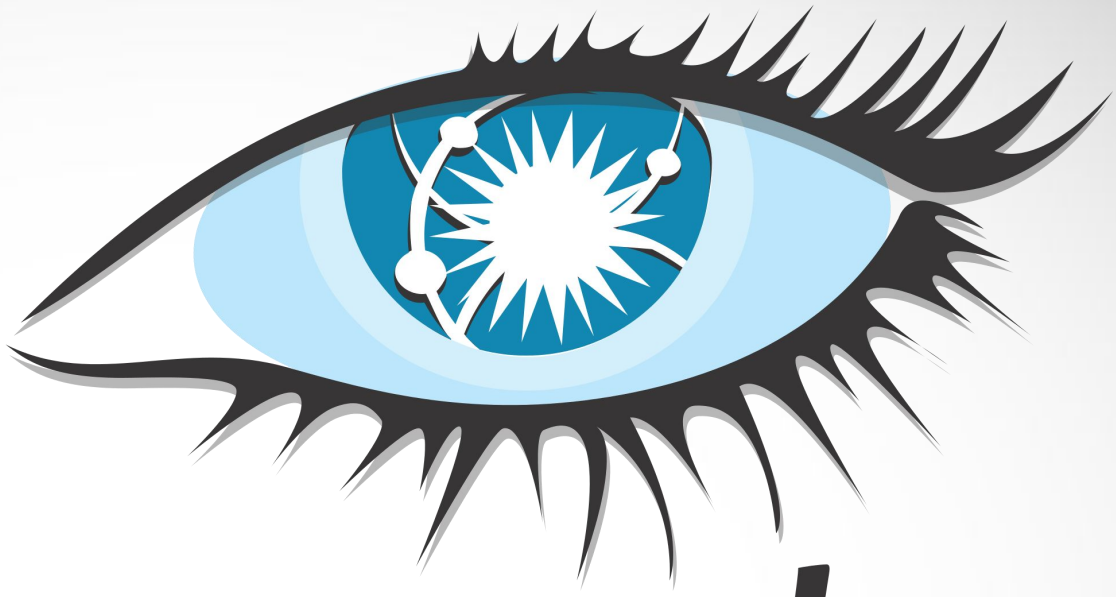


cassandra

By
Nirmallya Mukherjee

Certified Cassandra Developer
Certified Cassandra Administrator

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

Token generation

- To setup C* in a non vnode mode use the below command to generate the tokens
  - python -c 'print [str(((2**64 / number_of_tokens) * i) - 2**63) for i in range(number_of_tokens)]'
  - http://docs.datastax.com/en/cassandra/2.0/cassandra/configuration/configGenTokens_c.html
- For integrated hadoop it is recommended to go for nodes and not vnodes
  - http://docs.datastax.com/en/datastax_enterprise/4.8/datastax_enterprise/config/configVnodes.html

Node tool

- Very important tool to manage C* in production for day to day admin
- Nodetool has over 40 commands
- Can be found in the $CASSANDRA_HOME/bin folder

- Try a few commands
  - next slide...

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

...

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.peers
  - system.schema_columnfamilies
  - system.schema_columns
  - system.local (info about itself)
  - system.<many others>
- Local contains information about nodes (superset of gossip)

Part 4

Concepts II

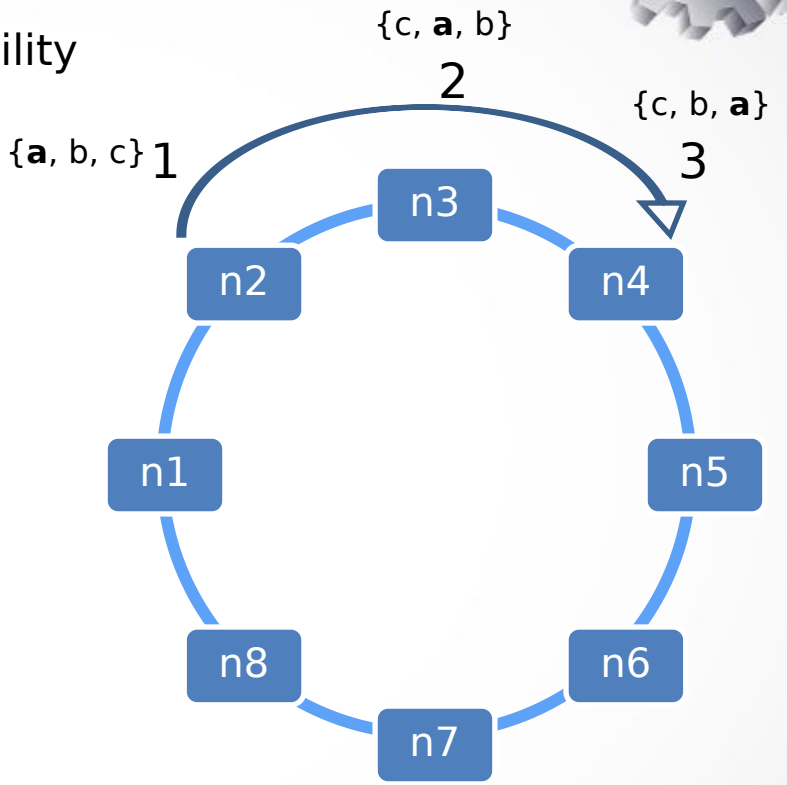
Failure/Partition Tolerance By Replication

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(.., .., ..);
```

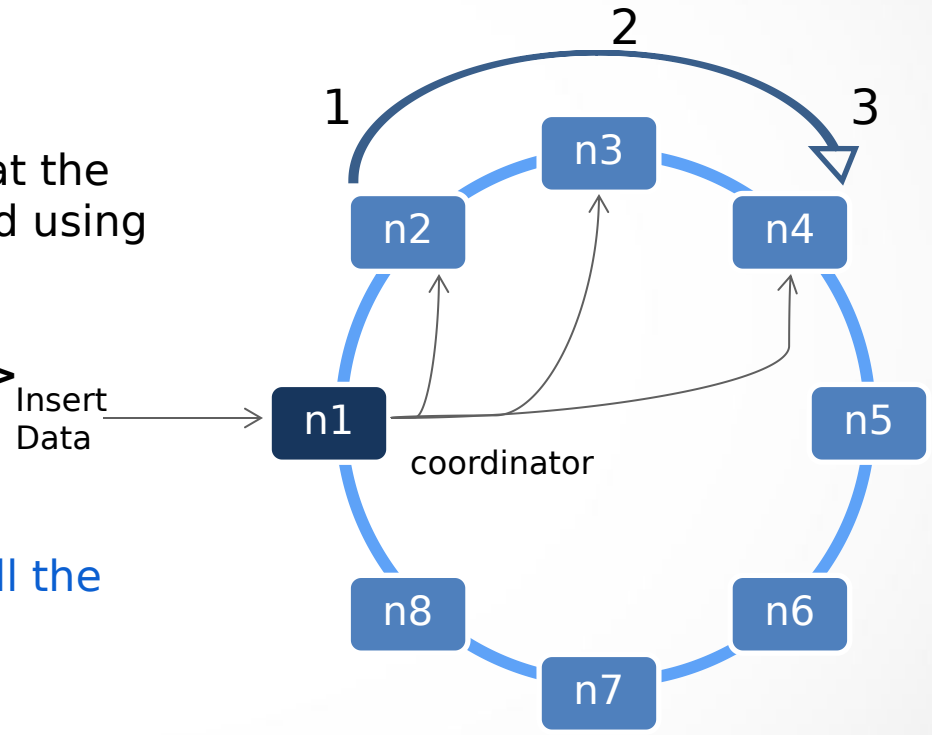
Coordinator Node - Single DC replication

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 ...

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.

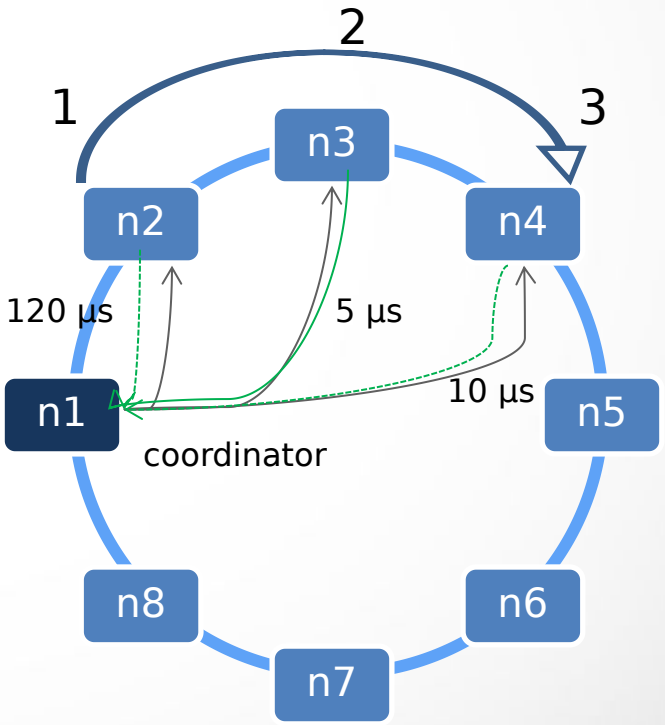
Write Consistency level

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

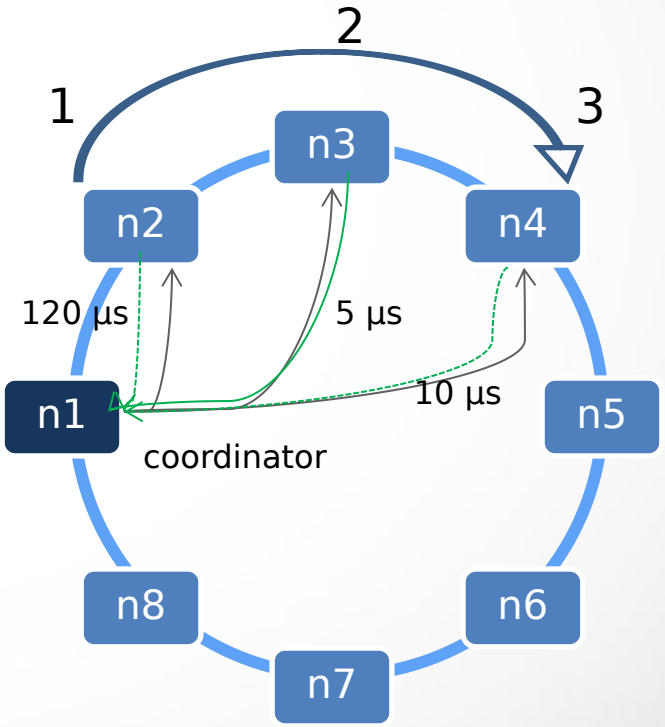


Write Consistency level

Write QUORUM

Send requests to **all replicas**

Wait for QUORUM ack before returning to client



Read Consistency level

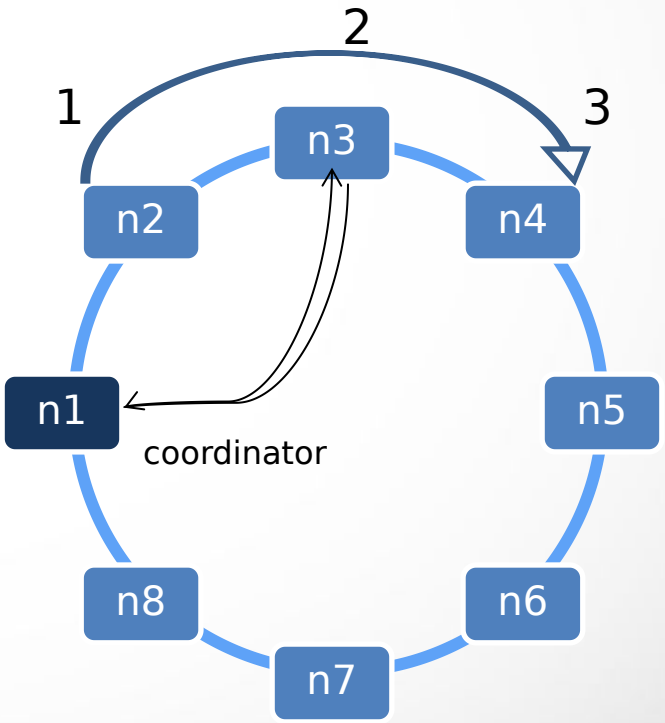
Read ONE

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

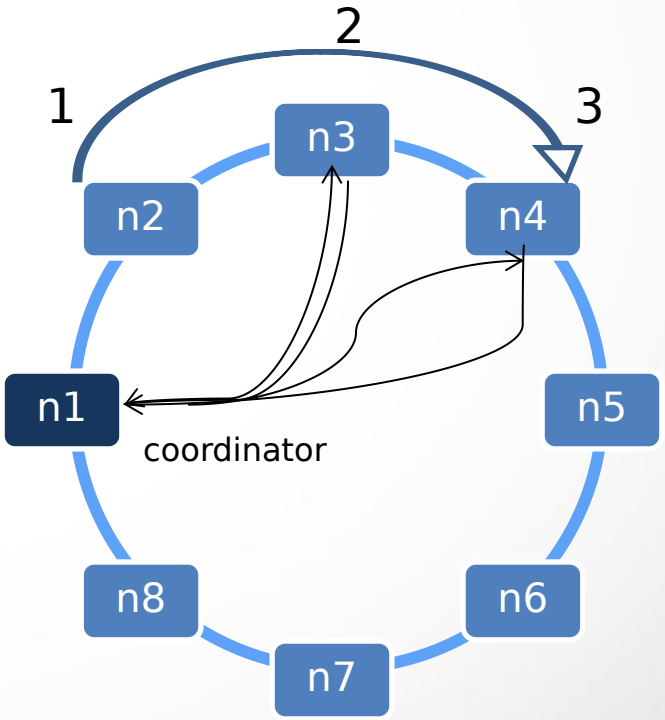
Read QUORUM

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

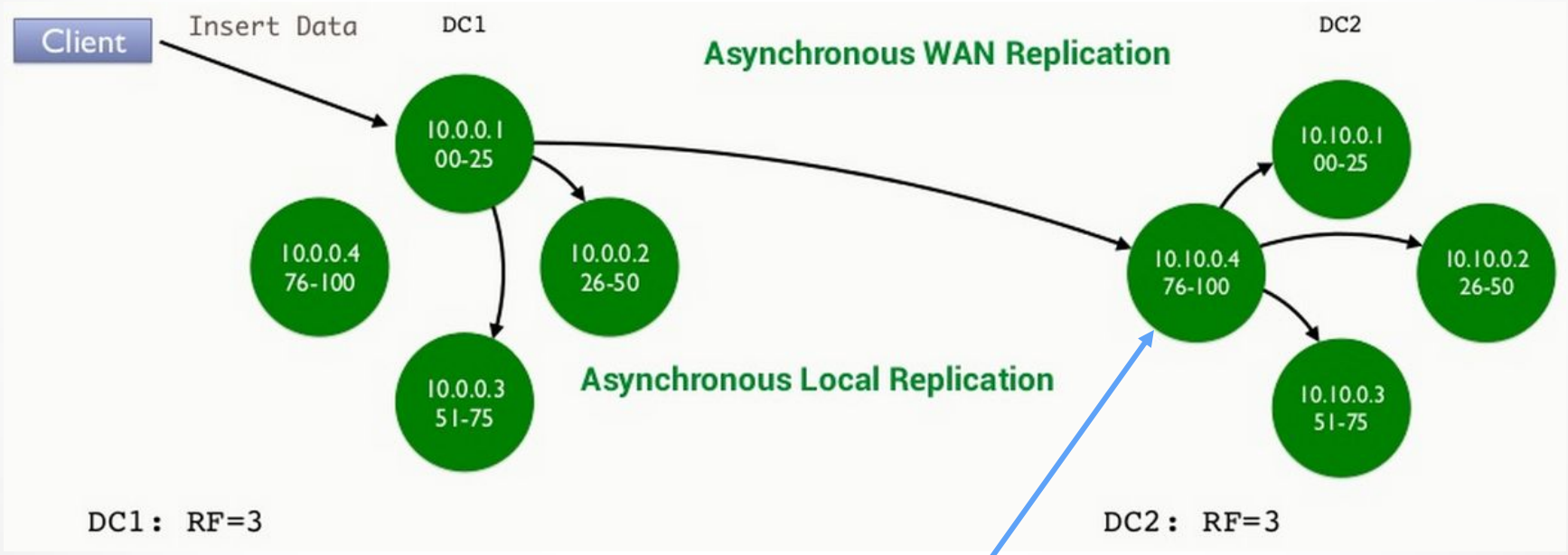
if(nodes_written + nodes_read) > replication factor
then you can get **immediate consistency**

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

Coordinator Node - Each quorum

In case of Each_Quorum C* needs to get a quorum ok from all DCs, it does so by selecting one node as a remote coordinator (Each Quorum is applicable to writes only).



Remote Coordinator is also chosen for cross DC writes

Ensuring consistency!

- C* allows for tunable consistency and hence we have eventual consistency
- For whatever reason there can be inconsistency of data among nodes
- Inconsistencies are fixed using
  - Hinted handoff
  - Read repair - update data partitions during reads
    - Blocking
    - Non-blocking
  - Nodetool repair - regular maintenance (aka Anti-entropy ops)

Hinted Handoff

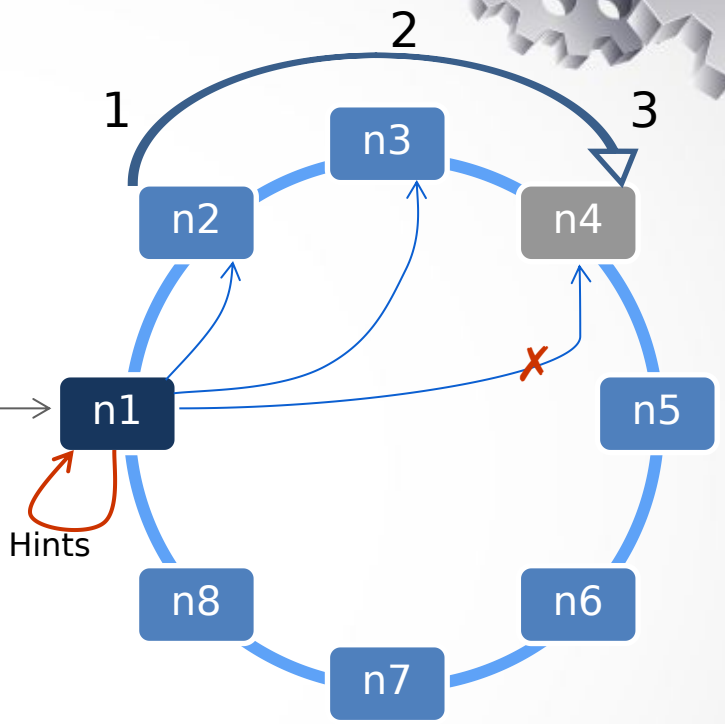
A write request is received by the coordinator

Coordinator finds a node is down/offline ahead of time OR does not ack a write mutation

Write Consistency level MUST be honored and a guarantee that any read consistency level can read the record, else no hints

It stores "Hints" on behalf of the offline node

Read repair / nodetool repair will fix inconsistencies.



Hinted Handoff

When the offline node comes up, the coordinator forwards the stored hints

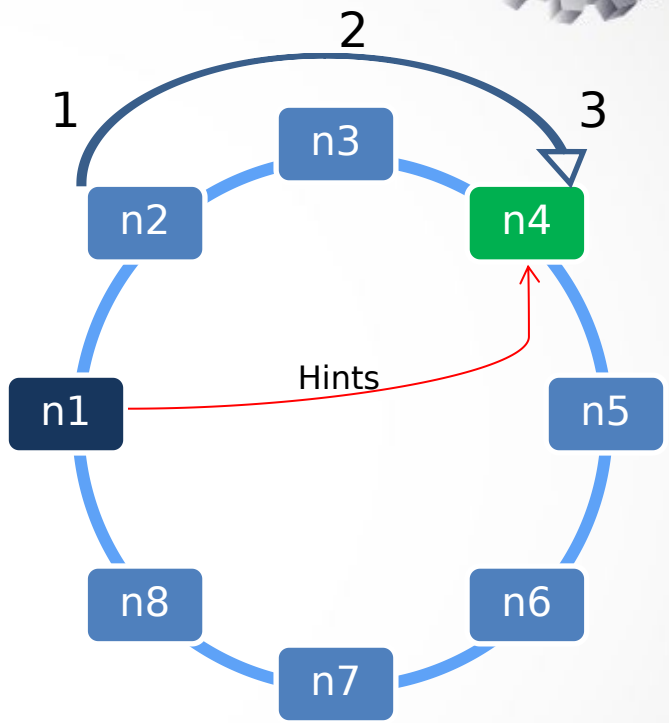
The node synchs up the state with the hints

The coordinator cannot perpetually hold the hints

Configure an appropriate duration for any node to hold hints - default 3 hrs (max_hint_window_in_ms)

If a node is down for longer than max_hint_window_in_ms, the coordinator node discards the stored hints

Can be enabled per DC or globally
hinted_handoff_enabled: true/false &
hinted_handoff_disabled_datacenters with list of DCs



A hinted handoff is taken in hints specific logs (max_hints_file_size_in_mb, default 128MB) on disk bypassing the regular write path. It is taken only when the write consistency level can be satisfied and a guarantee that any read consistency level can read the record. Earlier then v3.0, there was a system table called "hints" which stored the hints.

Hinted Handoff

- "Any" is a special consistency level where even if all nodes are down the write will succeed using a hinted handoff, use it very cautiously

- If too many nodes reach timeout at once, there can be substantial memory pressure on coordinator node because it tracks how many hints are currently writing
  - If the number of hints being written gets too high, coordinator will temporarily refuse writes
  - UnavailableException/withOverloadedException is thrown in this case

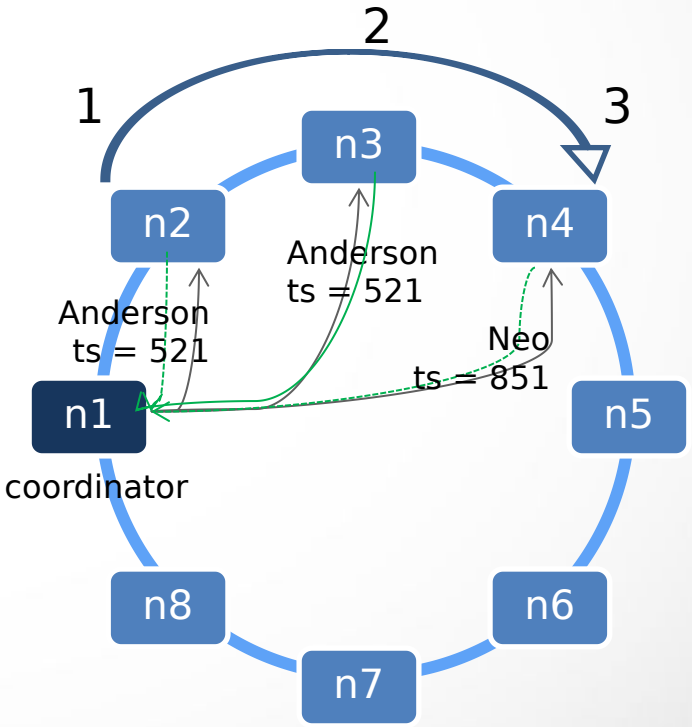
What is a read repair?

- During reads the full data is requested from ONE replica by the coordinator
- Others send a digest which is a hash of the data to the coordinator
- Coordinator checks if the data and the digests match
- If they do not then the coordinator asks the other nodes to send the data
- The coordinator then does the conflict resolution to figure who has the latest (in this case n4)
- Then the stale nodes (n2, n3) are updated automatically, blocking the read

- Assume consistency level = 1
- So who to compare with? Get data from all on a configurable % of reads and fix the data (background)

- dclocal_read_repair_chance is the configuration (defaults to 10% or 0.1) = a 10% chance that a read repair will happen when a read comes in for "low CL" (if CL=1 then who to compare the data with? Coordinator will issue a read from ALL the replicas and then compare - all in asynch)
- This is defined per table or column family

Repair with nodetool, a bit later ...



Read repair - a bit more!

- There are two scenarios of read repairs (data repair)
  - Blocking (CL>1)
  - Background (CL=1)

  You can see this in ./nodetool netstats
- Blocking
  - In case of a high consistency level where a node finds data from > 1 nodes
  - If any data is stale then it blocks read, fixes the data on the nodes as selected by the CL and then returns the result
- Background
  - This is the regular scenario where the node fixes other node and the op is governed by read_repair_chance parameter and at CL=1
  - Data is read and fixed for all replicas

- Anti entropy repair aka "manual repair" is invoked by nodetool repair (a bit later...)

Part 5

Write and Read path

Storage

- Writes are harder than reads to scale
- Spinning disks aren't good with random I/O
- Goal: minimize random I/O
  - Log Structured Merged Tree (LSM)
  - Push changes from a memory-based component to one or more disk components

Some more on LSM

- An LSM-tree is composed of two or more tree-like component data structures
- Smaller component is C0 sits in memory - aka Memtables
- Larger component C1 sits in the disk
  - aka SSTables (Sorted String Table)
  - Immutable component
- Writes are inserted in C0 and logs
- In time flush C0 to C1
- C1 is optimized for sequential access

Key components for Write

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

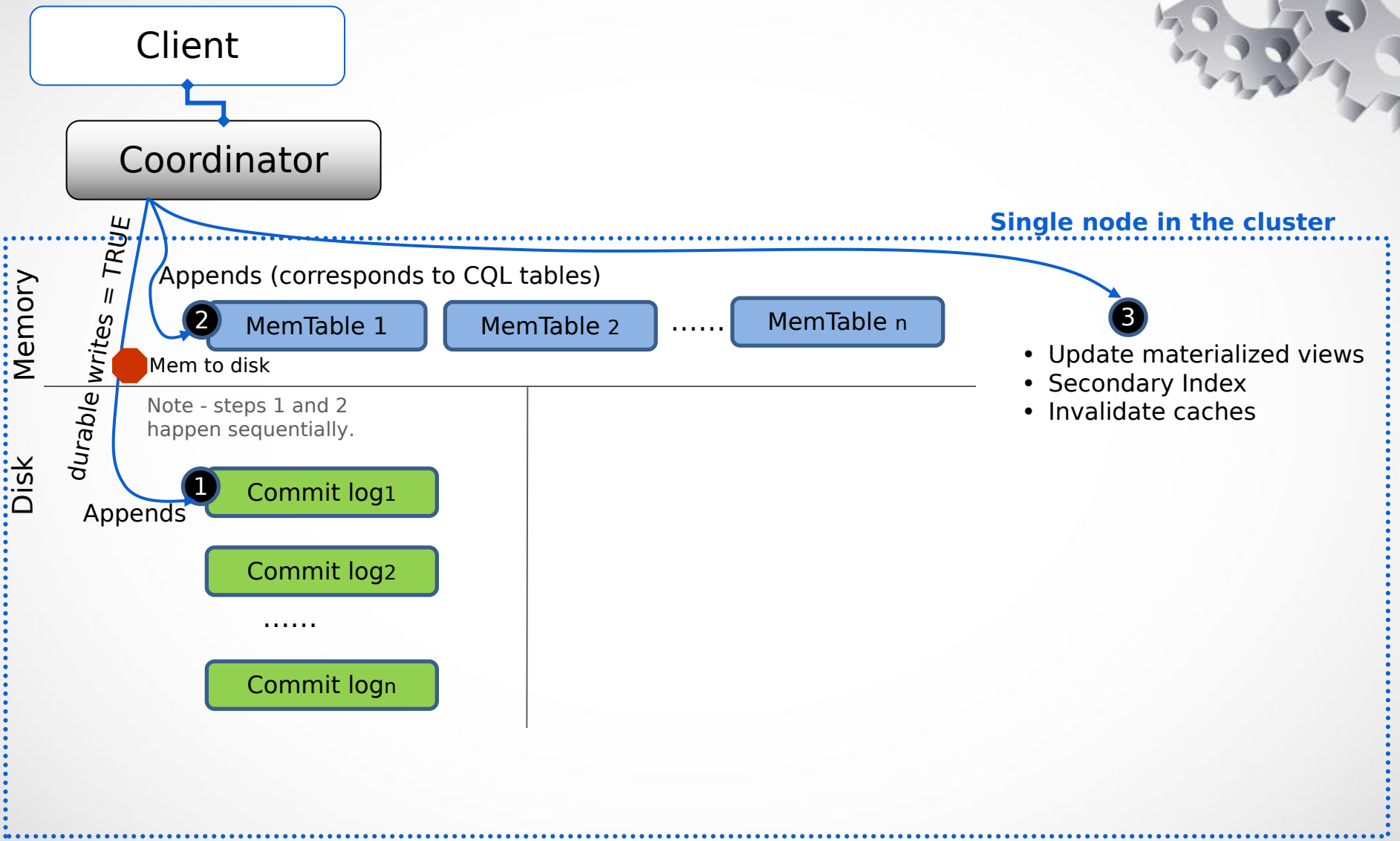
1. Memtables - the in-memory tables corresponding to the CQL tables along with the related indexes

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

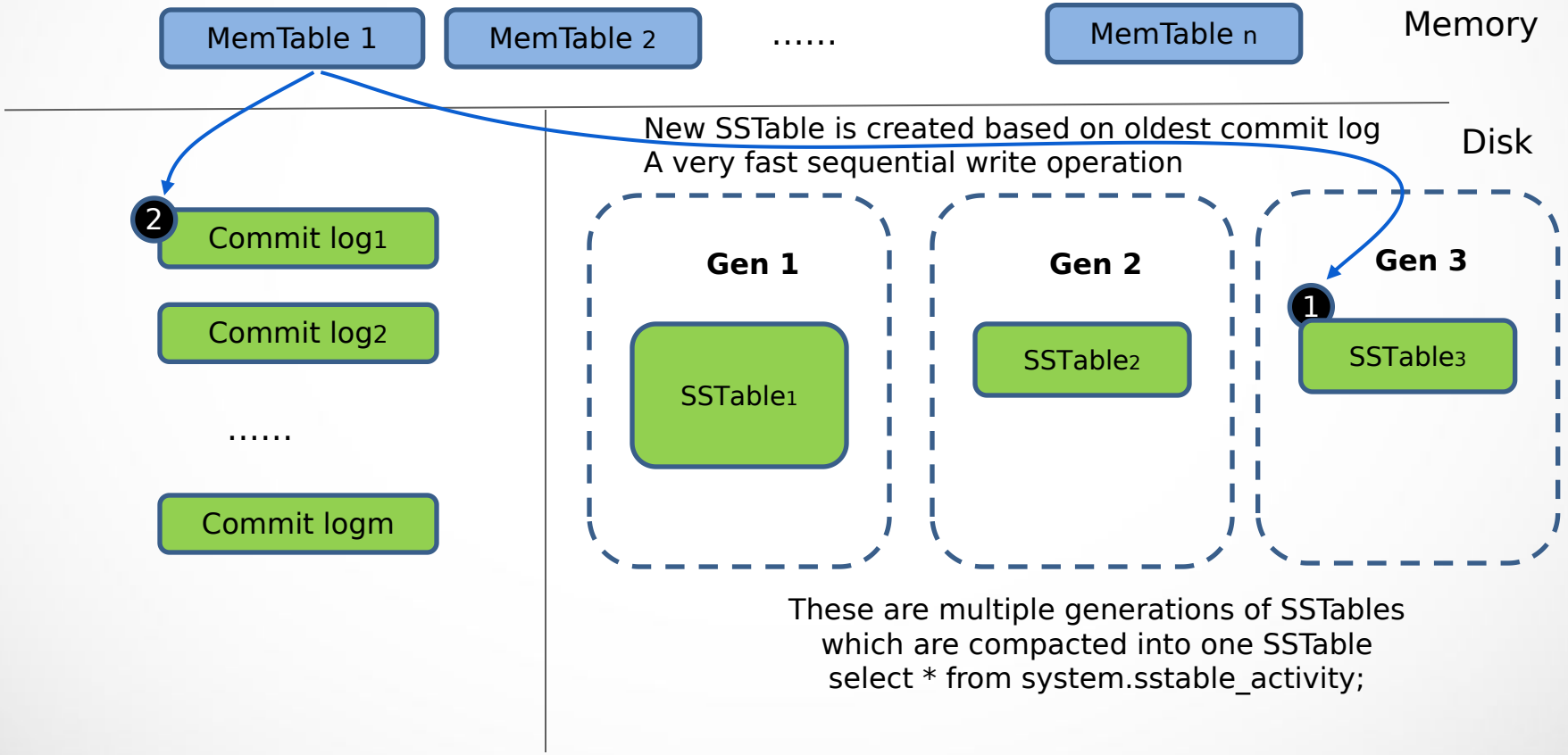
Write Path



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 heap corresponding to the memtables



Writes are completely isolated from reads

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

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 mutiple 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

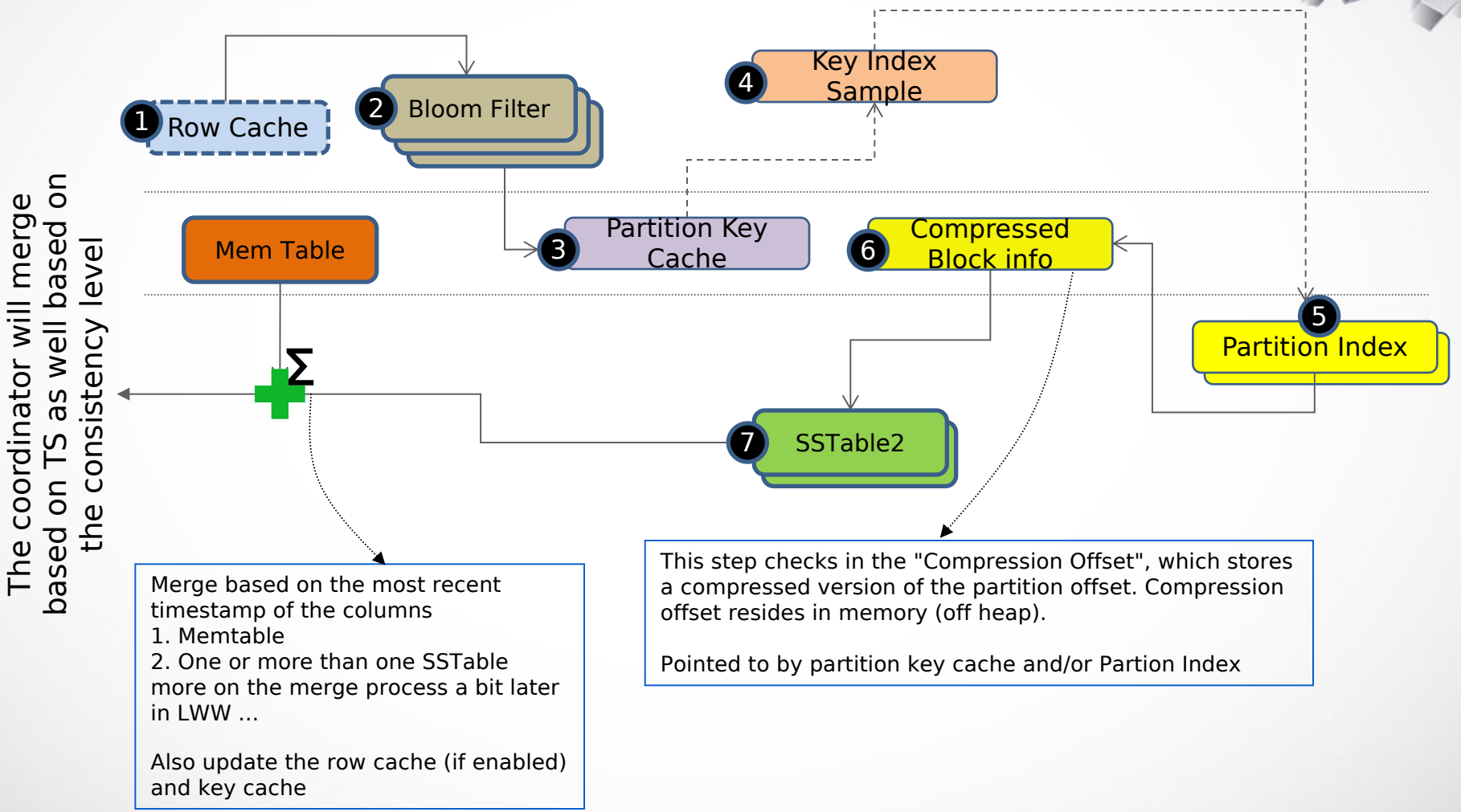
Debate - "periodic" setting gives better performance and we should use it. How to avoid the chances of data loss?

Data file name structure

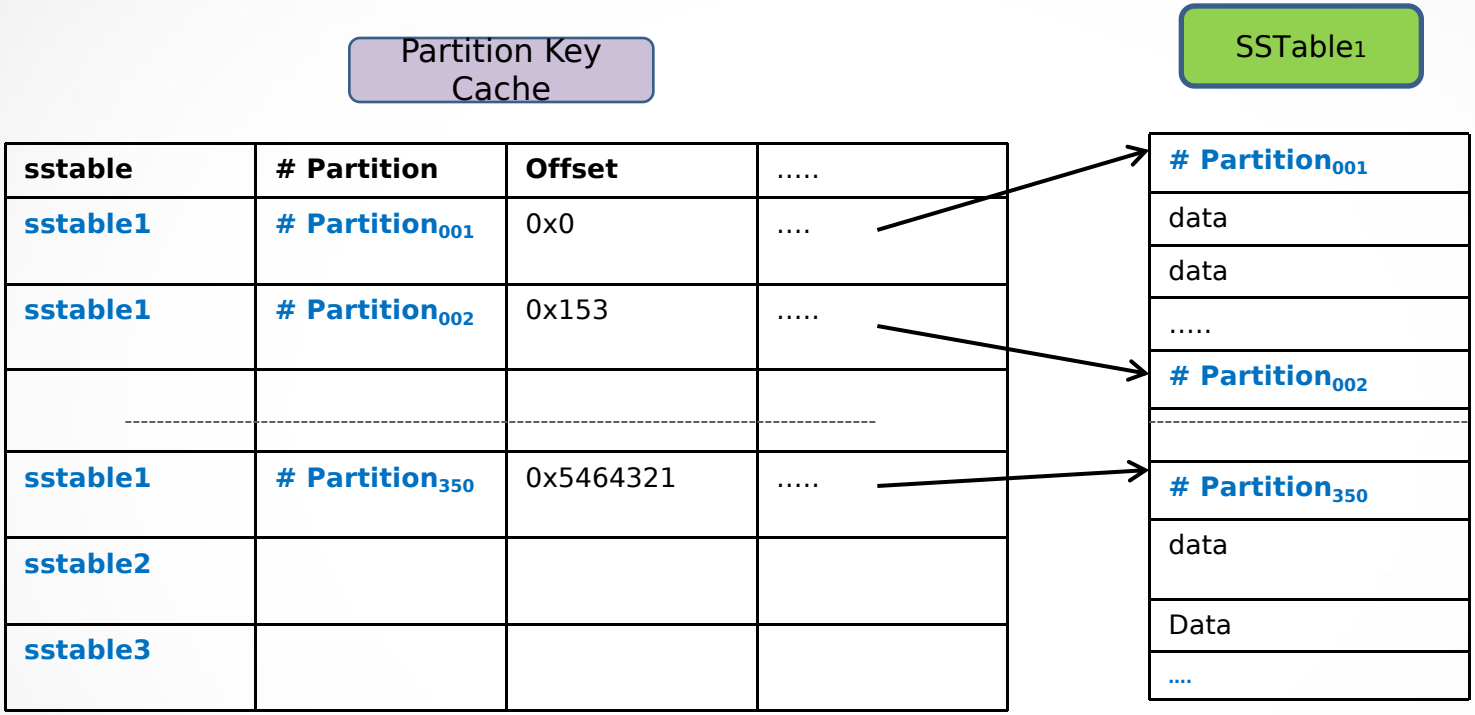
- The data directories are created based on keyspaces and tables
  - <as config in C* yaml data folder>/keyspace/table-<unique table id>/<DB files>
- The actual DB files are named as below (latest release, prior releases are different). Has been shortened to avoid windows breakage
  - format-generationNum-big-component
- 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

Read Path - complete flow

SELECT…FROM…WHERE #partition=….;

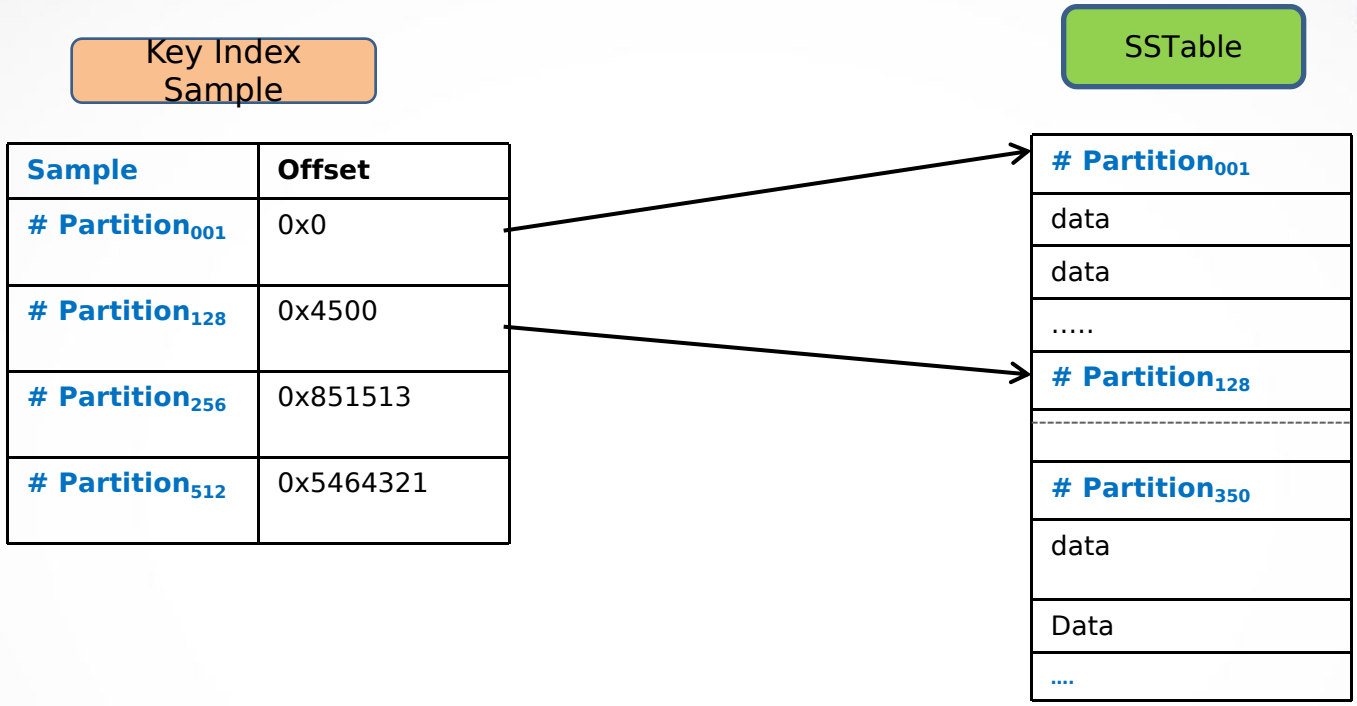


Partition Key Cache



Partition key cache stores the offset position of the partition keys that have been read recently.

Key Index Sample



Think of the offset as an absolute disk address that fseek in C can use

Ratio of the key index sample is 1 per 128 keys

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

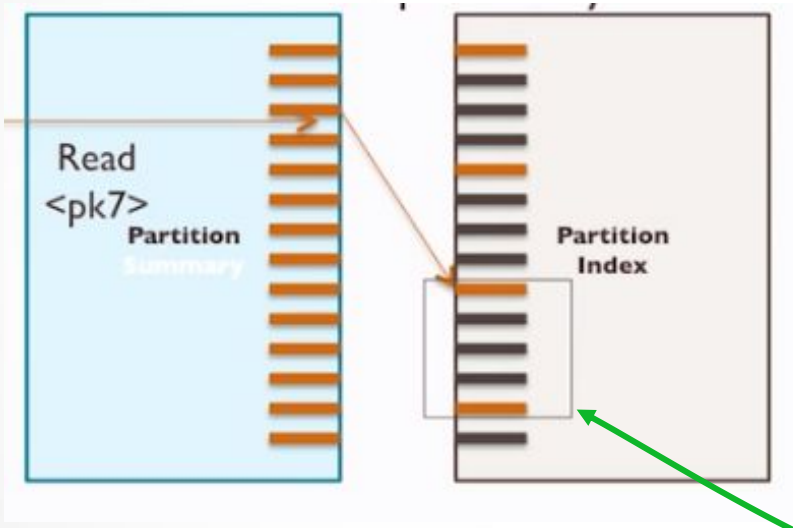
```
CREATE TABLE player (
  first text PRIMARY KEY,
  last text,
  level text
)
WITH caching = 'rows_only';
```

```
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

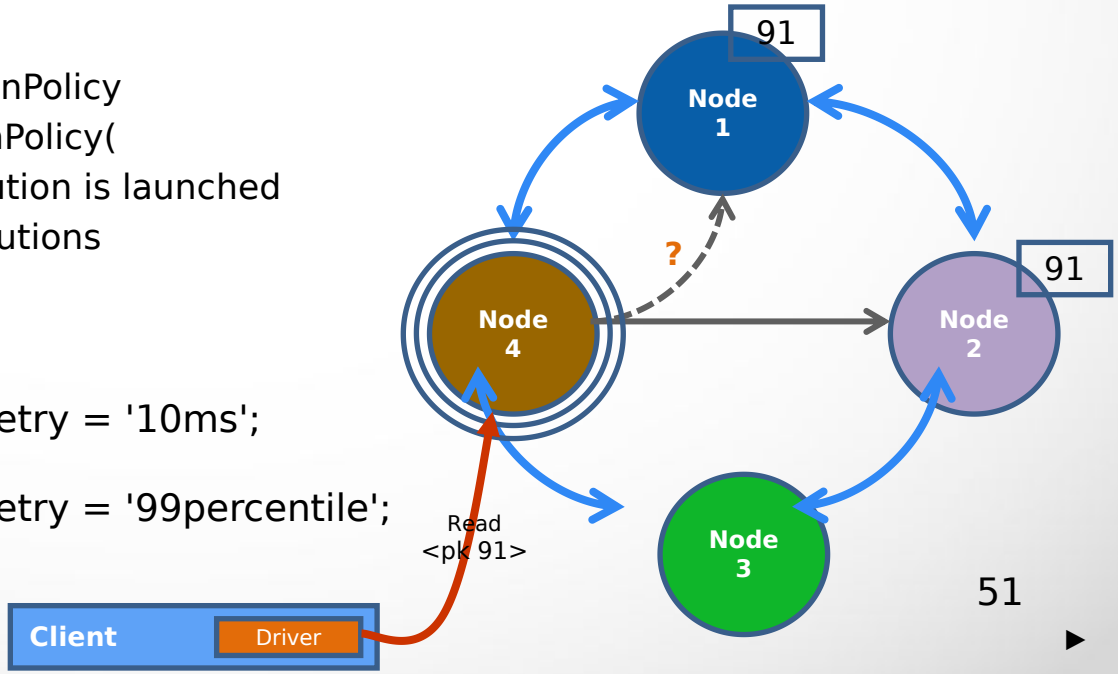
Eager retry

- 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';
```



Last Write Win (LWW)

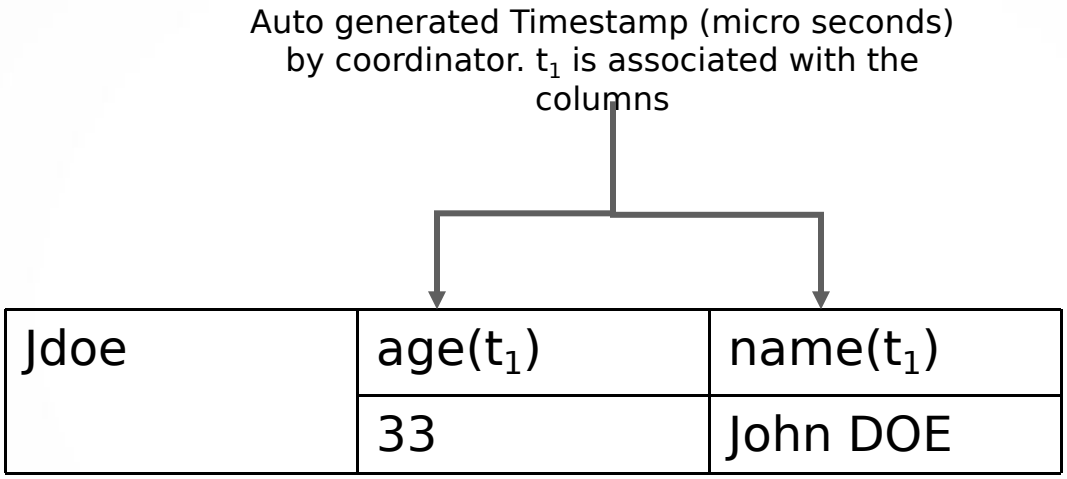
```
INSERT INTO users(login,name,age) VALUES ('jdoe', 'John DOE', 33);
```

#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

SSTable$_1$

Jdoe	age(t_1)	name(t_1)
	33	John DOE

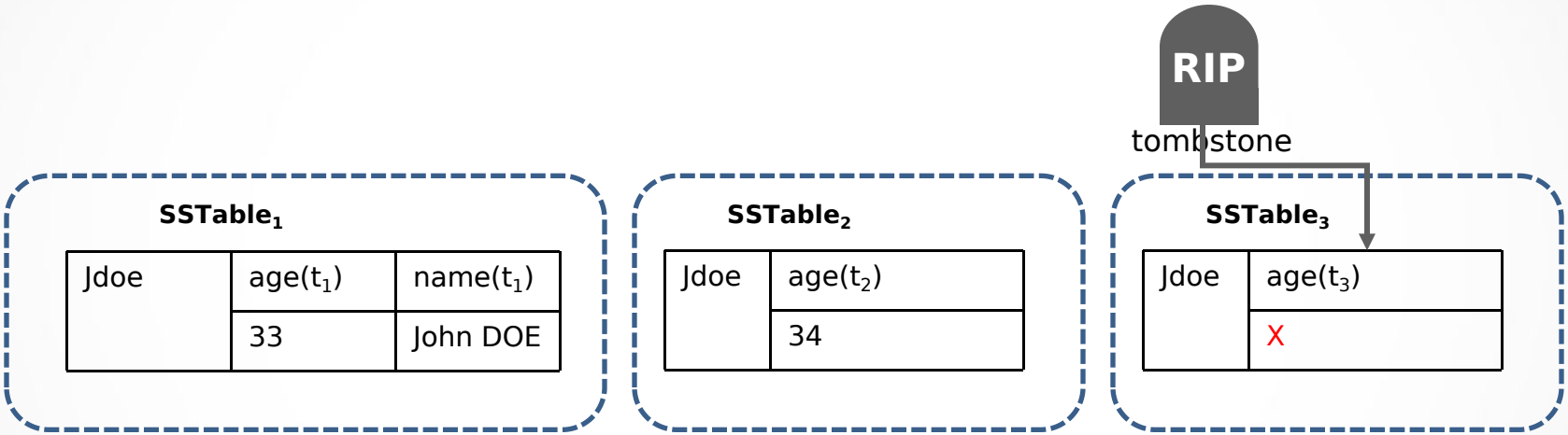
SSTable$_2$

Jdoe	age(t_2)
	34

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

Last Write Win (LWW)

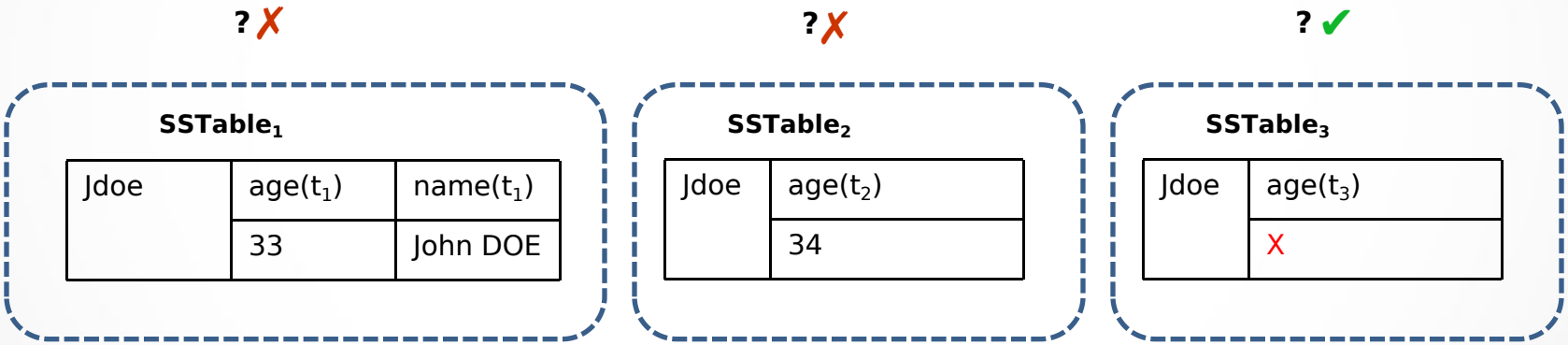
```
DELETE age from users WHERE login = 'jdoe';
```



Last Write Win (LWW)

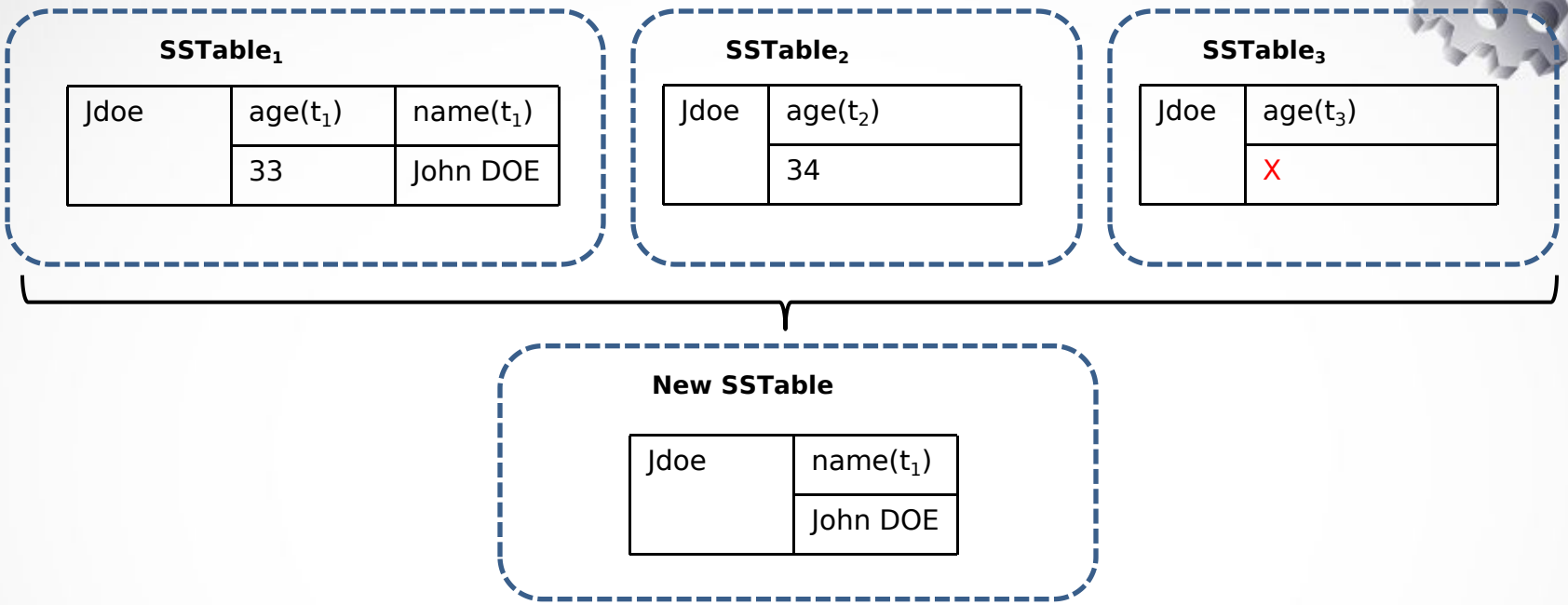
```
SELECT name, age from users WHERE login = 'jdoe';
```

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