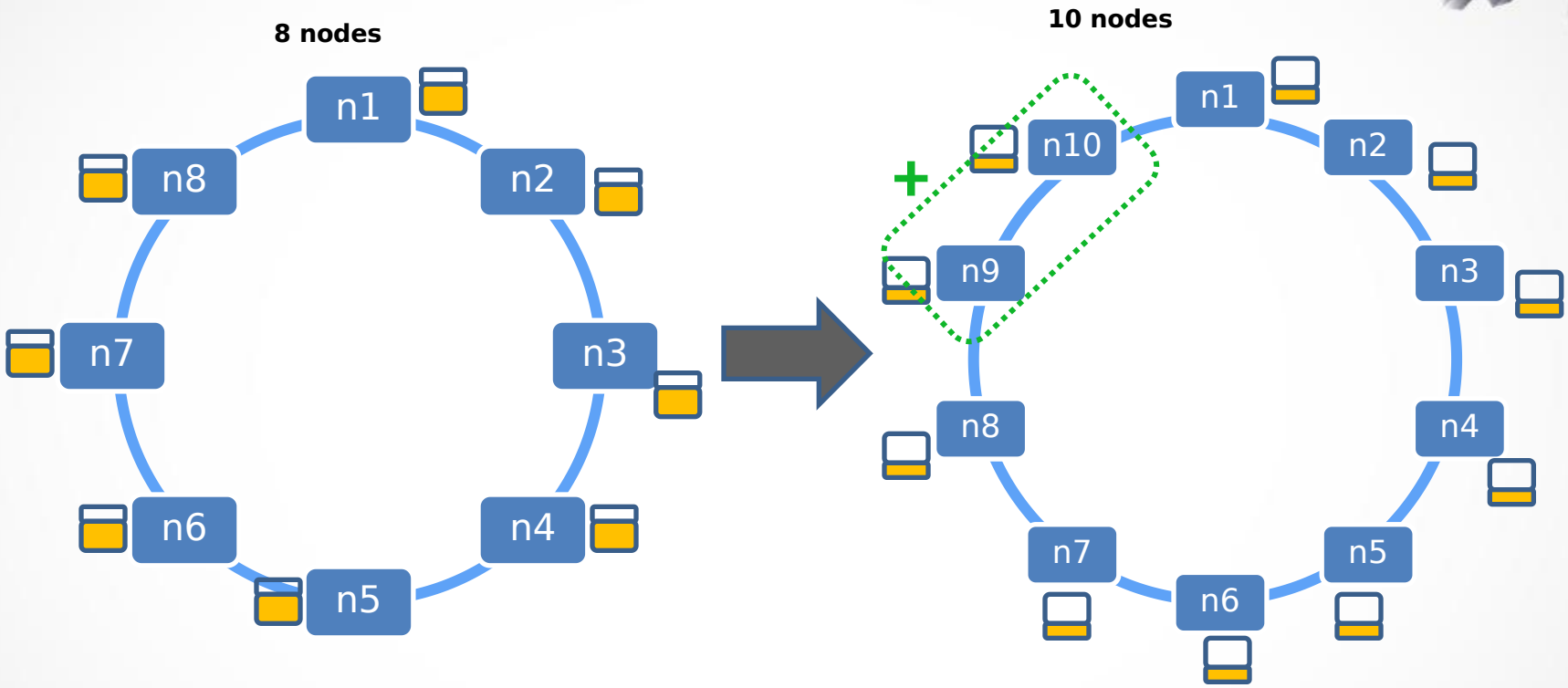
This is what Google, LinkedIn, Facebook do. The norm is now being adopted by large corporations as well.

1. Large CAPEX
2. Wasted/Idle resource
3. Failure takes out a large chunk
4. Expensive redundancy model
5. One shoe fitting all model
6. Too much co-existence

1. Low CAPEX (rent on IaaS)
2. Maximum resource utilization
3. Failure takes out a small chunk
4. Inexpensive redundancy
5. Specific h/w for specific tasks
6. Very less to no co-existence

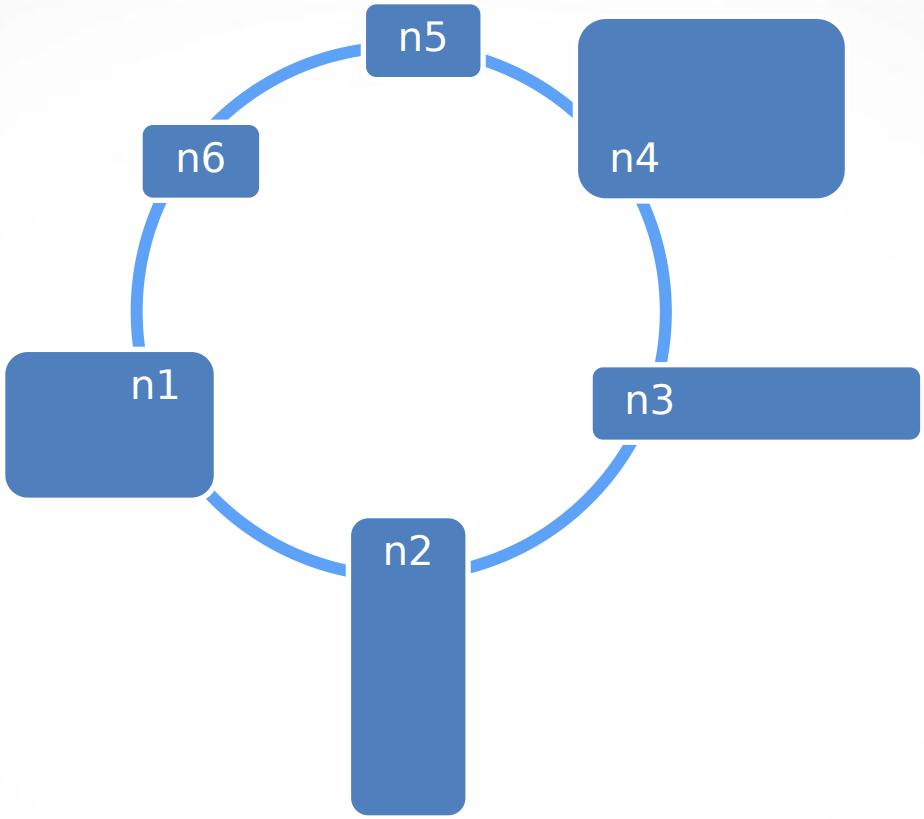
“Just in time” expansion; stay in tune with the load. No need to build ahead in time in anticipation

Elastic Linear Scalability



1.Need more capacity? Add servers
2.Auto Re-distribution of the cluster
3.Results in linear performance increase
4.This is also referred to as reduction of "Data density"

Debate - what if all machines are not similar?

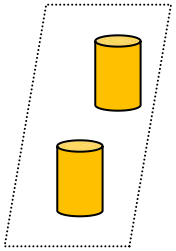


State your observations based on typical machine characteristics
1. CPU, Memory
3. Storage space (consider externally mounted space as local)
3. Network connectivity
4. Query performance and "predictability" of queries

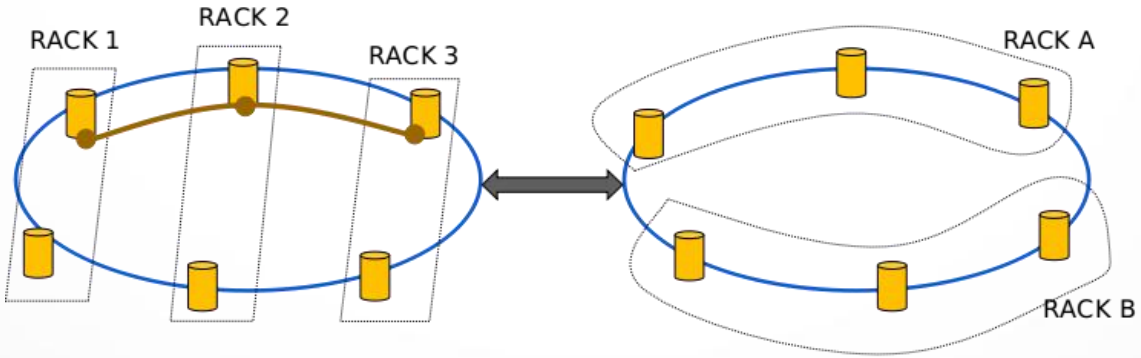
Deployment - 4 dimensions

1. Node - One C* instance

2. Rack - Logical set of nodes



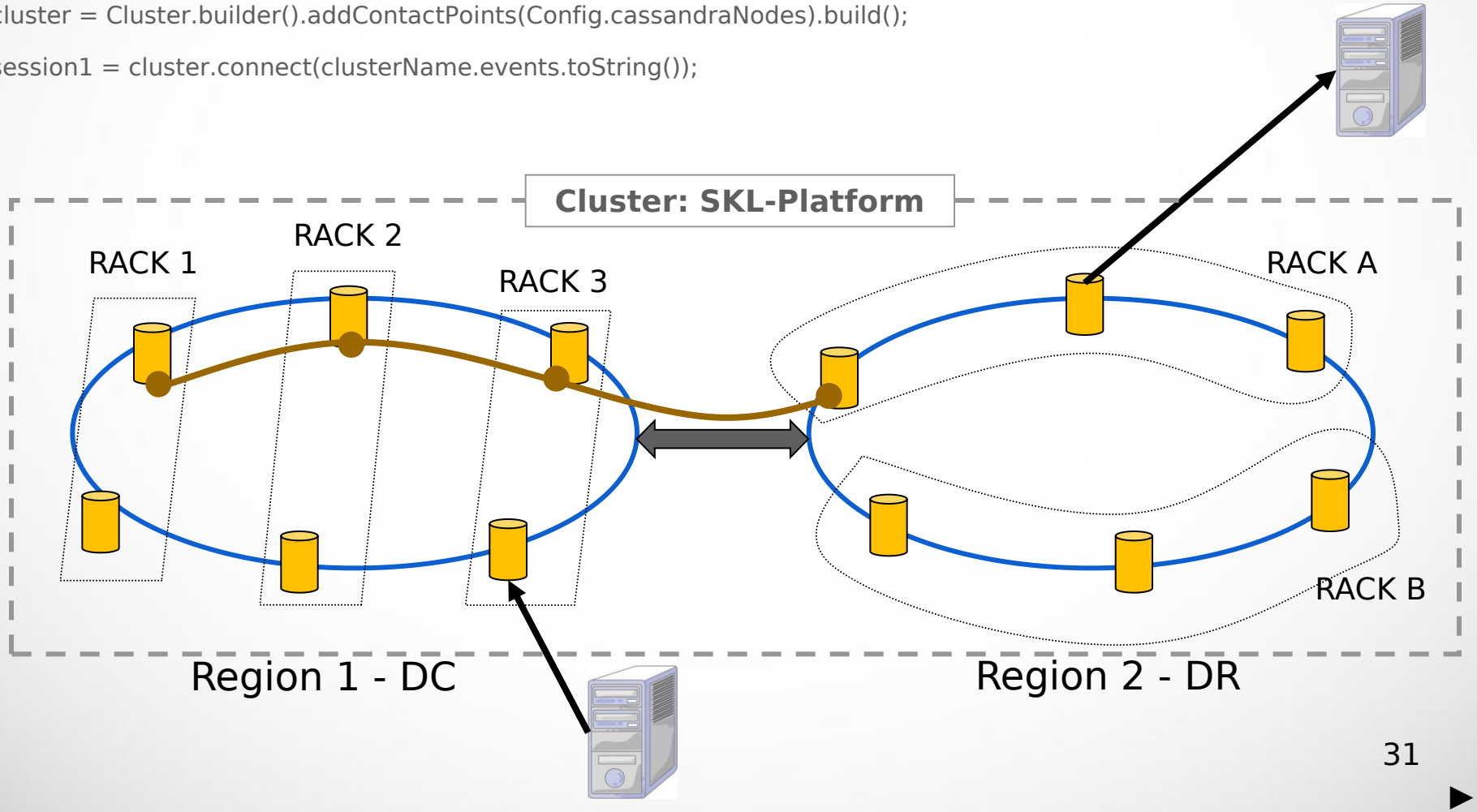
3. DC-DR - Logical set of racks



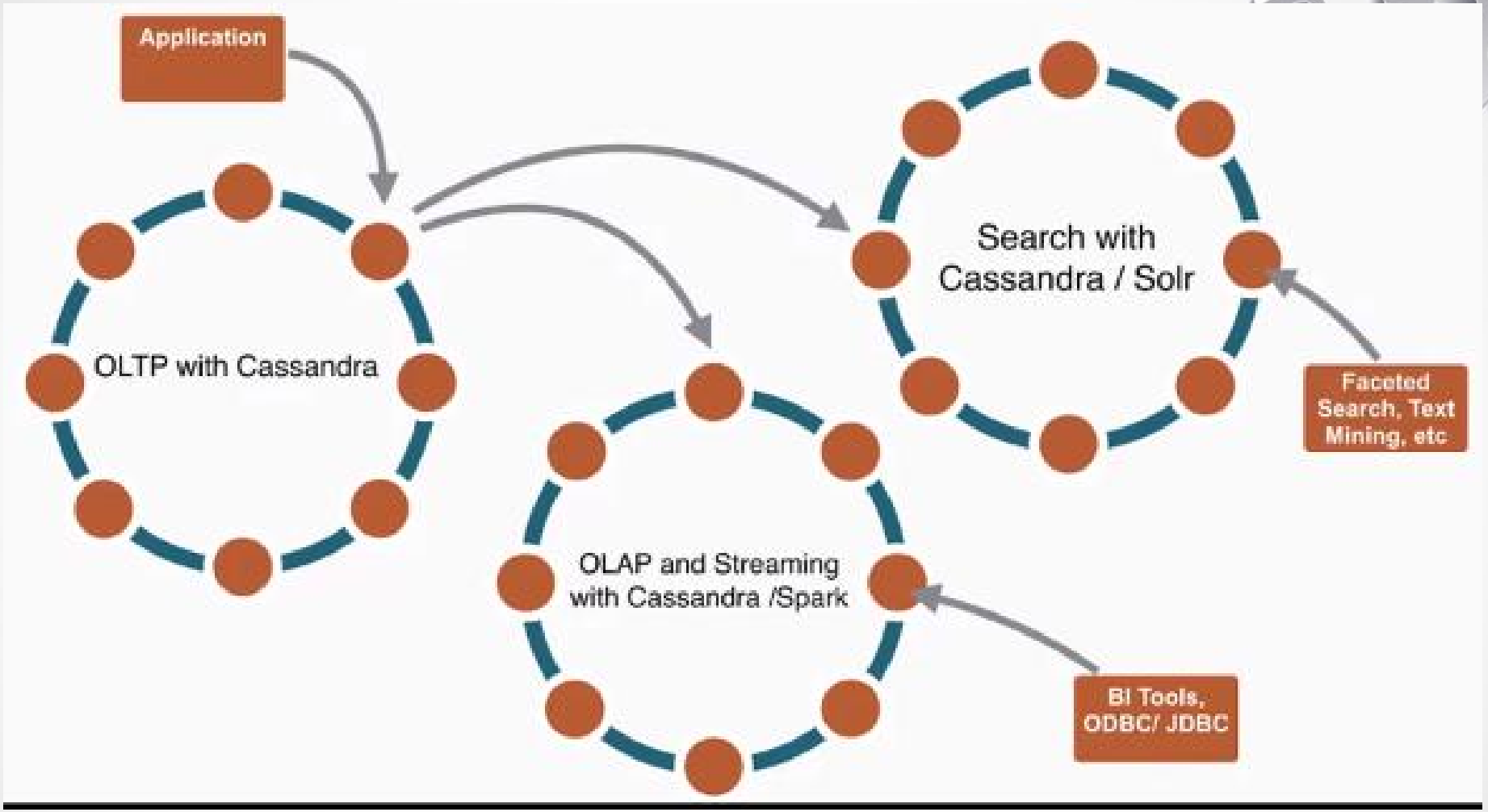
4. Cluster - Nodes that map to a single token ring (can be across DC), next slide ...

Distributed deployment

```
//These are the C* nodes that the DAO will look to connect to
public static final String[] cassandraNodes = { "10.24.37.1", "10.24.37.2", "10.24.37.3" };

public enum clusterName { events, counter }

//You can build with policies like withRetryPolicy(), .withLoadBalancingPolicy(Round Robin)
cluster = Cluster.builder().addContactPoints(Config.cassandraNodes).build();

session1 = cluster.connect(clusterName.events.toString());
```



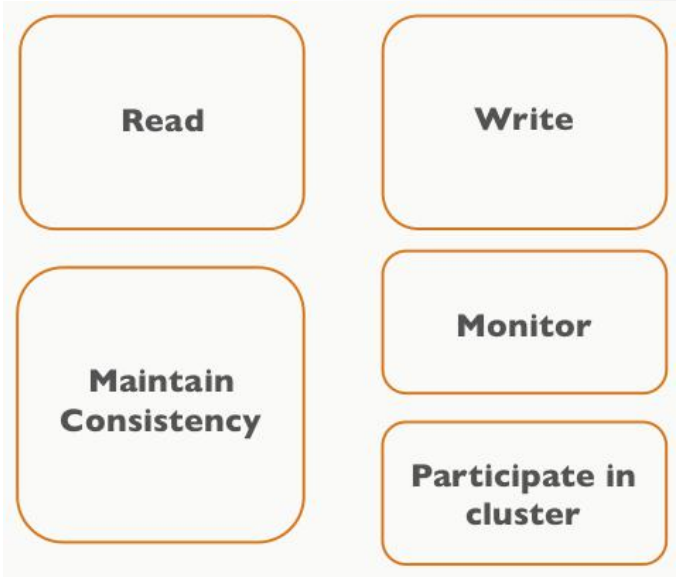
Distributed workloads



Each ring can be in a different region/availability zone

SEDA Architecture

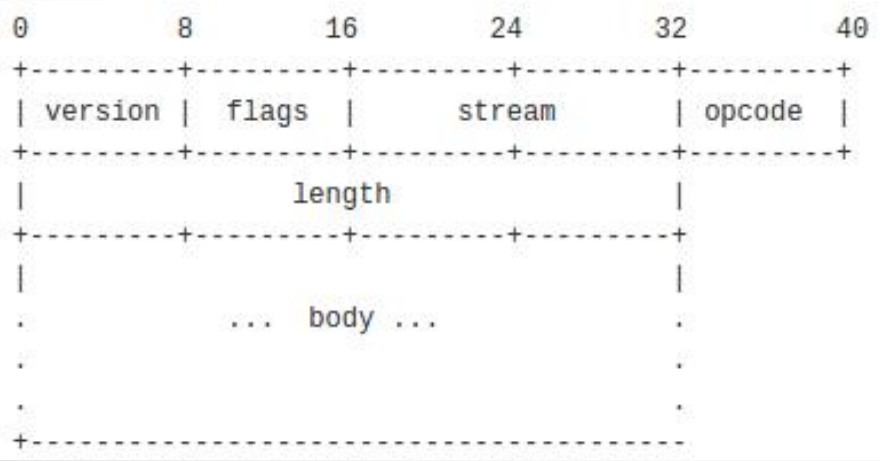
- Staged Event Driven Architecture (SEDA)
- Separates different tasks (gossip, read repair, replication etc) into stages that are connected by a messaging service
- Each like task is grouped into a stage having a queue and thread pool
- So what all pools do we have? Five. The size of the box matters, represents a number of stages and pools each has!
- Messages can get blocked. Why? Node is loaded more than it can handle. Two choices -
  - Continue to process and be overwhelmed
  - Drop messages and let C* pick another better performing node (good option!)
- The rest of the session will focus on these 5 areas



SEDA causes lots of context switches, future versions may have a single thread handle complete process like full read path

How to communicate with C*?

- CQL Binary protocol - in short just "CQL"
- This is used by the clients to talk to C*
- This is a frame based protocol
- This protocol has replaced the older Thrift protocol
- More details here (https://github.com/apache/cassandra/blob/cassandra-2.2.0/doc/native_protocol_v4.spec#L812)



Each frame contains a fixed size header (9 bytes) followed by a variable size body.

Part 3

Setup & Installation

Let's get going! Installation

- JDK 8 for v3.x is required (JAVA_HOME needs to be set & in path)
- Python 2.7 is required (put in the path)
- Acquiring and Installing C*
  - Understand the key components of Cassandra.yaml
- Configuring and Installation structure
  - Data directory
  - Commit Log directory
  - Cache directory
  - Hints directory
- System log configuration
  - Cassandra-env.ps1 (change the log file position)
  - Default is $logdir = "$env:CASSANDRA_HOME\logs"
  - In logback.xml change from .zip to .gz
    - <fileNamePattern>${cassandra.logdir}/system.log.%i.gz</fileNamePattern>
- Create a single node cluster on your machine!

Cassandra.yaml - an introduction

- cluster_name *(default: 'Test Cluster')*
  - All *nodes* in a *cluster* must have the same value.
- listen_address *(default: localhost)*
  - IP address or hostname other nodes use to connect to this *node*
- 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)*
  - Storage directory for data tables (*SSTables*)
- saved_caches_directory *(default: /var/lib/cassandra/saved_caches)*
  - Storage directory *for key caches* and *row caches*
- rpc_address / rpc_port *(default: localhost / 9160)*
  - listen address / port for *Thrift* client connections
- native_transport_port *(default: 9042)*
  - listen address for *Native CQL Driver* binary protocol

* hints_directory: /var/lib/cassandra/hints
  - This is where the hints will be stored in segments
* disk_optimization_strategy: ssd (default, other is spinning)

There is another undocumented parameter called auto_bootstrap: [true/false]
This can start C* node but will not start the bootstrap process until you explicitly join using nodetool
There are more configuration parameters, but we will cover those as we move along ...

Cassandra-env.sh

- JVM Heap Size settings
  - MAX_HEAP_SIZE="*value*"
    - Maximum recommended in 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

HEAP_NEWSIZE = The size of the young generation
Set NEW_HEAP to 100 MB per CPU core

Firewall and ports

- Cassandra primarily operates using these ports
  - Client connections (driver, cqlsh, devcenter)
    - 9042 (cql3 / native)
    - 9160 (old depricated thrift)
  - 7000 (or 7001 if SSL) - internode cluster communication
  - 7199 - JMX port
- Firewall must have these ports open for each node in the cluster/DC to operate internally, do not open for the world!

Different NIC setup

- Consider having a super fast network between nodes and not so fast to the client (great to have that too!)
- listen_address : NIC1 (inter node)
- rpc_address : NIC2 (client conn)
- where NIC1 speed >> NIC2 speed
- and NIC1 speed is gigabit or better

Nodetool important commands

- Options for nodetool [-h <IP> -p <PORT>] eg -h 10.21.24.11 -p 7199
  - info
  - describecluster
  - status
  - ring
  - gossipinfo
  - tpstats (see dropped mutations etc)
  - tablestats <keyspace>.<table> (read/write latency, sstable count, cell count)
  - tablehistograms ks table (recent statistics, gets cleared)
  - proxyhistograms (r/w latency recorded by the coordinator, internode comm latency)
  - netstats
  - compactionstats
  - snapshot mykeyspace
  - cleanup [run this after adding/removing nodes, removes data that is not owned by a node]
  - drain [flushes the memtables and cleans the commit logs]

- Get the list of SST that have a given PK
  - ./nodetool getsstables meterdb meter_data m_1:2015:3:87

- Get a dump of the PK and tokens for a given KS (see next slide for sample output)
  - ./nodetool describering meterdb > describering_meterdb.txt

- How to find out which nodes own my data?
  - ./nodetool getendpoints <keyspace> <table> <composite partition key>
    Can also tell about data that does not exist - it tells IF it were to exist which nodes it will go!
  - ./nodetool getendpoints meterdb meter_data m1:2015:3:87

- If node is down and you want to remove the node then (when up use decommission)
  - ./nodetool removenode

- ./sstablemetadata /mnt/disk-101/cassandra/data/meterdb/meter_data-4af50480d55711e487e689028f1b0dc9/meterdb-meter_data-ka-235-Data.db

- ./sstable2json /mnt/disk-101/cassandra/data/meterdb/meter_data-4af50480d55711e487e689028f1b0dc9/meterdb-meter_data-ka-235-Data.db > meter_data_json.txt

Describe ring - sample o/p

./nodetool describering meterdb > describering_meterdb.txt

...

TokenRange(
start_token:8413097228790211419, end_token:8413364087732140475,

endpoints:[10.240.72.75, 10.240.119.61, 10.240.123.124],

rpc_endpoints:[10.240.72.75, 10.240.119.61, 10.240.123.124],

endpoint_details:[
    EndpointDetails(host:10.240.72.124, datacenter:DC1, rack:RAC2),
    EndpointDetails(host:10.240.123.75, datacenter:DC1, rack:RAC3),
    EndpointDetails(host:10.240.119.61, datacenter:DC2, rack:RAC2)]
)

...

CQL shell - cqlsh

- Command line tool
  - ./cqlsh `hostname -I` -u cassandra -p cassandra

  In case of 3.x protocol error add the cqlversion (could be because current cqlsh is not yet ready for the next version)
  - ./cqlsh `hostname -I` -u cassandra -p cassandra --cqlversion=3.3.0
- Specify the host and port (none means localhost)
- Provides tab finish
- Supports two different command sets
  - DDL
  - Shell commands (describe, source, tracing, copy etc)
- Try some commands
  - show version
  - show host
  - describe keyspaces
  - describe keyspace <name>
  - describe tables
  - describe table <name>

CQL shell "source" command - loading a few million rows
"sstableloader" which is another tool - beyond a few million rows

Admin/System keyspaces

- SELECT * FROM
  - system_schema.keyspaces
  - system_schema.columnfamilies
  - system_schema.columns
  - system.local (info about itself)
  - system.<many others>
  - system.peers

- Local contains information about nodes (superset of gossip)

Authentication in C*

- CQL supports creating users and granting them access to tables etc..
- You need to enable authentication in the cassandra.yaml config file
  - authenticator: PasswordAuthenticator
  - authorizer: CassandraAuthorizer
  - role_manager: CassandraRoleManager
- Best practice is to increase the replication of the auth table (Important link below)
  - http://docs.datastax.com/en/archived/datastax_enterprise/4.0/datastax_enterprise/sec/secConfSysAuthKeyspRepl.html

- The SU is cassandra / cassandra
- When C* starts it will create system keyspaces and the SU
- It is recommended to increase the replication of system_auth_keyspace to the number of nodes so that C* does not have to ask other nodes for A&A, instead it can do all A&A with data that it has locally

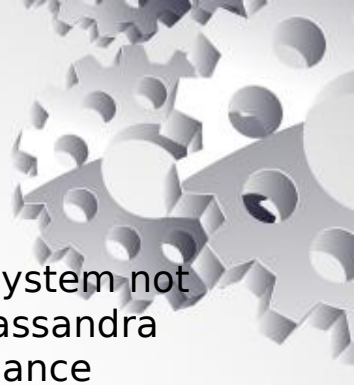
- Connect to C* using cqlsh as
  - ./cqlsh -u cassandra -p cassandra
- Can create, alter, drop and list users
  - http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/create_user_r.html
- Can then GRANT permissions to users accordingly – ALTER, AUTHORIZE, DROP, MODIFY,SELECT to users
  - http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/grant_r.html

- Creating roles is possible in C* 2.2+, roles are given to users
  - http://www.datastax.com/dev/blog/role-based-access-control-in-cassandra
- Enables authentication from the clients cqlsh and drivers

Setup - OS Tunings

- Latency is disk->Mem->CPU
- Get more memory, JVM will use 8GB but the rest is used by the OS as buffer cache where it will bring a lot of data from the disk to the main mem
- Way to size
  - Ops/sec = (C * nodes * cores) / Replication factor
  - C is hardware constant
  - Ops/sec means number of columns written / sec
  - Some results are 3000 AWS non SSD, will get a 50% boost if Ubuntu/Kernel 3.0 and C* version 2+
- Do not use shared drive/storage

Setup - file system tuning



- noatime. This stands for “no access time” updates. It tells the operating system not to update the inode information on each access. This is useful because Cassandra doesn’t access inode information and there is no need to pay the performance penalty for updating it.

- data = journal, commit = 15 By default, the operating system will sync all data (including metadata) to disk every five seconds. By extending this value, you can improve performance. commit applies only if you set data=journal

- barriers = 0 This option, when set to 0, disables write barriers. Write barriers enforce the proper on-disk order of journal commits. Since ext4 is a journal-based file system, you incur a performance penalty when ensuring the order of the journals. With hardware-based systems, it is safe to disable barriers if you have battery backed disks (as is common to hardware RAIDs).

- data = writeback, nobh When you set data=writeback, metadata for files is lazily written after the file is written. While this won’t cause file system corruption, it may cause the most recent changes to be lost in the event of a crash. Since most files in Cassandra are written on and not edited in place, this is a low-risk change. nobh (no buffer heads) refers to the operating system’s attempt to avoid associating buffer heads to data pages (ok because C* does not update in place)

Setup - OS tunings

- Warm up the buffer cache when the OS boots
  - find /var/lib/cassandra -name '*.db' -exec cat {} > /dev/null \;
- Disable NUMA zone reclaim
  - Each CPU looks to read from its own mem allocated to each socket
  - Disabling allows for reading from other CPUs mem hence hits more buffer cache
- In addition to changing the sysctl setting, you will need to raise the open file limit for the system. By adding the two lines below to your /etc/security/limits.conf, you should be able to give Cassandra enough room to operate under normal and heavy loads
  - * soft nofile 16,384
  - * hard nofile 32,768

Setup - OS Tunings for SSD

- The OS is not designed for SSD and does not know you have SSD
- It is setup for mechanical disks
- Need to tell the OS that you have SSDs
- Need to tune SSD read ahead - no real need for read ahead
  - sudo blockdev --report
  - sudo blockdev --setra 8 /dev/xvd
  - Set read ahead to 8 you can see the SSD seek time drop from 6ms to 200µs

Setup - JBOD, 3.2+ enhancements

- http://www.datastax.com/dev/blog/improving-jbod
- One disk evicts tombstones and other does not then data comes back
- To solve above "no splitting ranges" across data directories
- No compaction across data diretories, it will be disk local - consider SST from one disk only and not across disks
- Flushing is one thread per disk
- Backup disks individually, earlier all together

Setup - Useful resources

- Single "go to" place for all your C* needs
  - http://docs.datastax.com/en/cassandra/3.x/index.html
- http://docs.datastax.com/en/landing_page/doc/landing_page/planning/planningHardware.html
- http://docs.datastax.com/en/landing_page/doc/landing_page/recommendedSettingsLinux.html
- http://docs.datastax.com/en/cassandra/3.x/cassandra/operations/opsTuneJVM.html
- http://docs.datastax.com/en/landing_page/doc/landing_page/planning/planningEC2.html
- Do not use SATA disks! SAS are better (http://www.webopedia.com/DidYouKnow/Computer_Science/sas_sata.asp)

Part 4

Concepts II

Keyspace aka Schema

It is like the database (MySQL) or schema/user (Oracle)

Cassandra	Database
create keyspace [IF NOT EXISTS] meterdata with replication strategy (optionally with DC/DR setup), durable writes (True/False)	create database meterdata;

durable writes = false bypasses the commit log. You can lose data!

Table (older nomenclature CF)

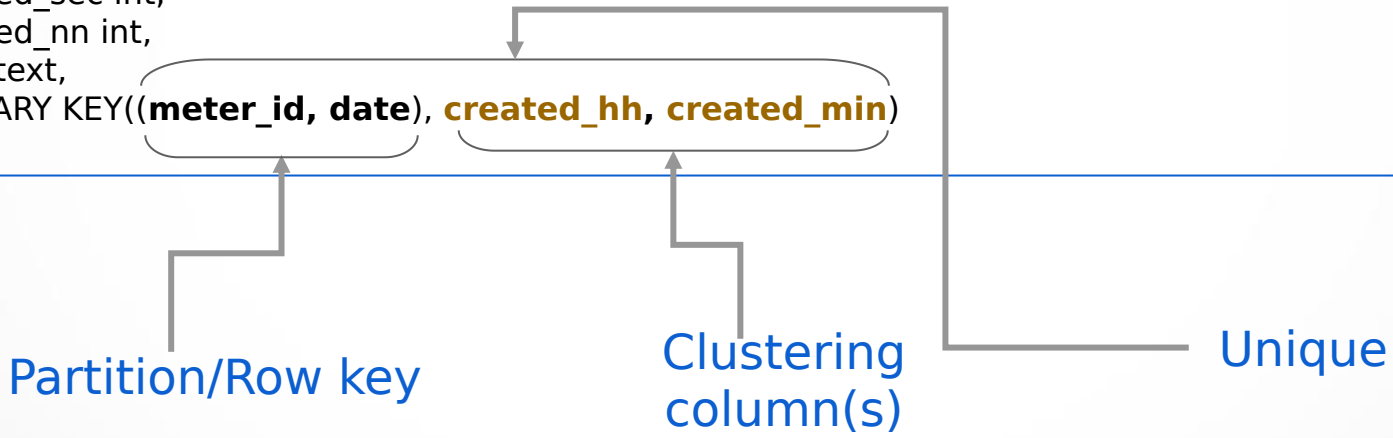
Cassandra	Database
create table meterdata.bill_data (...) primary key (compound key) with compaction, clustering order Primary Key is mandatory in C*	create table meterdata.bill_date (...) pk, references, engine, charset etc
Insert into bill_data () values (); Update bill_data set=.. where .. Delete from bill_data where ..	Standard SQL DML statements

What happens if we insert with a primary key that already exists?
Hold on to your thoughts ...

Visualizing the Primary Key

What are the components of Primary key?

```
CREATE TABLE meter_data (
  meter_id text,
  date int,    //format as yyyymmdd number (more efficient)
  created_hh int,
  created_min int,
  created_sec int,
  created_nn int,
  data text,
  PRIMARY KEY((meter_id, date), created_hh, created_min)
);
```

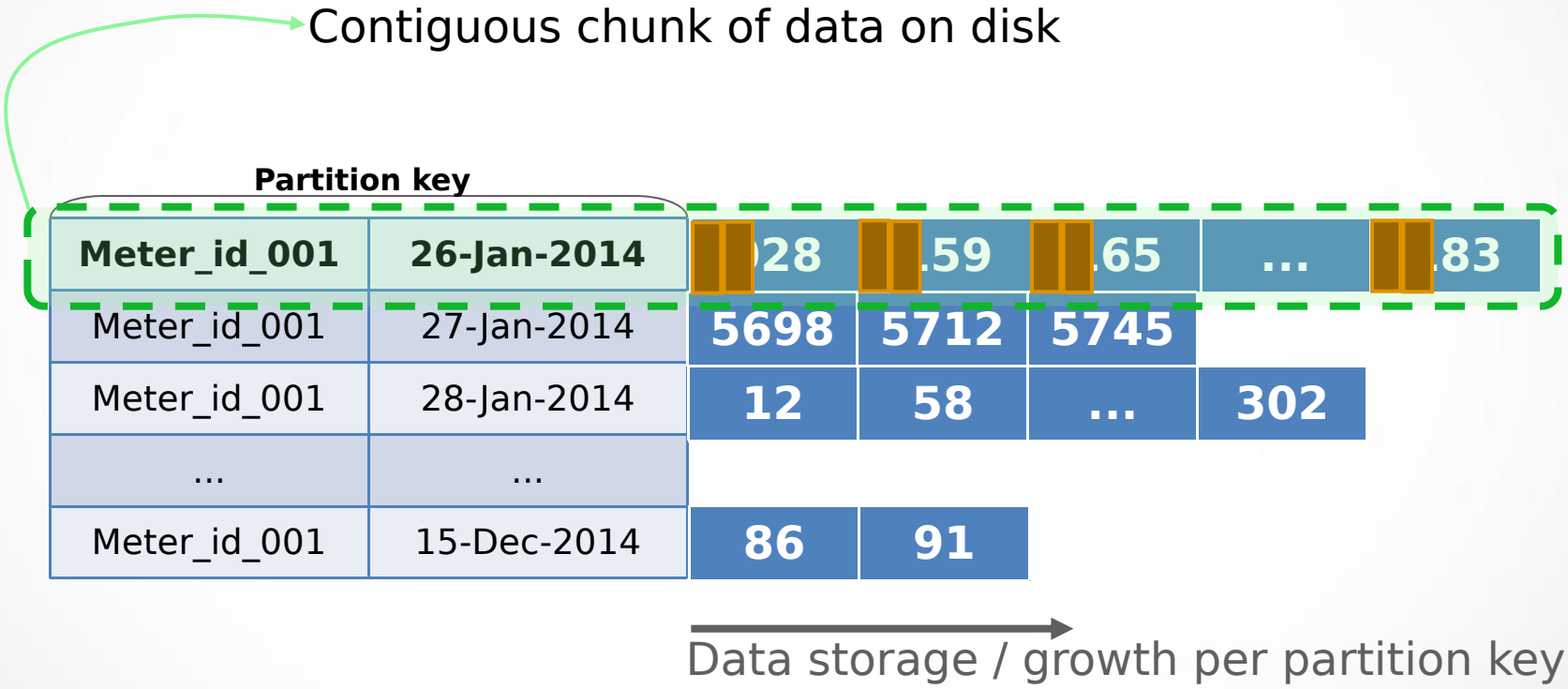


Primary key = Partition/Row key columns(s) + Clustering column(s)

Naturally by definition the primary key must be unique

Visualizing PK based storage

- Question - how does C* keep the data based on the partition/row key definition of a given table?



Note: C* creates a **key/name** using the clustering column values with regular column names and then stores the **value** of the regular column as a "**cell**". Visuals on next slide...

The brown boxes indicate the clustering column values being part of the key of the cell that contains the meter data.

ANOTHER note: the internal representation has undergone a change in version 3.0 (Rows concept)

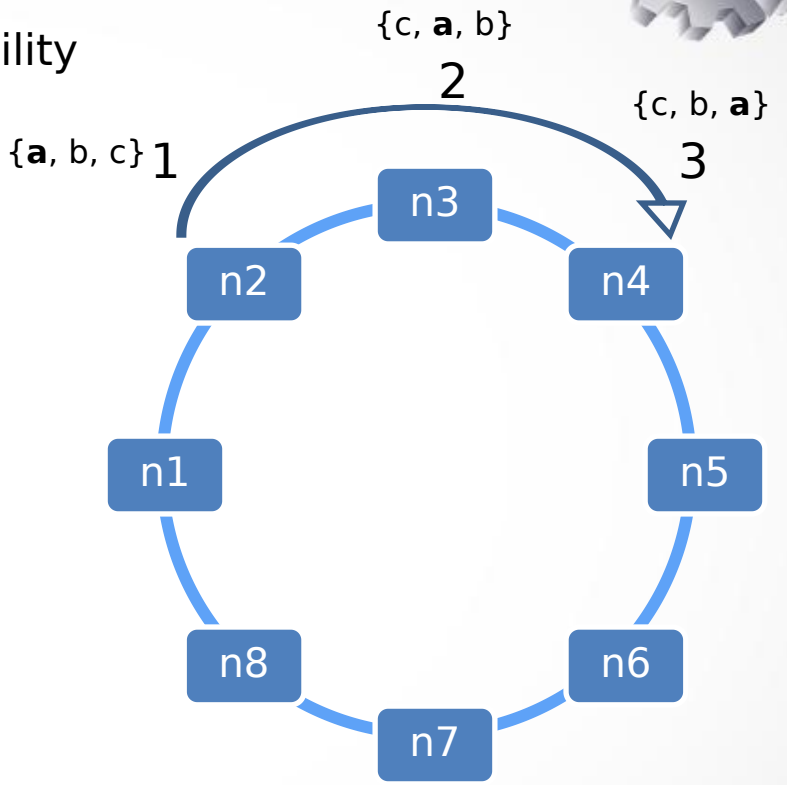
Failure/Partition Tolerance By Replication

Remember CAP theorem? C* gives availability and partition tolerance

Quick Q: Availability is achieved by?

Assume Replication Factor (RF) = 3

Let's insert a record with Pk=a




```
private static final String allEventCql = "insert into all_events (event_type, date, created_hh, created_min,
                                                created_sec, created_nn, data) values(?, ?, ?, ?, ?, ?, ?)";
Session session = CassandraDAO.getEventSession();
BoundStatement boundStatement = getWritableStatement(session, allEventCql);
session.execute(boundStatement.bind(.., .., ..);
```

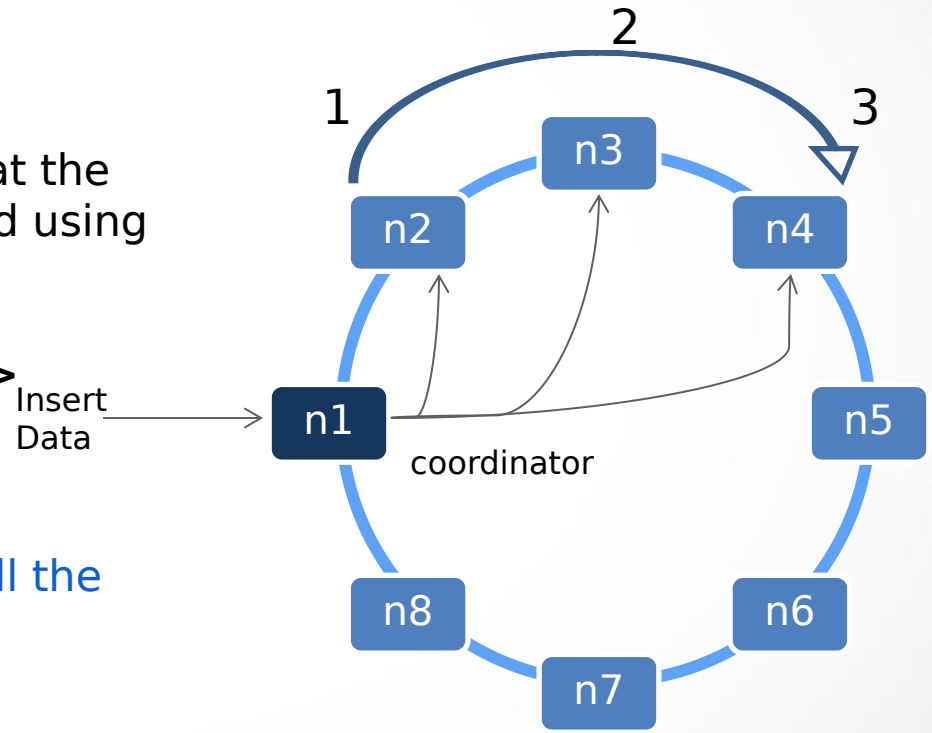
Coordinator Node - Single DC replication

Incoming Requests (read/Write)

Coordinator handles the request

An interesting aspect to note is that the data (each column) is timestamped using the coordinator node's time

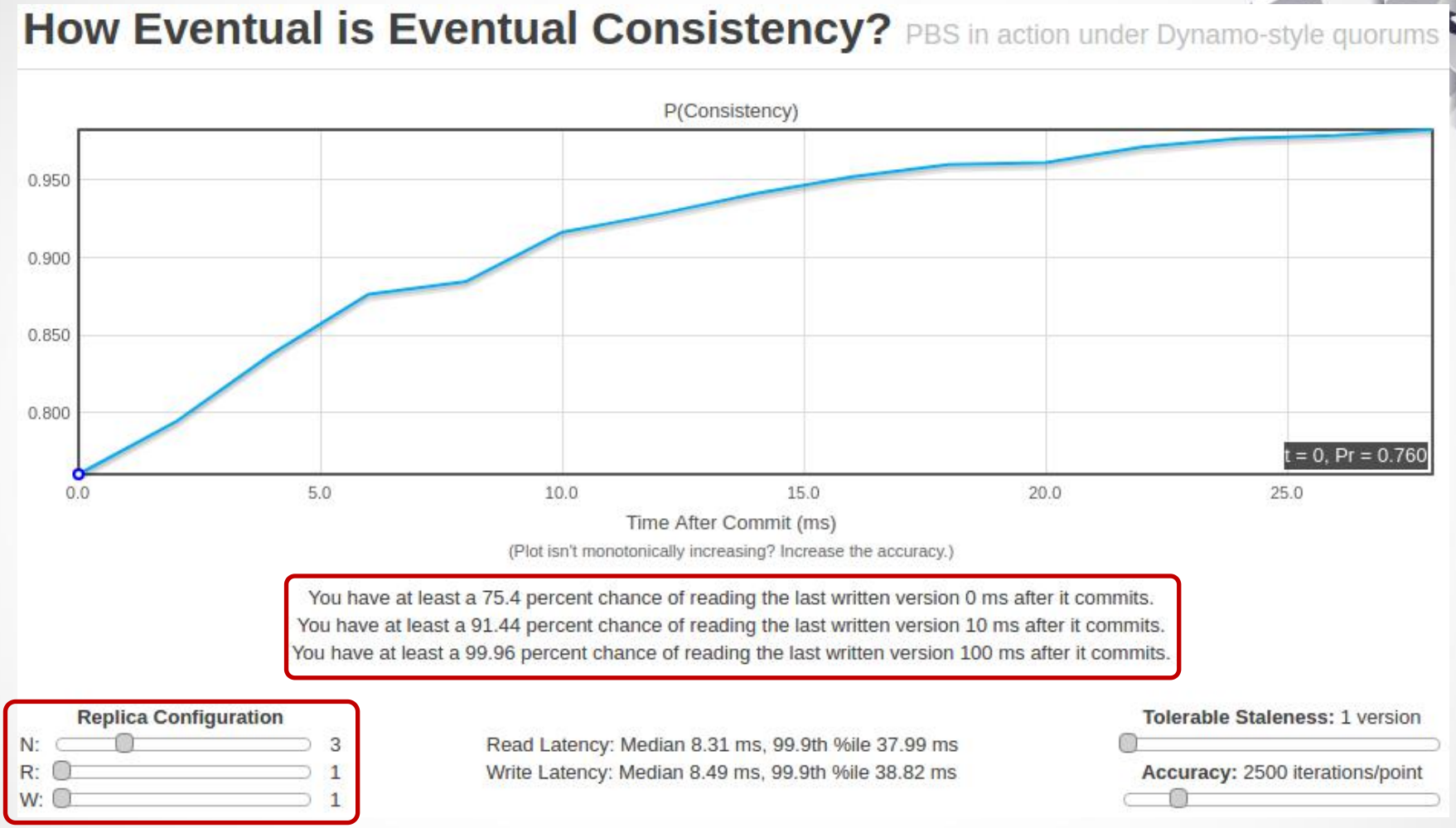
Every node can be **coordinator -> masterless**



You can see the coordinator and all the nodes it connects to in the new DevCenter "Query Trace" tab!

Hinted handoff a bit later ...

How eventual?



Awesome article...

https://www.datastax.com/dev/blog/your-ideal-performance-consistency-tradeoff

Consistency Level

What is CL? Can you define it?

"At what point in time the coordinator can respond to the client?"

Tunable at runtime

- ONE (default)
- QUORUM (strict majority w.r.t RF)
- ALL

Can be applied both to read and write

```
protected static BoundStatement getWritableStatement(Session session, String cql,
                                                     boolean setAnyConsistencyLevel) {
        PreparedStatement statement = session.prepare(cql);  //Ideally prepare a statement once per session
        if(setAnyConsistencyLevel) {
              statement.setConsistencyLevel(ConsistencyLevel.QUORUM);
        }

        BoundStatement boundStatement = new BoundStatement(statement);
        return boundStatement;
}


cqlsh:meterdb> consistency;
cqlsh:meterdb> consistency all;
```

Other consistency levels are ANY, LOCAL_QUORUM, TWO, THREE, LOCAL_ONE (see documentation on the web)

What is a Quorum?

quorum = RoundDown(sum_of_replication_factors / 2) + 1

sum_of_replication_factors = datacenter1_RF + datacenter2_RF + . . . + datacentern_RF

- Using a replication factor of 3, a quorum is 2 nodes. The cluster can tolerate 1 replica down.
- Using a replication factor of 6, a quorum is 4. The cluster can tolerate 2 replicas down.
- In a two data center cluster where each data center has a replication factor of 3, a quorum is 4 nodes. The cluster can tolerate 2 replica nodes down.
- In a five data center cluster where two data centers have a replication factor of 3 and three data centers have a replication factor of 2, a quorum is 7 nodes.

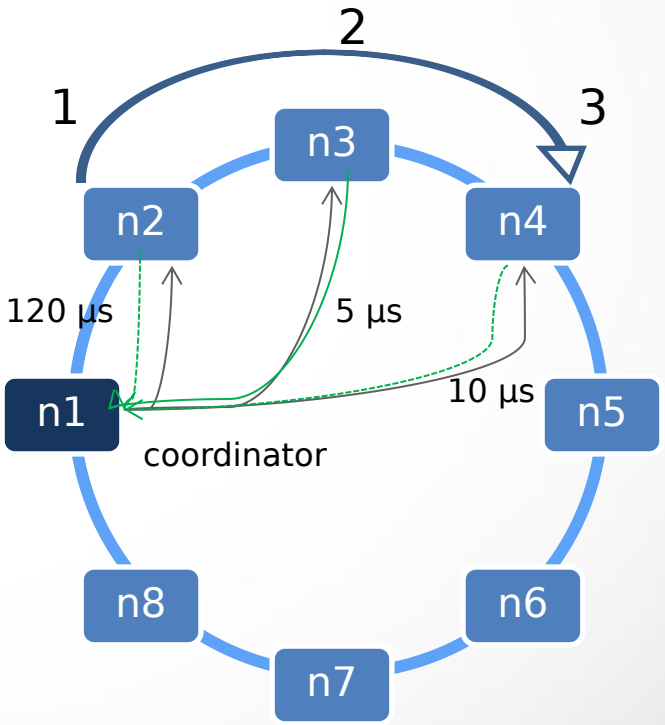
Write Consistency level

Write ONE

Send requests to **all replicas** in the cluster applicable to the PK

Wait for ONE ack before returning to client

Other acks later , asynchronously

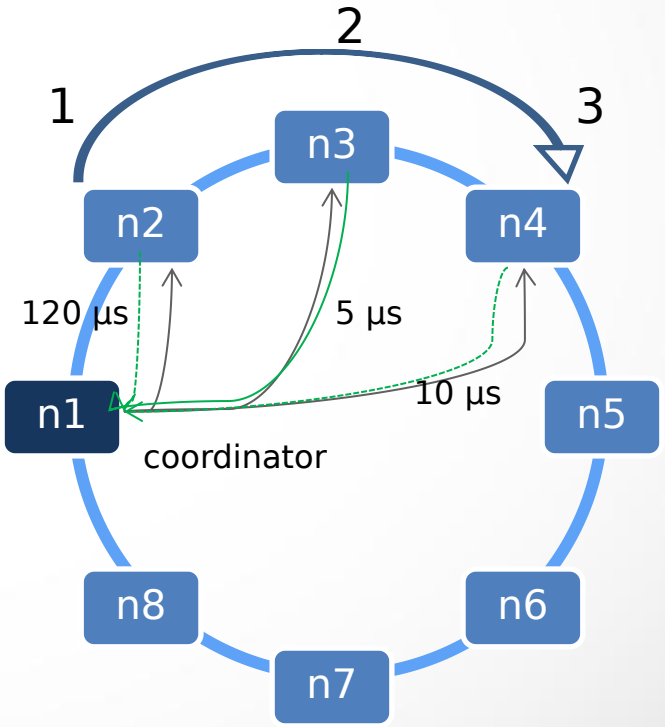


Write Consistency level

Write QUORUM

Send requests to **all replicas**

Wait for QUORUM ack before returning to client



Read Consistency level

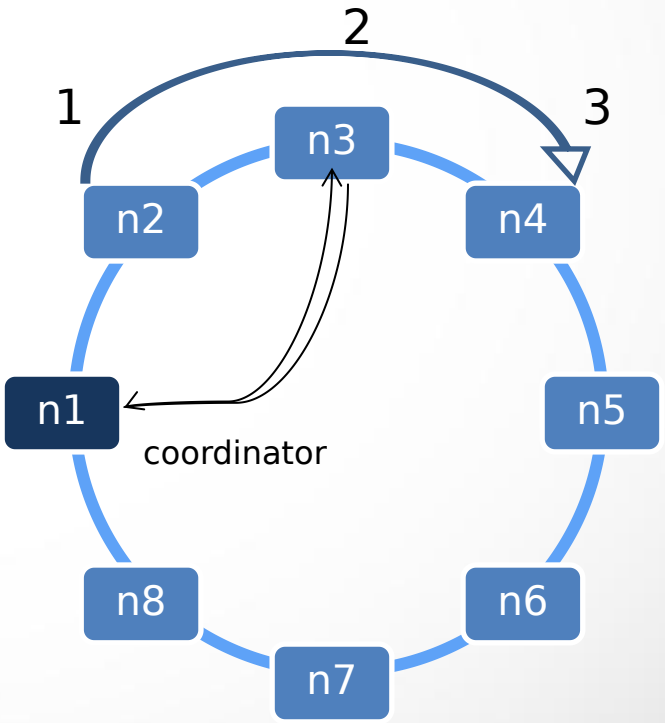
Read ONE

Read from one node among all replicas

Contact the "faster" or "preferred" node (stats)

"Preferred"??
C* pins a host for a given partition. This helps in managing cache on few nodes.

Parameter "dynamic_snitch_badness_threshold" which is a % eg 0.1 = 10% (default) means that a pinned host will be preferred over the replicas until it is 10% worse



Read Consistency level

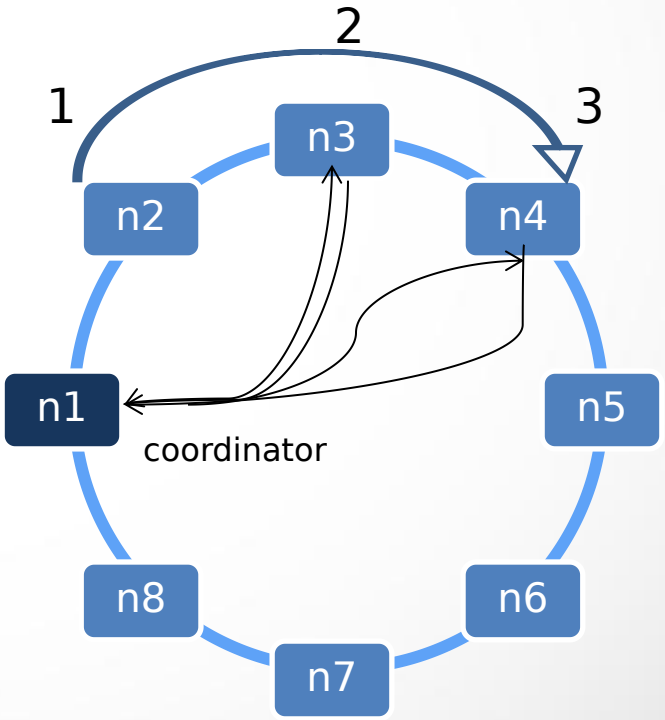
Read QUORUM

Read from one fastest/preferred node

AND **request digest** from other replicas to reach QUORUM

Return most up-to-date data to client

Read repair a bit later ...



Consistency Level - summary

ONE

Fast write, may not read latest
written value

QUORUM / LOCAL_QUORUM

Strict majority w.r.t. Replication Factor
Good balance

ALL

Not the best choice
Slow, no high availability

if(nodes_written + nodes_read) > replication factor
then you can get **immediate consistency**

Debate - Immediate consistency

- The following consistency level gives immediate consistency
  - Write CL = ALL, Read CL = ONE
  - Write CL = ONE, Read CL = ALL
  - Write CL = QUORUM, Read CL = QUORUM
- Debate which CL combination you will consider in what scenarios?
- The combinations are
  - Few write but read heavy
  - Heavy write but few reads
  - Balanced

Part 5

Write and Read path

Key components for Write

Each node implements the following key components (3 storage mechanism and one process) to handle writes

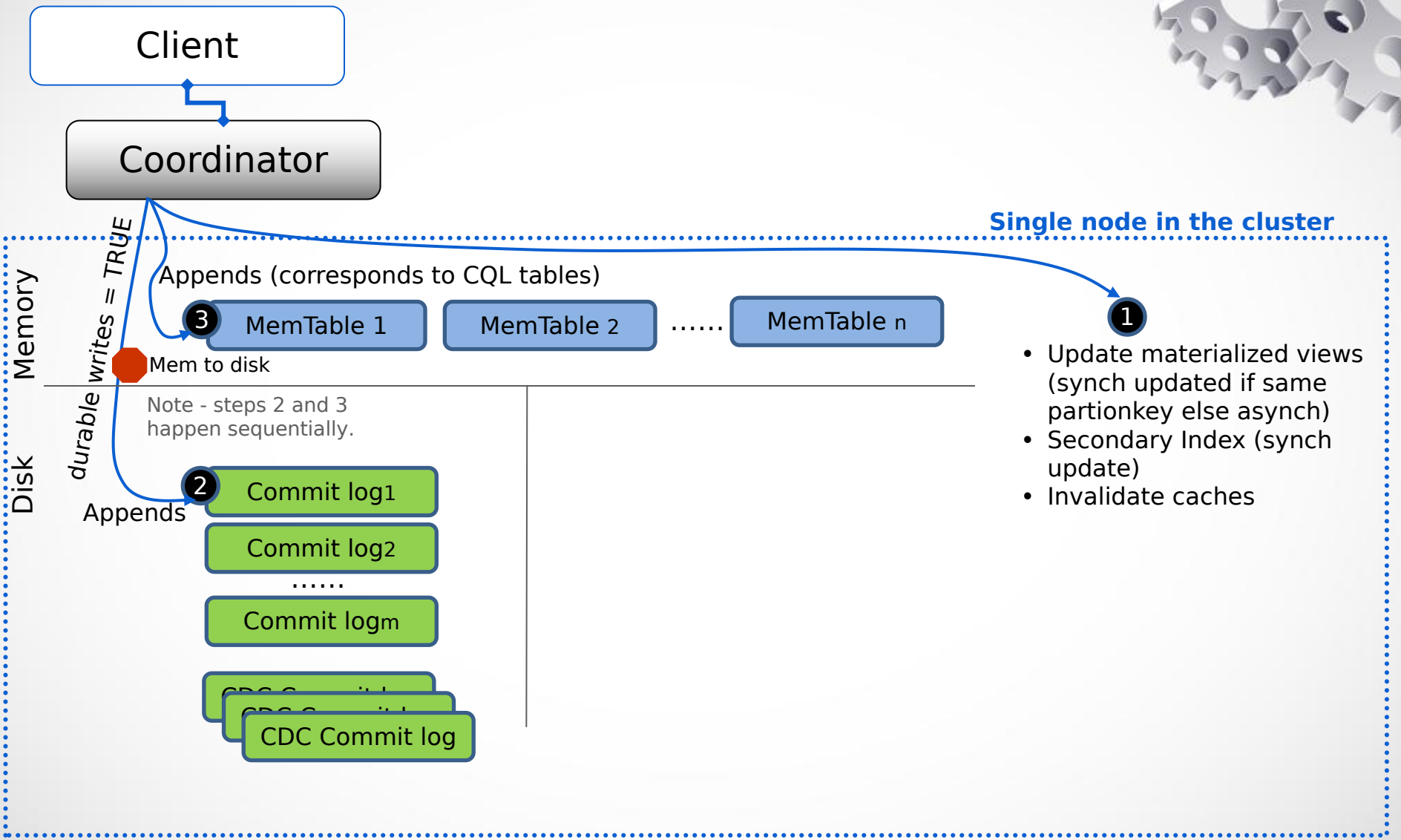
1. Memtables - the in-memory tables corresponding to the CQL tables along with the related indexes

2. CommitLog - an append-only log, replayed to restore node Memtables

3. SSTables - Memtable snapshots periodically flushed (written out) to free up heap

4. Compaction - periodic process to merge and streamline multiple generations of SSTables

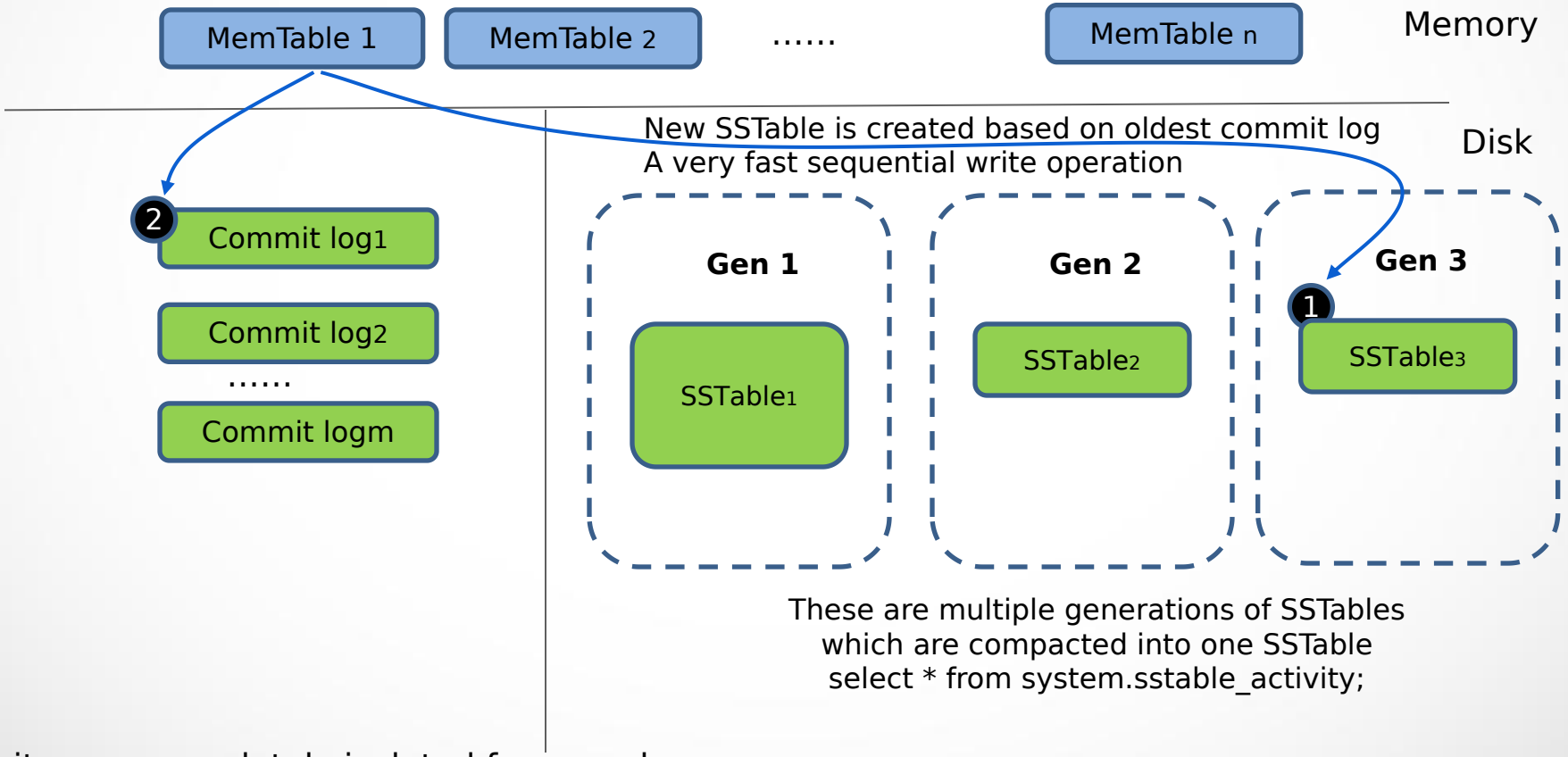
Write Path



Related configurations are commitlog_sync (batch/**periodic**), commitlog_sync_batch_window_in_ms, commitlog_synch_period_in_ms (default 10000)

Write Path

Memtable exceeds a threshold, flush to SSTable, clean log, clear memory corresponding to the memtables



Writes are completely isolated from reads

No updated columns are visible until entire row is finished (technically, entire partition)

CDC commit logs are not purged

When does the flush trigger?

1. Memtable total space in MB reached
2. OR Commit log total space in MB reached
3. OR Nodetool flush (manual)
   nodetool flush <keyspace> <table>
4. OR taking a snapshot of the node
   nodetool snapshot
5. OR node is restarting and finds data in commitlog, builds memtables and then flushes (varies depending on C* version)

The default settings about memtables in yaml are usually good and are there for a reason

Table data state

A Table data

=

It's Memtable

+

All of it's SSTables that have been flushed

One partition's data can be across multiple SST and one SST can have data of more than one partition

Memtables & SSTables

- Memtables and SSTables are maintained per cql table
  - Memtables can have in place updates if the data (for a given partition + column) already exists
  - SSTables are immutable, not written to again after the memtable is flushed.
  - A partition is typically stored across multiple SSTables
  - What is sorted? - Partitions are sorted and stored
- Size of memtables
  - Default is 25% of JVM Heap
  - Can be more in case of off heap
  - Here are the C* yaml settings (default commented)
    - # memtable_heap_space_in_mb: 2048
    - # memtable_offheap_space_in_mb: 2048
- C* pushes the data of a memtable into a queue for flushing (internally), no config required
- For each SSTable, C* creates these structures
  - Partition index - A list of partition keys and the start position of rows in the data file (on disk)
  - Partition index summary (in memory sample to speed up reads)
  - Bloom filter (optional and depends on % false positive setting)
  - There are more structures but the above are the primary components

Offheap memtables

- As of C* 2.1 a new parameter has been introduced (memtable_allocation_type) This capability is still under active development and performance over the period of time is expected to get better
- Heap buffers - the current default behavior
- Off heap buffers - moves the cell/column name and value to DirectBuffer objects. This has the lowest impact on reads — the values are still “live” Java buffers — but only reduces heap significantly when you are storing large strings or blobs
- Off heap objects - moves the entire cell off heap, leaving only the NativeCell reference containing a pointer to the native (off-heap) data. This makes it effective for small values like ints or uuids as well, at the cost of having to copy it back on-heap temporarily when reading from it (likely to become default in C* 3.0)
  - Writes are about 5% faster with offheap_objects enabled, primarily because Cassandra doesn’t need to flush as frequently. Bigger sstables means less compaction is needed.
  - Reads are more or less the same
- For more information http://www.datastax.com/dev/blog/off-heap-memtables-in-Cassandra-2-1

Commit log

- It is replayed in case a node went down and wants to come back up
- This replay creates the MemTables for that node
- Commit log comprises of pieces (files) and it can be controlled (commitlog_segment_size_in_mb)
- Total commit log is a controlled parameter as well (commitlog_total_space_in_mb), usual default is 4GB
- Commit log itself is also acrued in memory. It is then written to disk in two ways
  - Batch - if batch then all ack to requests wait until the commit log is flushed to disk
  - Periodic - the request is immediately ack but after some time the commit log is flushed. If a node were to go down in this period then data can be lost if RF=1

- CommitLogs can get fragmented as it is no longer being re-used from v2.1 because C* uses memory-mapped log write
  - http://www.datastax.com/dev/blog/updates-to-cassandras-commit-log-in-2-2

Debate - "periodic" setting gives better performance and we should use it. How to avoid the chances of data loss?

Data file name structure

- The data directories are created based on keyspaces and tables
  - <as config in C* yaml data folder>/keyspace/table-<unique table id>/<DB files>
- The actual DB files are named as below (latest release, prior releases are different). Has been shortened to avoid windows breakage
  - format-generationNum-big-component
- format - internal C* format eg 2.0 has "jb", 2.1 has "ka", 2.2 has "la"
- generationNum is a sequence and "big" is a constant
- component can be
  - CompressionInfo - compression info metadata
  - Data - PK, data size, column idx, row level tombstone info, column count, column list in sorted order by name
  - Filter - Bloom filter
  - Index - index, also include Bloom filter info, tombstone
  - Statistics - histograms for row size, gen numbers of files from where this SST was compacted
  - Summary - index summary (that is loaded in mem for read optimizations)
  - TOC - list of files
  - Digest - Text file with a digest

Change Data Capture (CDC)

- Change data capture (CDC) provides a mechanism to flag specific tables for archival as well as rejecting writes to those tables once a configurable size-on-disk for the combined flushed and unflushed CDC-log is reached
- The goals (JIRA 8844)
  - Use CQL as the primary ingestion mechanism, in order to leverage its Consistency Level semantics, and in order to treat it as the single reliable/durable SoR for the data.
  - To provide a mechanism for implementing good and reliable (deliver-at-least-once with possible mechanisms for deliver-exactly-once) continuous semi-realtime feeds of mutations going into a Cassandra cluster.
  - To eliminate the developmental and operational burden of users so that they don't have to do dual writes to other systems.
  - For users that are currently doing batch export from a Cassandra system, give them the opportunity to make that realtime with a minimum of coding.

CDC contd...

- The mechanism (JIRA 8844). We propose a durable logging mechanism that functions similar to a commitlog, with the following nuances:
  - Takes place on every node, not just the coordinator, so RF number of copies are logged.
  - Separate log per table.
  - Per-table configuration. Only tables that are specified as CDC_LOG would do any logging.
  - Per DC. We are trying to keep the complexity to a minimum to make this an easy enhancement, but most likely use cases would prefer to only implement CDC logging in one (or a subset) of the DCs that are being replicated to
  - In the critical path of ConsistencyLevel acknowledgment. Just as with the commitlog, failure to write to the CDC log should fail that node's write. If that means the requested consistency level was not met, then clients should experience UnavailableExceptions.
  - Be written in a Row-centric manner such that it is easy for consumers to reconstitute rows atomically.
  - Written in a simple format designed to be consumed directly by daemons written in non JVM languages

- Consumption (important points)
  - Cassandra would only write to the CDC log, and never delete from it
  - Cleaning up consumed logfiles would be the client daemon's responibility
  - Daemons should be able to checkpoint their work, and resume from where they left off. This means they would have to leave some file artifact in the CDC log's directory
  - More details here ... https://issues.apache.org/jira/browse/CASSANDRA-8844

- This link contains the CDC commit log reader (perhaps need to have /lib in the C* folder in path to include all C* libraries)
  - http://cassandra.apache.org/doc/latest/operating/cdc.html

Read Path

SELECT…FROM…WHERE #partition=….;

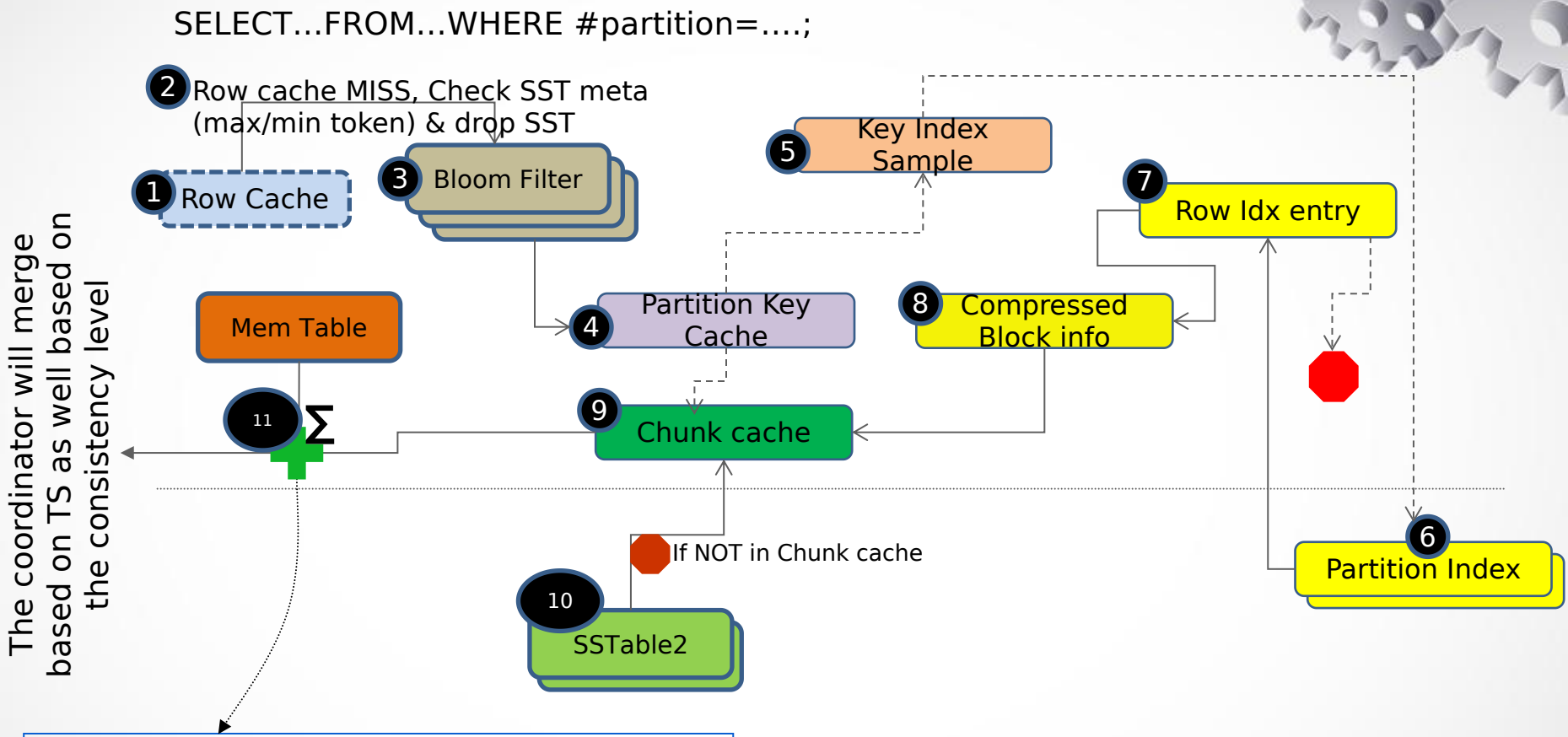
Coordinator hashes the partition key, figures out what nodes qualify, picks 1 node for data and others for digest (depending on the CL).

The components in play "per node" are -

1. Row cache
2. Prilim dropping of SST
3. Bloom filters
4. Partition key cache
5. Key index sample
6. Index
6. Row index entry (only structure on heap)
8. Compressed block info
9. Chunk cache (v3.6+ only)
10. SSTable
11. Memtable

Finally the data OR the digest is dispatched.

Read Path - Complete flow



Merge based on the most recent timestamp of the columns
1. Memtable
2. One or more than one SSTable
more on the merge process a bit later in LWW ...

Also update the row cache (if enabled) and key cache

Summary - kinds of read requests

- There are three types of read requests that a coordinator can send to a replica
  - A direct read request
  - A digest request
  - A background read repair request

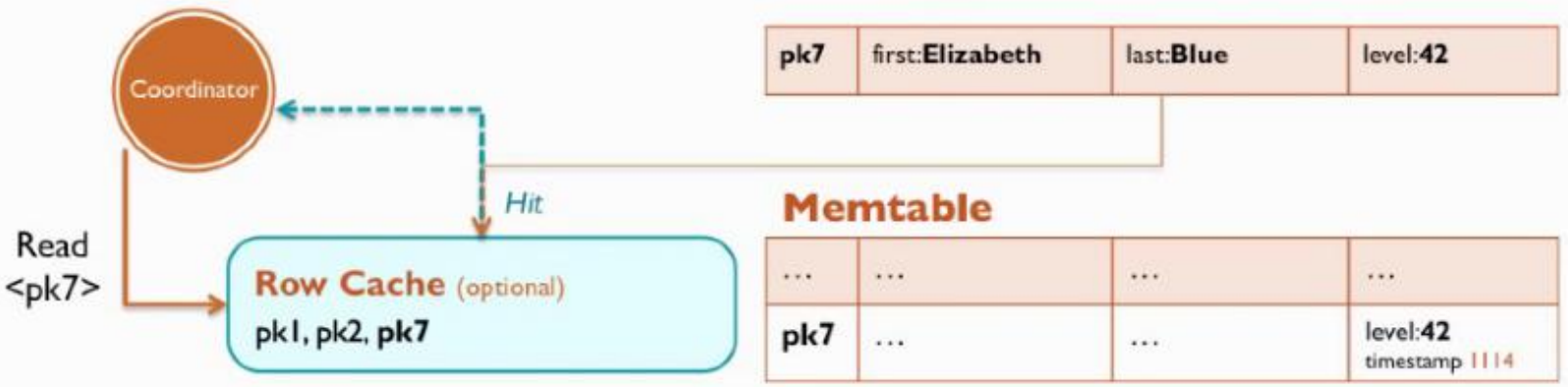
Bloom filters in action

- bloom_filter_fp_chance – table setting to control the percentage chance of false positive filter results
  - greater chance (1%) > smaller filter > higher false positives > more seeks
  - lower chance (.01%) > larger filter > lower false positives > less seeks
- Values range from 0.0 to 1.0
  - 0.0 no false positives, greatest memory use
  - 0.01 default setting (varies by compaction strategy)
  - 0.1 maximum recommended setting, diminishing returns if higher
  - 1.0 Bloom filtering disabled for this table
- Setting alterable via CQL

```
ALTER TABLE player
WITH bloom_filter_fp_chance = 0.1;
```

Setting change takes effect when the SSTables are regenerated
In development env you can use nodetool/CCM scrub BUT it is time intensive

Row caching



- Entire *merged* row for a *partition key* is saved in off-heap memory
- Caching enabled by *table property*
  - all – enable both key and row caching, for this table
  - keys_only – (default) enable key caching only, for this table
  - rows_only – enable row caching only, for this table
  - none – disable caching this table

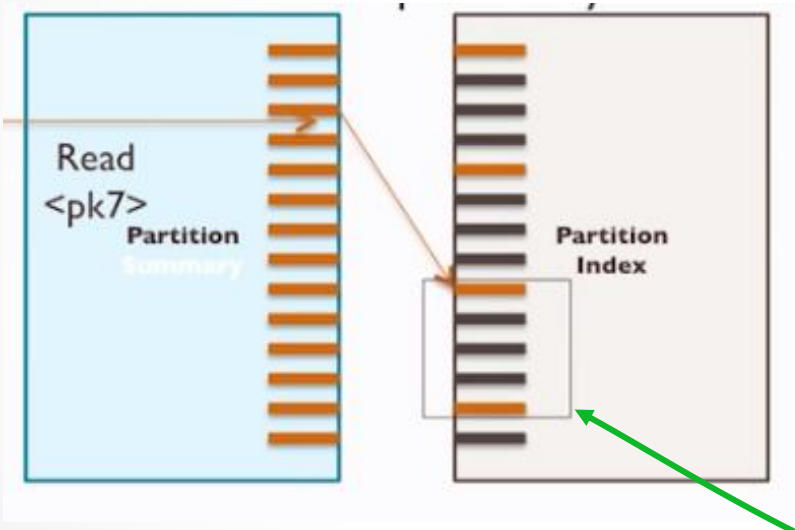
```
CREATE TABLE player (
  first text PRIMARY KEY,
  last text,
  level text
)
WITH caching = 'rows_only';
```

```
ALTER TABLE player
WITH caching = 'rows_only';
```

Caches are also written on disk so that it comes alive after a restart
Global settings are also possible at the c* yaml file

Key caching

- Stored per SSTable even if it is a common store for all SSTables
- Stores only the key
- Reduces the seek to just one read per replica (does not have to look into the disk version of the index which is the full index)
- This also periodically get saved on disk as well



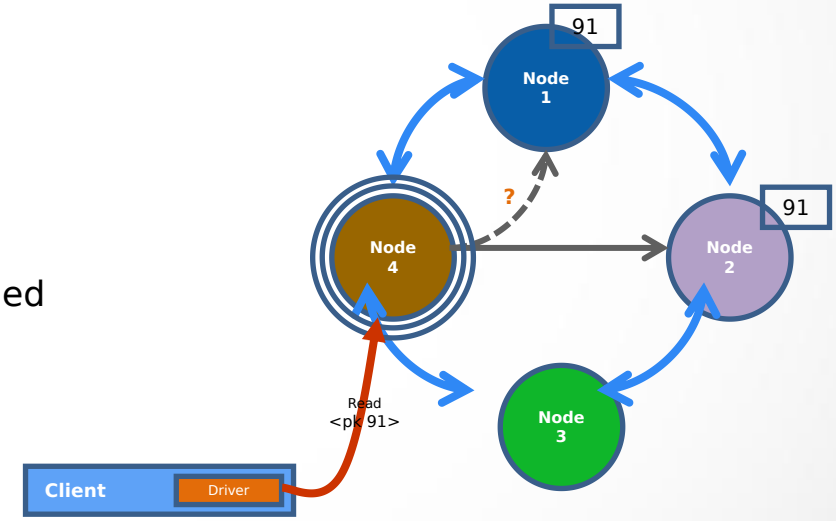

```
CREATE TABLE player (
   first text PRIMARY KEY,
   last text,
   level text
)
WITH min_index_interval = 256
AND max_index_interval = 2048;
```

- Changing index_interval increases/decreases the gap using alter table
- Increasing the gap means more disk hits to get the partition key index
- Reducing the gap means more memory but get to partition key without looking into disk based partition index

Eager retry

- If a node is slow in responding to a request, the coordinator forwards it to another holding a replica of the requested partition
- Valid if RF>1
- C* 2.0+ feature
- You can programatically control this as well in the later drivers

```
Cluster cluster = Cluster.builder()
    .addContactPoint("127.0.0.1")
    .withSpeculativeExecutionPolicy(
        //OR PercentileSpeculativeExecutionPolicy
        new ConstantSpeculativeExecutionPolicy(
            500, // delay before a new execution is launched
            2    // maximum number of executions
        ))
    .build();
```



ALTER TABLE users WITH speculative_retry = '10ms';
Or,
ALTER TABLE users WITH speculative_retry = '99percentile';

Note - retry policy will not be executed when using a local consistency level but with no local nodes available

Last Write Win (LWW)

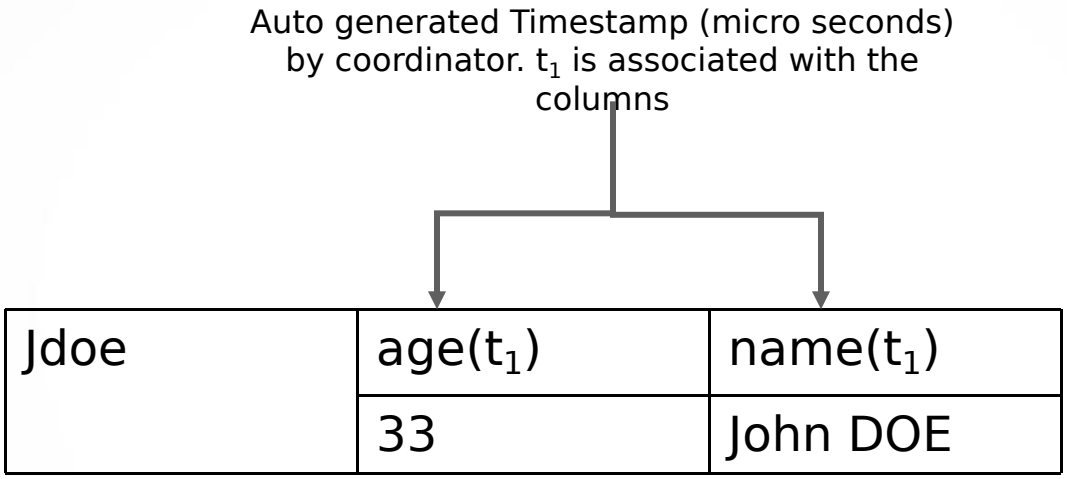
```
INSERT INTO users(login,name,age) VALUES ('jdoe', 'John DOE', 33);
```

#Partition

Jdoe	age	name
	33	John DOE

Last Write Win (LWW)

```
INSERT INTO users(login,name,age) VALUES ('jdoe', 'John DOE', 33);
```



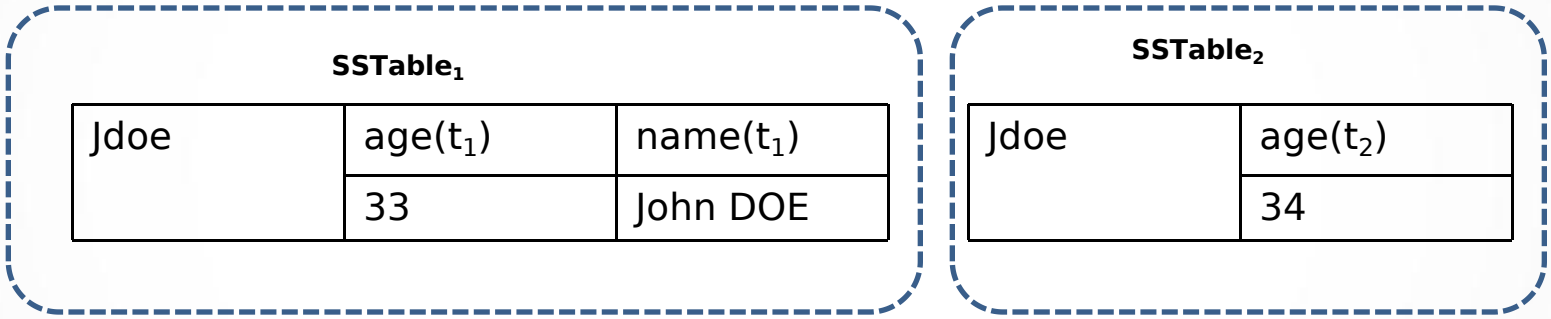
Q: How do I know what was the timestamp associated with the column?

A: select writetime(age) from users where user_id = 'Jdoe';

Last Write Win (LWW)

```
UPDATE users SET age = 34 WHERE login = 'jdoe';
```

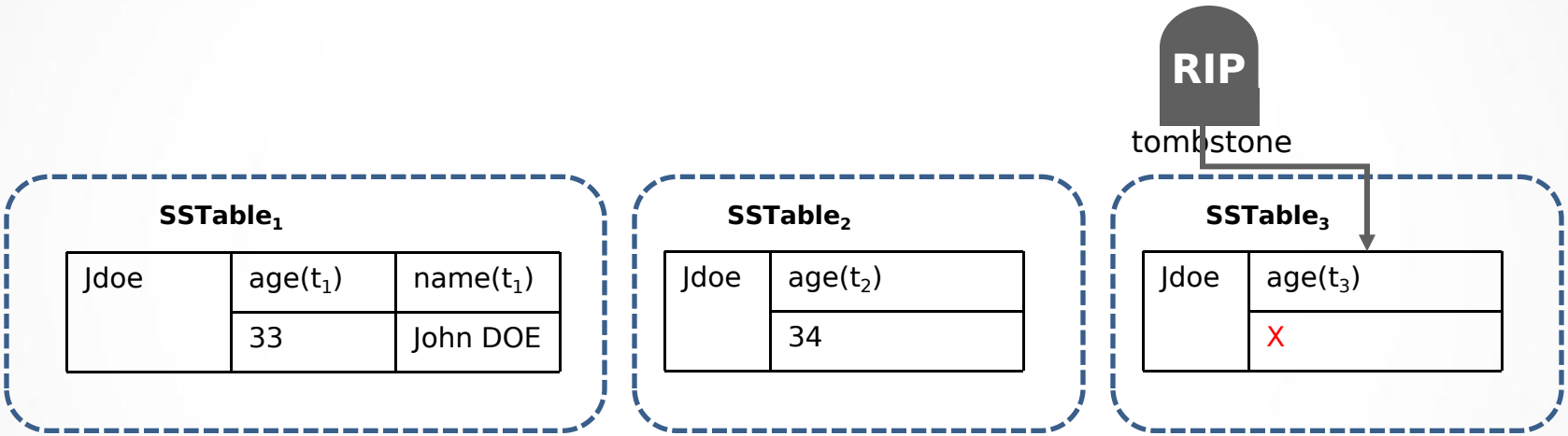
Assume a flush occurs



Remember that SSTables are immutable, once written it cannot be updated.
Creates a new SSTable

Last Write Win (LWW)

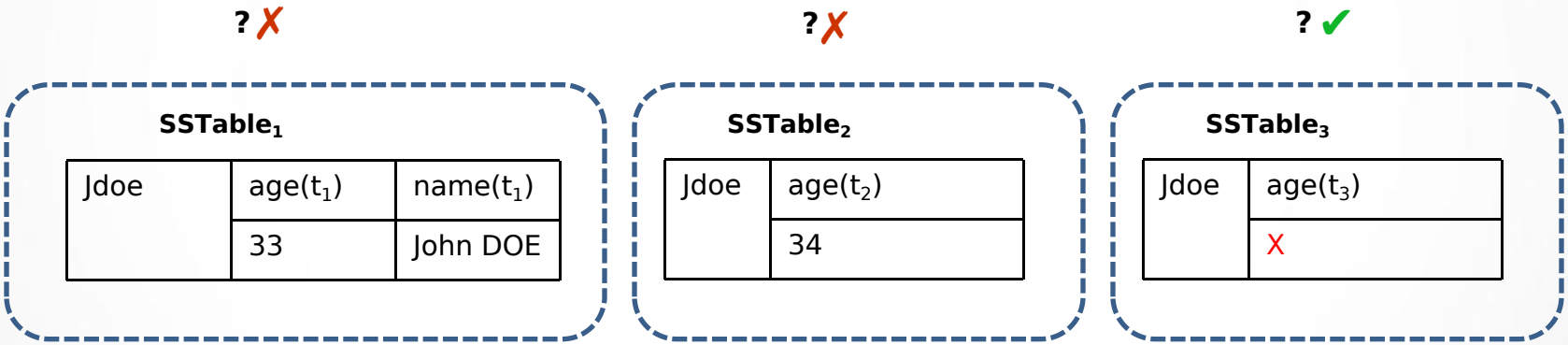
```
DELETE age from users WHERE login = 'jdoe';
```



Last Write Win (LWW)

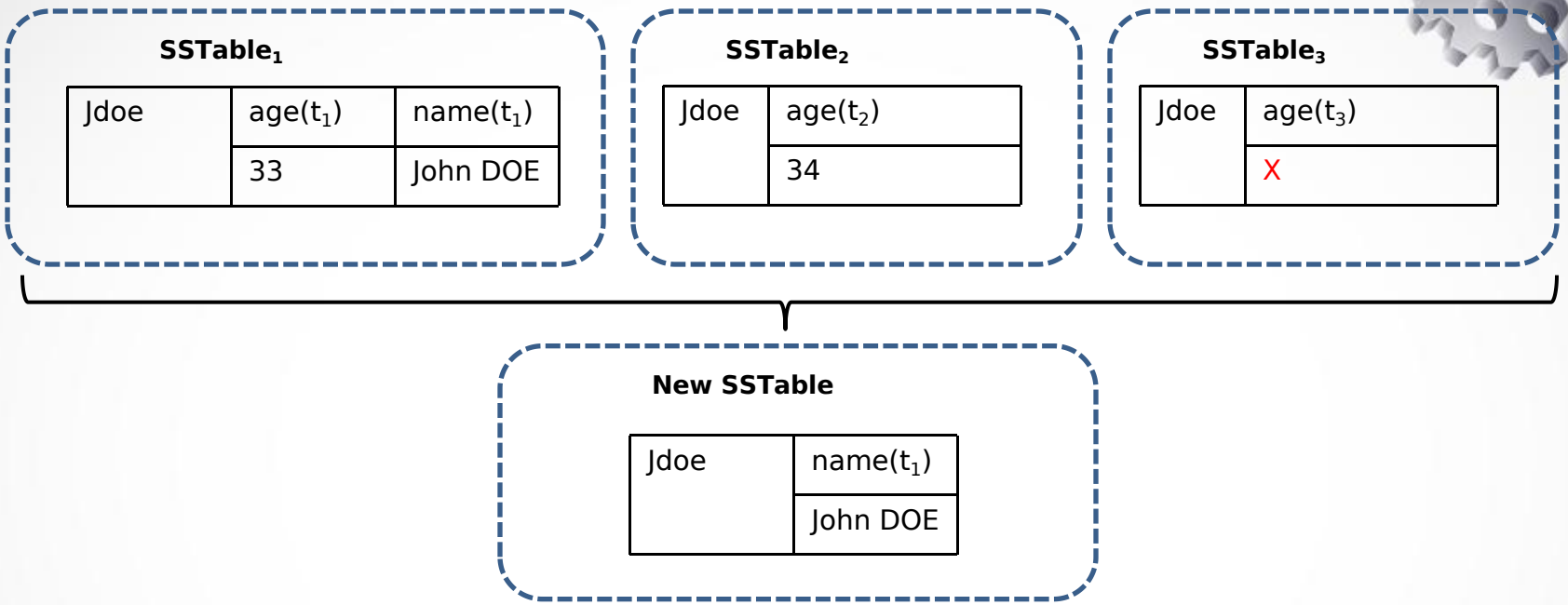
```
SELECT name, age from users WHERE login = 'jdoe';
```

Where to read from? How to construct the response?



Debate: Writes in C* are idempotent. Why?

Compaction



- Related SSTables are merged
- Most recent version of each column is compiled to one partition in one new SSTable
- Columns marked for eviction/deletion & tombstones older than gc_grace_seconds are removed
- Old generation SSTables are deleted
- A new generation SSTable is created (hence the generation number keep increasing over time)

Disk space freed = sizeof(SST1 + SST2 + .. + SSTn) - sizeof(new compact SST)
Commit logs (containing segments) are versioned and reused as well
Newly freed space becomes available for reuse

"Coming back to life"

- Let's assume multiple replicas exist for a given row/column
- One node was NOT able to recored a tombstone because it was down
- If this node remains down past the gc_grace_seconds without a repair then it will contain the old record
- Other nodes meanwhile have evicted the tombstone column (compaction OR nodetool repair will do the eviction once the tombstone > gc_grace_seconds)

- When the downed node does come up it will bring the old column back to life (will replicate) on the other nodes!

- To ensure that deleted data never resurfaces, make sure to run repair at least once every gc_grace_seconds, and never let a node stay down longer than this time period

Compaction

- Three strategies
  - Size tiered (default)
  - Leveled tiered (needs about 50% more IO than size tiered but the number of SSTables visited for data will be less)
  - Date tiered (deprecated as of 3.8) instead use "Time window" compaction
- Choice depends on the disk
  - Mechanical disk = Size tiered
  - SSD = Leveled tiered
- Choice depends on use case too
  - Size - It is best to use this strategy when you have insert-heavy, read-light workloads
  - Level - It is best suited for read-heavy workloads that have frequent updates to existing rows
  - Date - for timeseries data along with TTL
- As per Datastax documentation
  - Cassandra 2.1 improves read performance after compaction by performing an incremental replacement of compacted SSTables.
  - Instead of waiting for the entire compaction to finish and then throwing away the old SSTable (and cache), Cassandra can read data directly from the new SSTable even before it finishes writing.
- Can be initiated by nodetool compact BUT this is an antipattern because it is also called as a "Major compaction" and clumps all related SST into a single SST which can become massive!
- Revisiting disk space calculation (http://docs.datastax.com/en//cassandra/2.0/cassandra/architecture/architecturePlanningDiskCapacity_t.html)