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

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

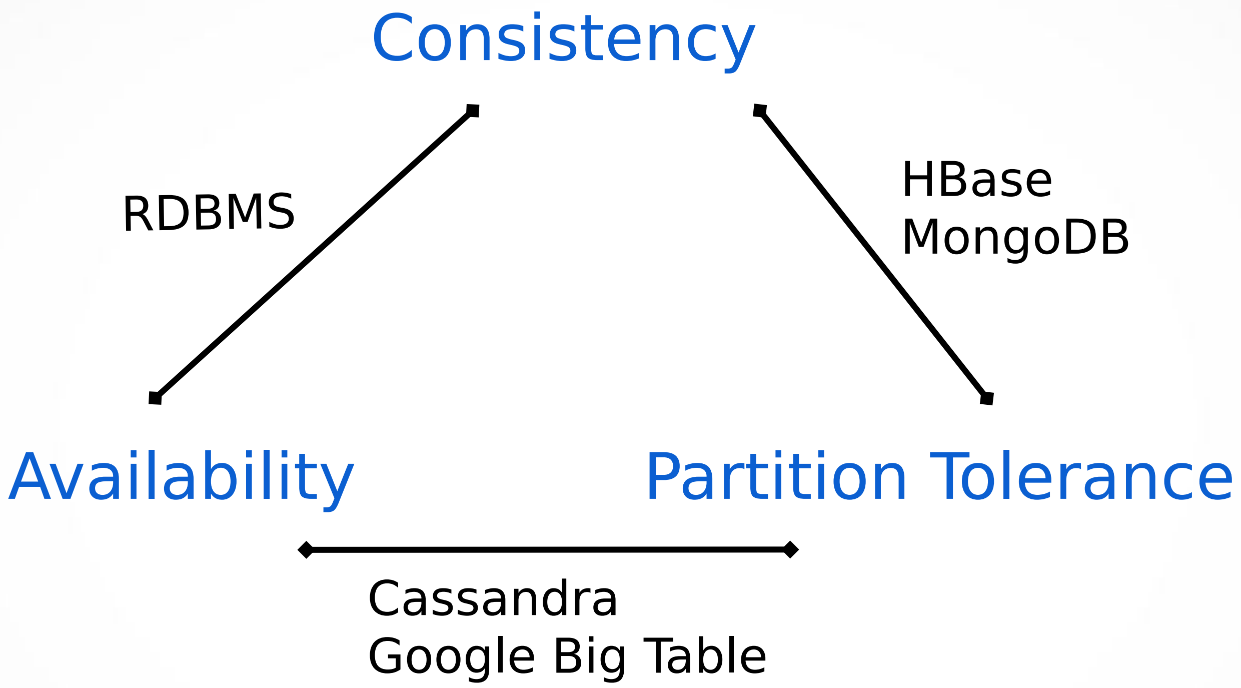
Applicability of SQL / NoSQL

- 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

- V3
- 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.planetcassandra.org/apache-cassandra-use-cases

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

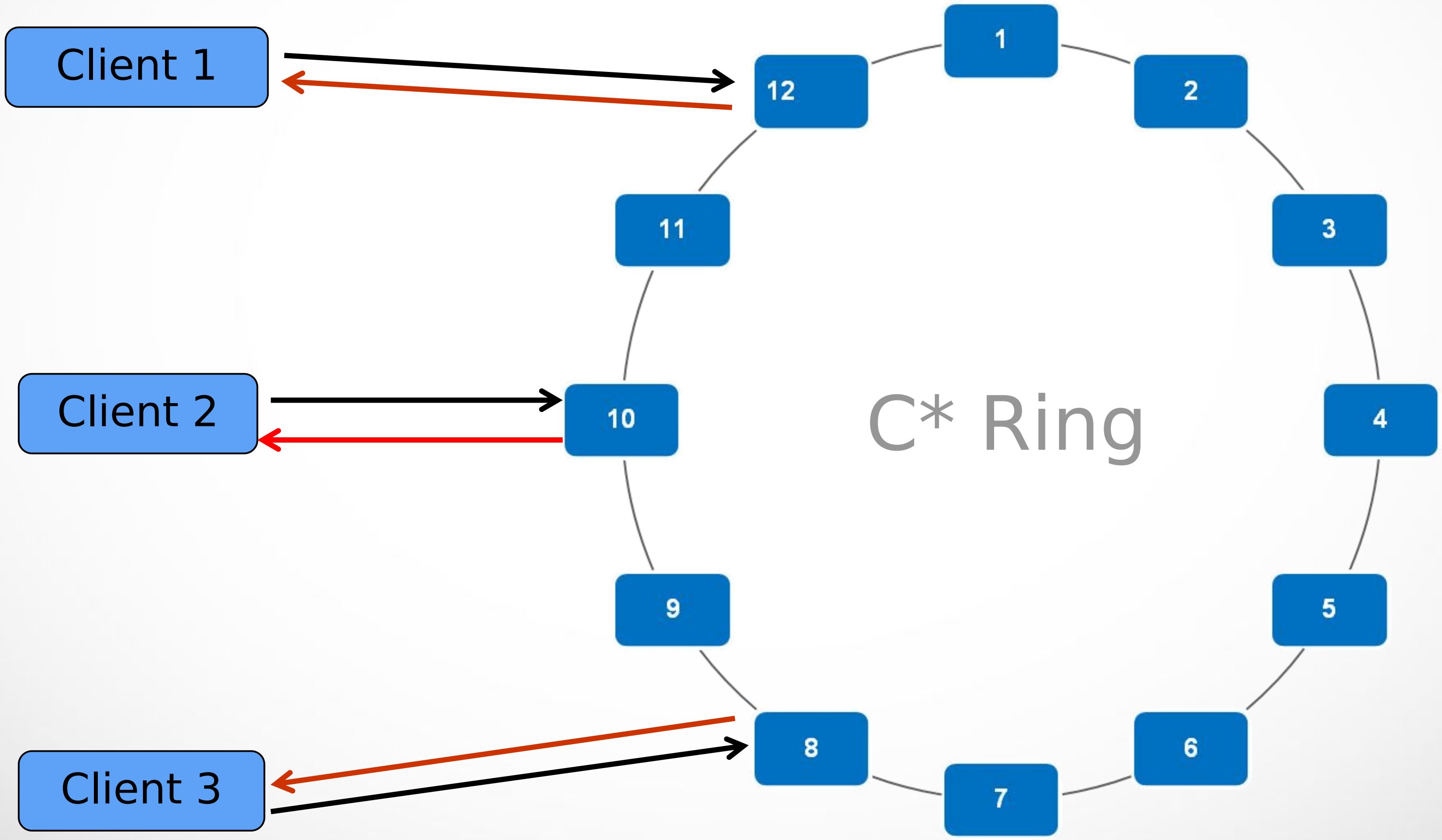
- A few interesting observations about datastore
  - Put = insert/update (Upsert?)
  - Get = select

- 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 communication mechanism among nodes
- 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
  - Load it is under
- 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
- 10% of the time a node may gossip with nodes from the seed list
- 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

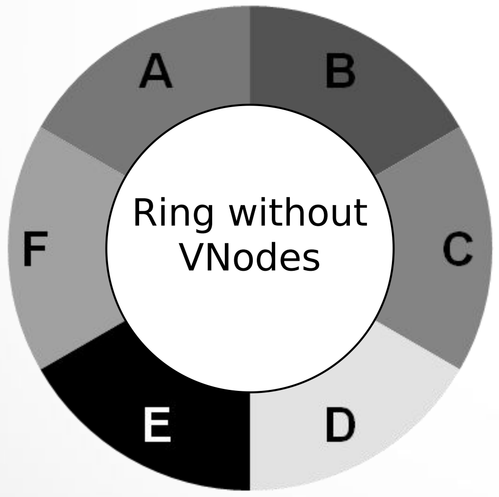
- C* uses a mechanism called "Accrual Failure Detector"
- The basic idea is that a node’s state is not necessarily up or down. What else? "Busy" maybe!
- It is an educated guess which takes multiple factors into account
- A server (node n1) suspects that a node is down because it hasn’t received the 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

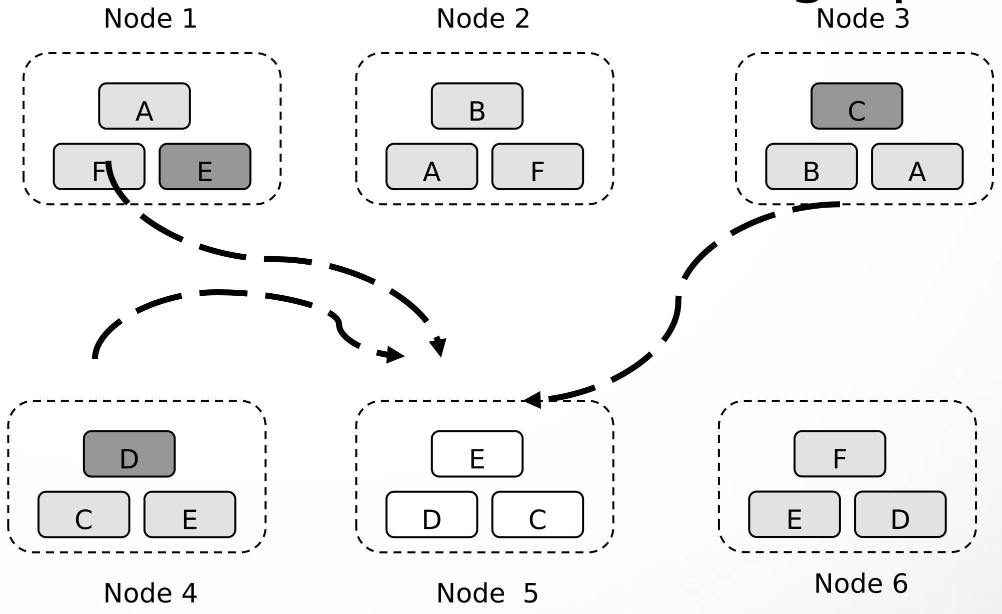
Partitioner

- 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*



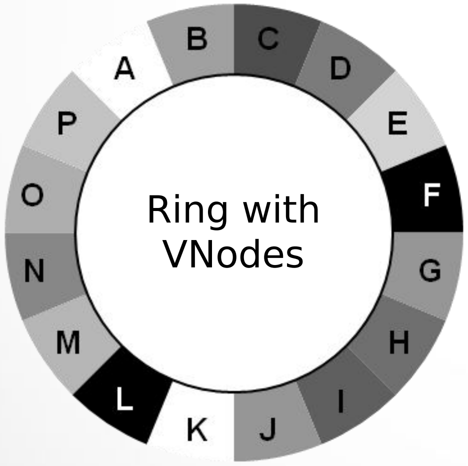


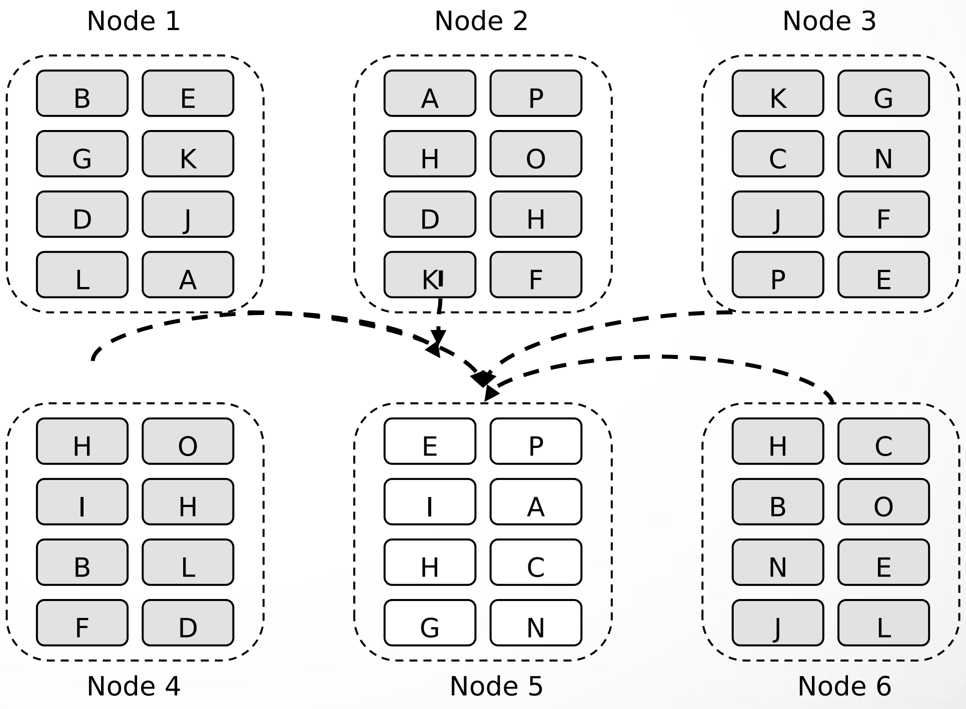
Few peers helping to bring a downed node up, Increased load on the selected peers

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

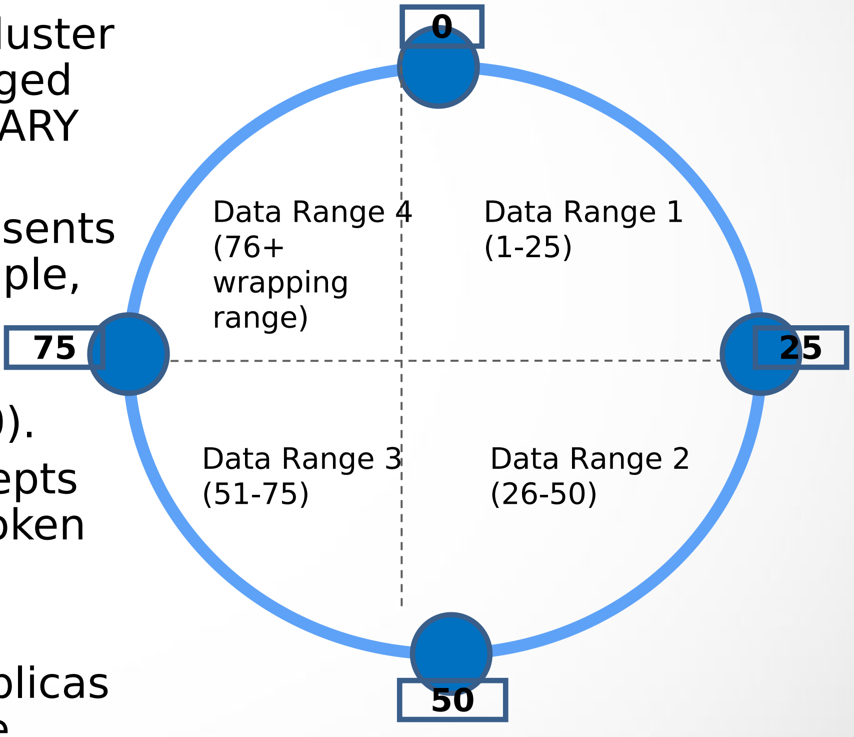
Partitioner and Replication

- 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"

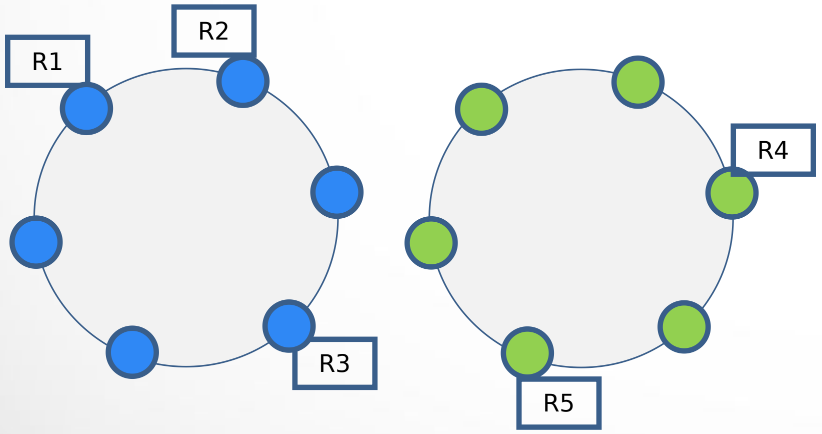
.. a bit later

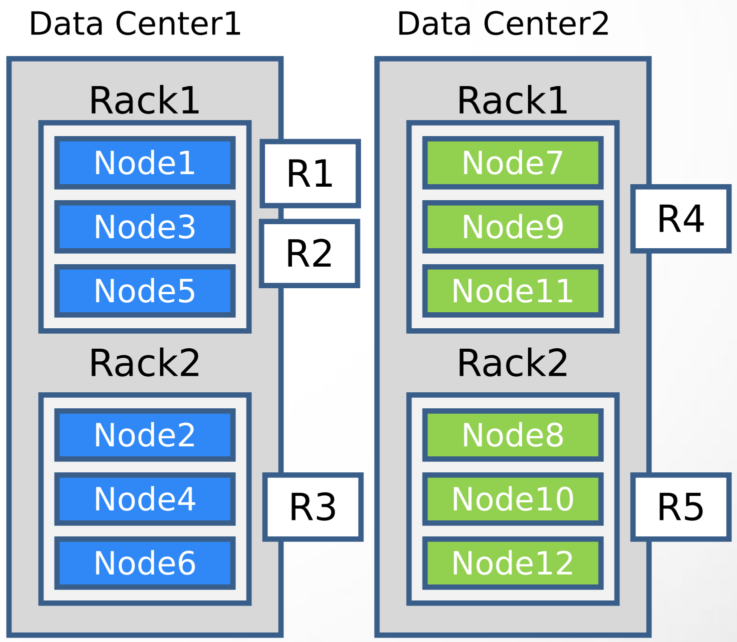


Multi DC replication

- 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.
```

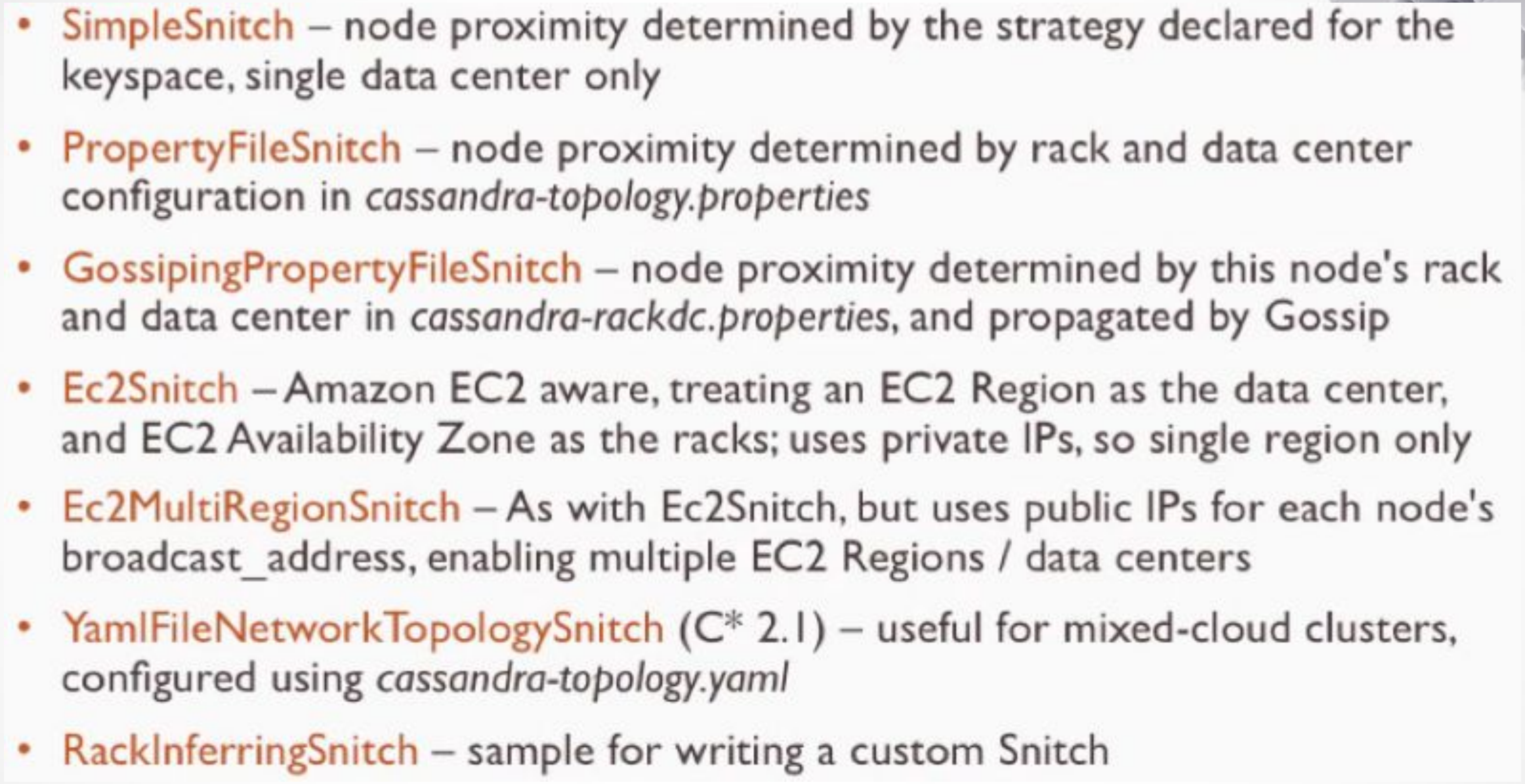




Snitch

- 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

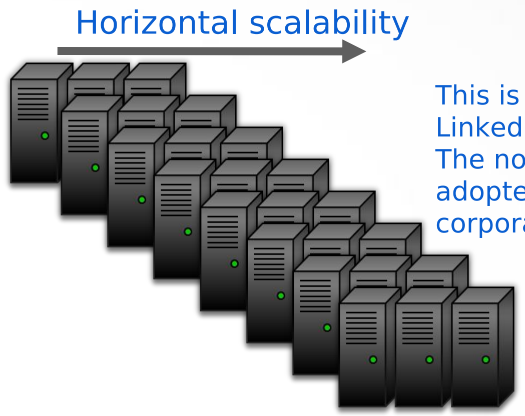
A bit more about property file snitch

- 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

Commodity vs Specialized hardware

Vertical scalability

vs



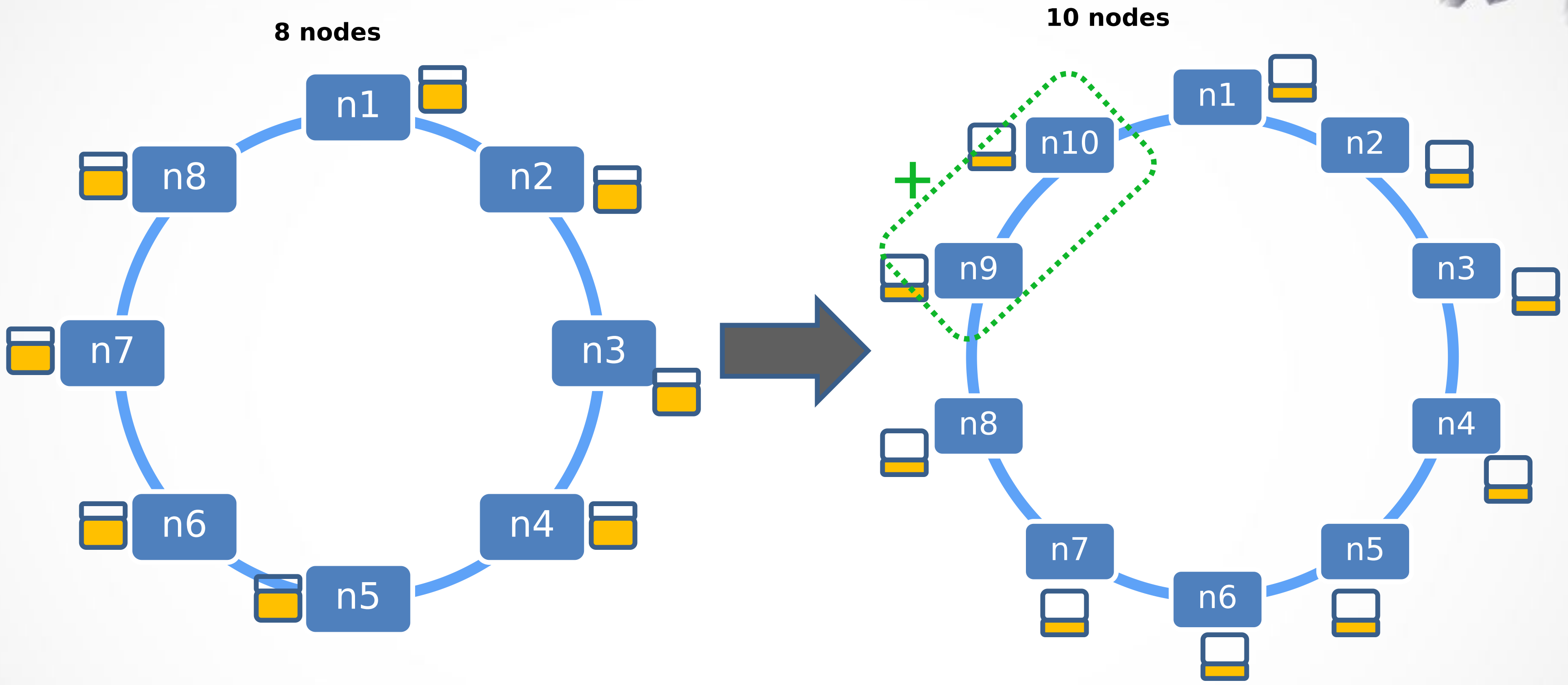
This is what Google, LinkedIn, Facebook do. The norm is now being adopted by large corporations as well.

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

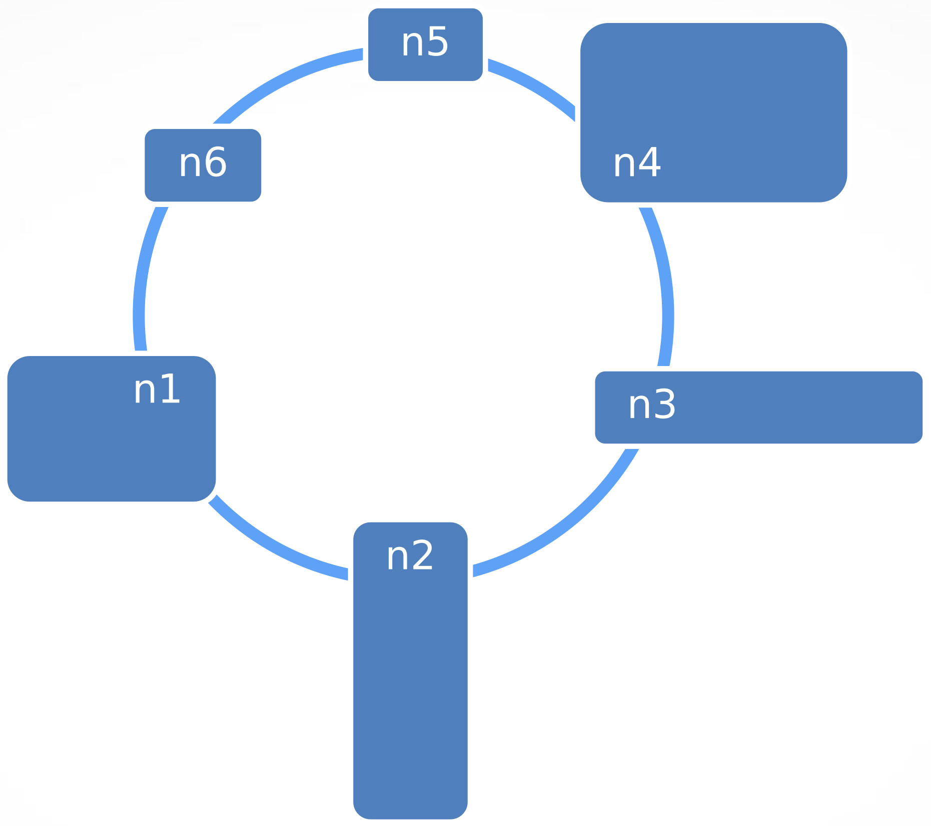
"Just in time" expansion; stay in tune with the load. No need to build ahead in time in anticipation

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?



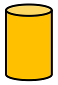
State your observations based on typical machine characteristics
1. CPU, Memory
3. Storage space (consider externally mounted space as local)
3. Network connectivity
4. Query performance and "predictability" of queries

Heterogeneous cluster

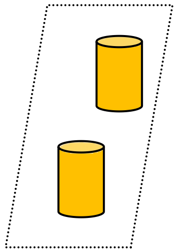
- As time goes on, everyone is going to come to a point where it’s time to replace older, weaker machines with newer, more powerful ones
- While in transition however, it would be nice if the newer nodes could bear more load immediately
- While this may be leveraged (by changing the num_tokens) but it is advisable not to
- See the default value in the C* yaml file. This value may change as C* evolves over time

Deployment - 4 dimensions

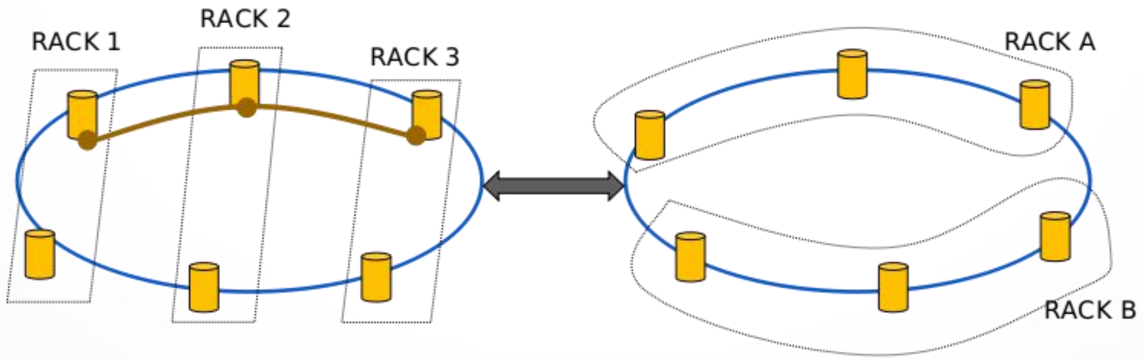
1 Node - One C* instance



2 Rack - Logical set of nodes



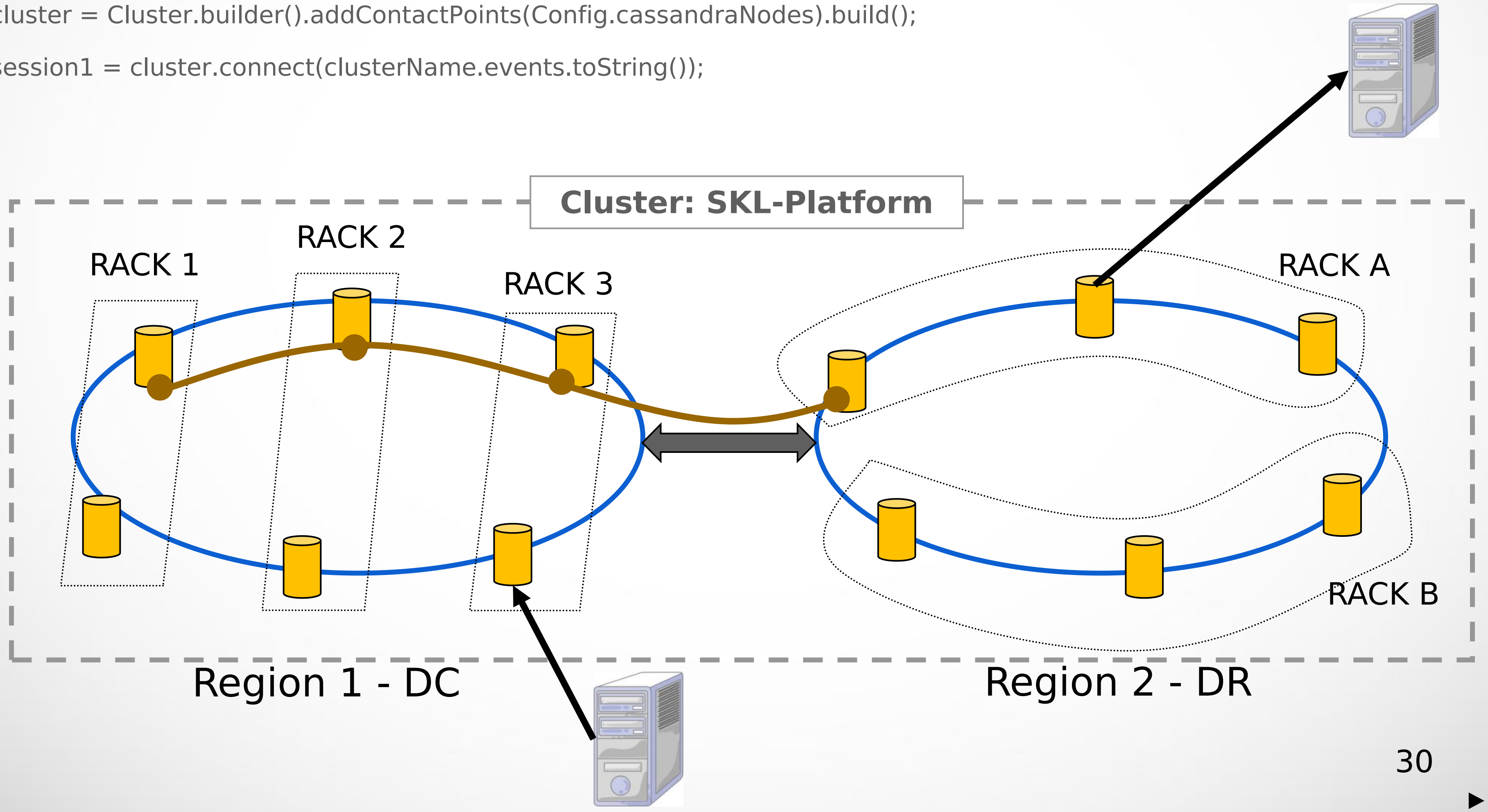
3 DC-DR - Logical set of racks



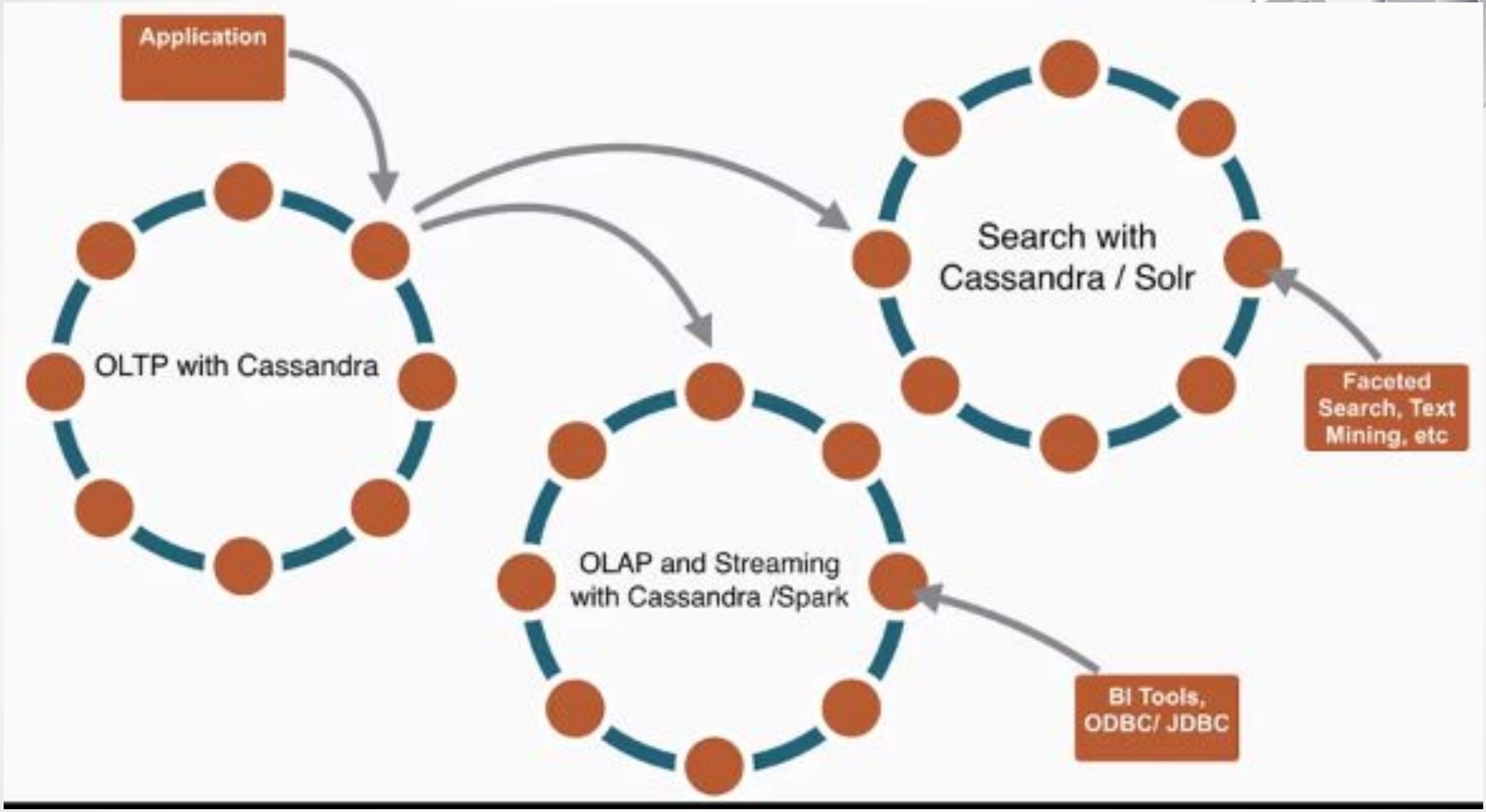
4 Cluster - Nodes that map to a single token ring (can be across DC), next slide ...

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


Application
OLTP with Cassandra
Search with Cassandra / Solr
OLAP and Streaming with Cassandra /Spark
Faceted Search, Text Mining, etc
BI Tools, ODBC/ JDBC

Each ring can be in a different region/availability zone

Regions and Zones

- Many IaaS (Google / Amazon) providers have multiple data centers around the world called "Regions"
- Each region is further divided into availability "zones"
- Depending on your IaaS or even your own DC, you must plan the topology in such a way that
  - A client request should not travel over the network too long to reach C*
  - An outage should not take out a significant chunk of your cluster
  - A replication across DC should not add to the latency for the client

▶

Debate - replication setting

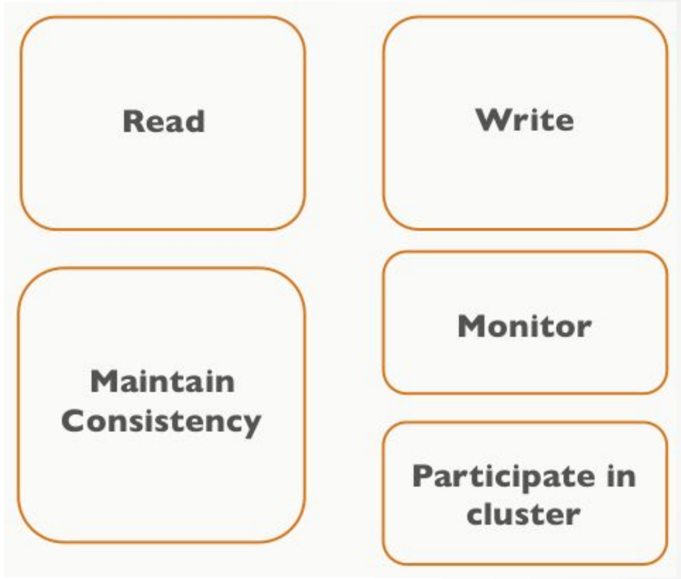
- Let's assume we have the cluster as follows
  - DC1 having 5 nodes
  - DC2 having 5 nodes

- What will happen if the following statements are executed (assume no syntax errors)?
  - create keyspace if not exists test with replication = { 'class' : 'NetworkTopologyStrategy', 'DC1' : 2, 'DC2' : 10 };
  - create table test (col1 int, col2 varchar, primary key(col1));
  - insert into test (col1, col2) values (1, 'Sir Dennis Ritchie');

  - Will this succeed?

SEDA Architecture

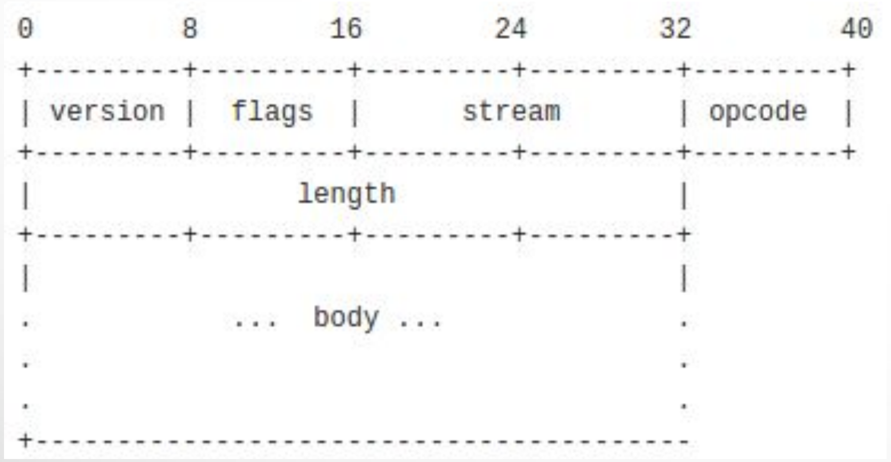
- Staged Event Driven Architecture (SEDA)
- Separates different tasks (gossip, read repair, replication etc) into stages that are connected by a messaging service
- Each like task is grouped into a stage having a queue and thread pool
- So what all pools do we have? Five. The size of the box matters, 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)



```
0         8        16        24        32         40
+---------+---------+---------+---------+---------+
| version |  flags  |      stream       | opcode  |
+---------+---------+---------+---------+---------+
|                length                 |
+---------+---------+---------+---------+
|                                       |
.            ...  body ...              .
.                                       .
.                                       .
+----------------------------------------
```

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

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 ...

PK (Partition key) aka RK (Row key)

- Partition/Row key determines which node the data will reside
  - Simple key
  - Composite key
- It is the unit of replication (atomic set of data) in a cluster
  - All data for a given PK replicates around in the cluster
  - All data for a given PK is co-located on a node
- The Partition/Row Key is hashed
  - Murmer3 hashing is the preferred option and is also the default
  - Random does based on MD5 (not considrerd secure)
  - Others are for legacy support only
- A partition is fetched from the disk in one disk seek, every partition requires a seek

PK
meter_id_001
meter_id_002
meter_id_003

$\rightarrow$

Hash
99985875512
56985412353
-12598745413

More of PK during modeling …

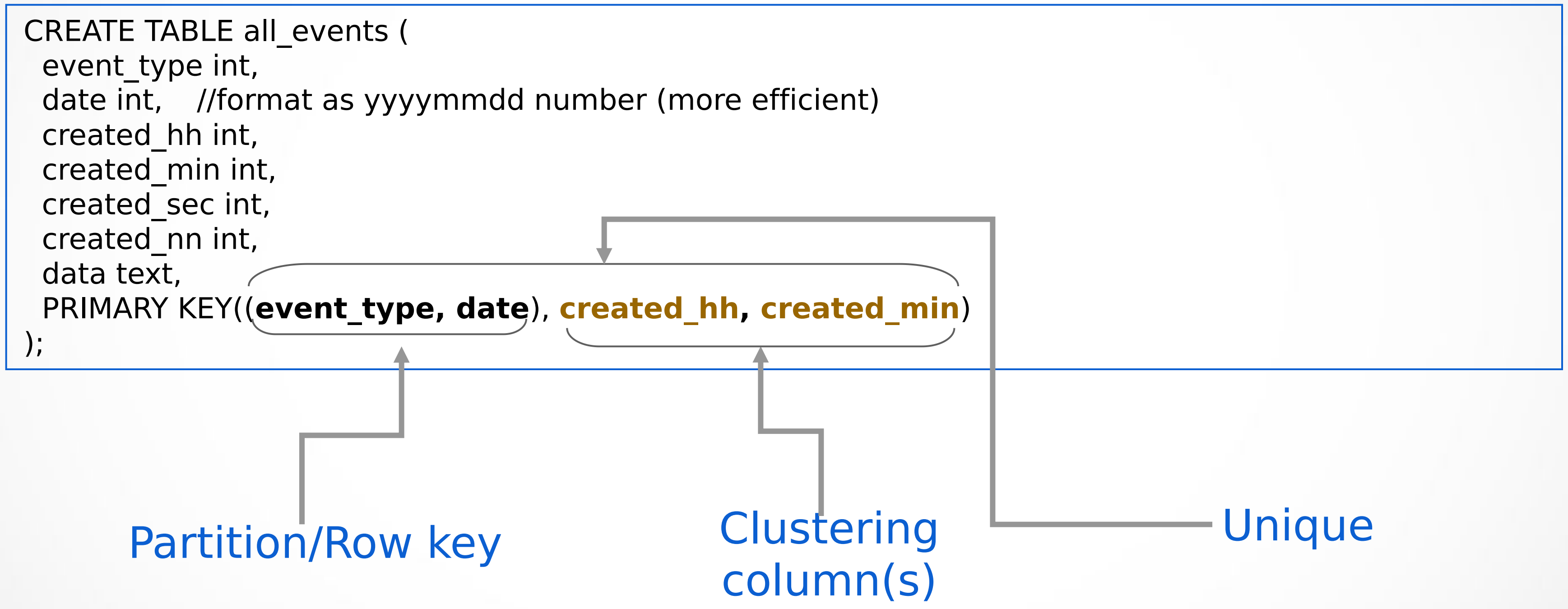
select token(meter_id, year, month, day) as token_value, meter_id, year, month, day from meter_data limit 1;

PK/RK - data partitioning process

- Every node in the cluster is an owner of a range of tokens that are calculated from the PK
- After the client sends the write request the coordinators partitioner uses the configured partitioner to determine the token value
- Then the coordinator looks for the node that has the token range and puts the first replica there
- Subsequent replicas is with respect to the node of the first copy/replica
- Data with the same partition/row key will reside on the same physical node
- Clustering columns (in case of compound PK) does not impact the choice of node
  - Default sort based on clustering columns is ASC

Visualizing the Primary Key

What are the components of Primary key?

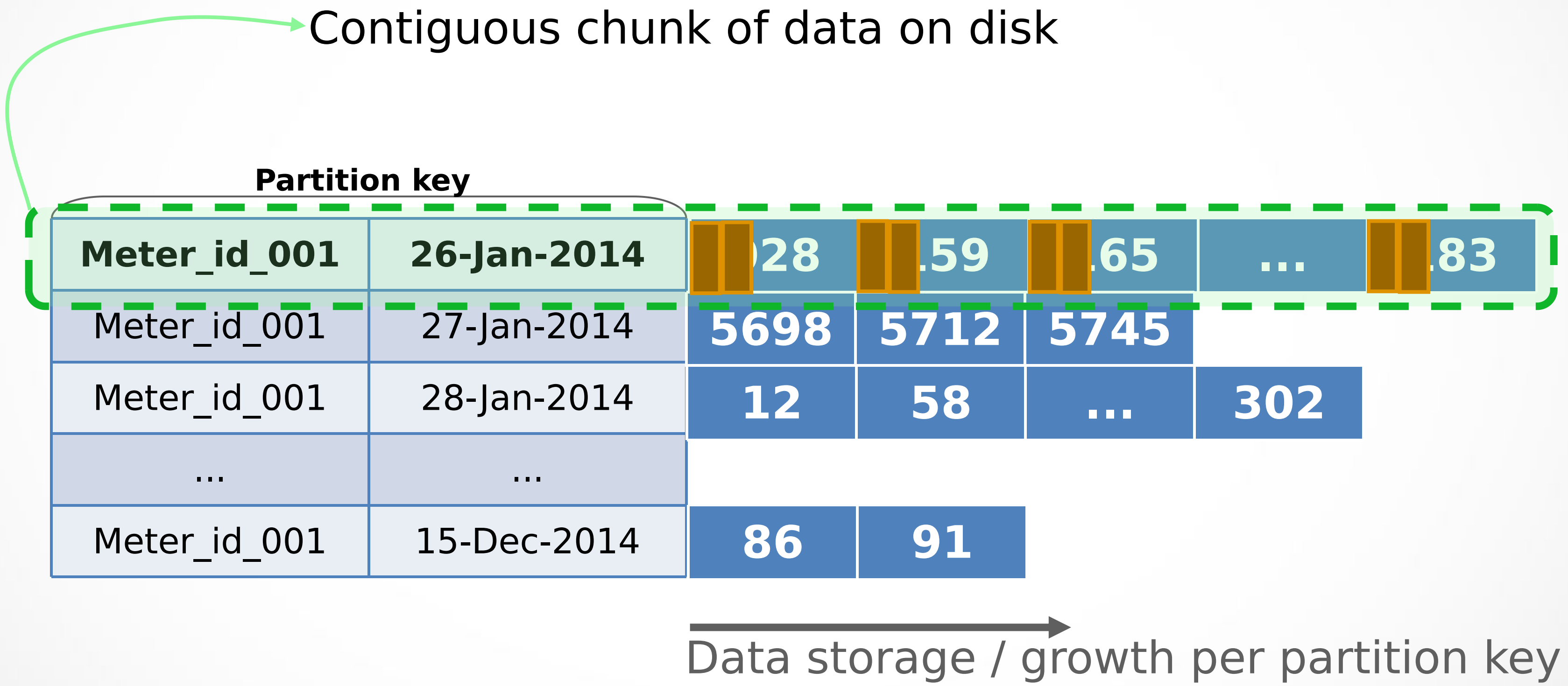


```
CREATE TABLE all_events (
  event_type int,
  date int,    //format as yyyymmdd number (more efficient)
  created_hh int,
  created_min int,
  created_sec int,
  created_nn int,
  data text,
  PRIMARY KEY((event_type, date), created_hh, created_min)
);
```

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?

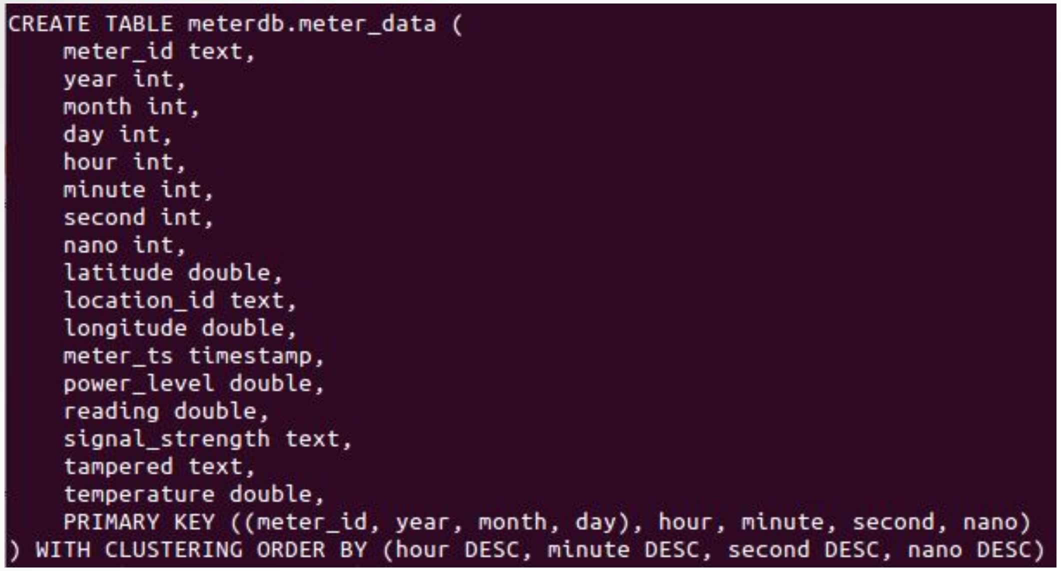


Note: C* creates a **key/name** using the clustering column values with regular column names and then stores the **value** of the regular column as a "**cell**". Visuals on next slide...

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)

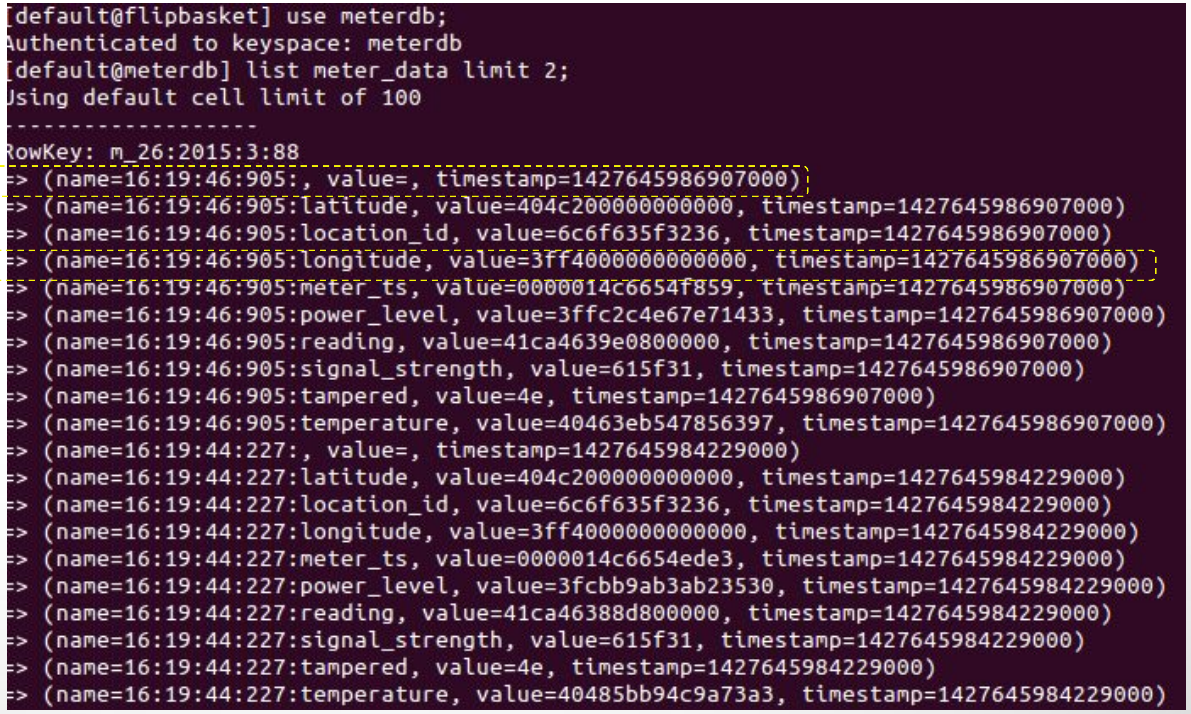
Cell based storage - visuals (pre v3.0)



```
CREATE TABLE meterdb.meter_data (
    meter_id text,
    year int,
    month int,
    day int,
    hour int,
    minute int,
    second int,
    nano int,
    latitude double,
    location_id text,
    longitude double,
    meter_ts timestamp,
    power_level double,
    reading double,
    signal_strength text,
    tampered text,
    temperature double,
    PRIMARY KEY ((meter_id, year, month, day), hour, minute, second, nano)
) WITH CLUSTERING ORDER BY (hour DESC, minute DESC, second DESC, nano DESC)
```

Notice a blank cell with only the key containing clustering key values are stored

cell



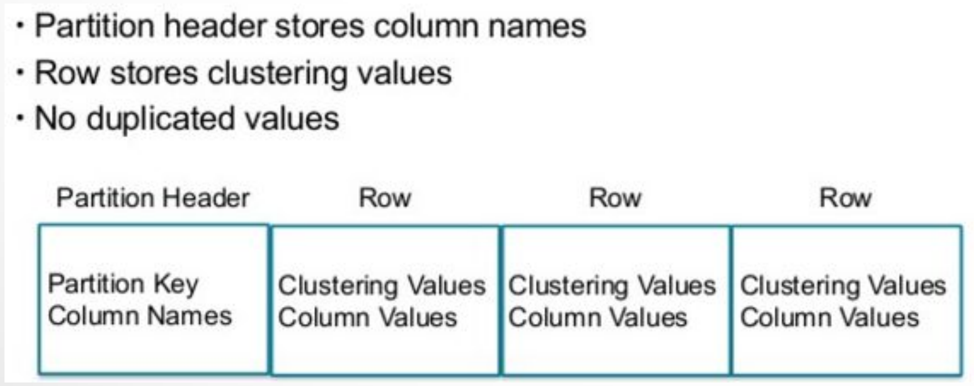
```
[default@flipbasket] use meterdb;
Authenticated to keyspace: meterdb
[default@meterdb] list meter_data limit 2;
Using default cell limit of 100
-------------------
RowKey: m_26:2015:3:88
=> (name=16:19:46:905:, value=, timestamp=1427645986907000)
=> (name=16:19:46:905:latitude, value=404c200000000000, timestamp=1427645986907000)
=> (name=16:19:46:905:location_id, value=6c6f635f3236, timestamp=1427645986907000)
=> (name=16:19:46:905:longitude, value=3ff4000000000000, timestamp=1427645986907000)
=> (name=16:19:46:905:meter_ts, value=0000014c6654f859, timestamp=1427645986907000)
=> (name=16:19:46:905:power_level, value=3ffc2c4e67e71433, timestamp=1427645986907000)
=> (name=16:19:46:905:reading, value=41ca4639e0800000, timestamp=1427645986907000)
=> (name=16:19:46:905:signal_strength, value=615f31, timestamp=1427645986907000)
=> (name=16:19:46:905:tampered, value=4e, timestamp=1427645986907000)
=> (name=16:19:46:905:temperature, value=40463eb547856397, timestamp=1427645986907000)
=> (name=16:19:44:227:, value=, timestamp=1427645984229000)
=> (name=16:19:44:227:latitude, value=404c200000000000, timestamp=1427645984229000)
=> (name=16:19:44:227:location_id, value=6c6f635f3236, timestamp=1427645984229000)
=> (name=16:19:44:227:longitude, value=3ff4000000000000, timestamp=1427645984229000)
=> (name=16:19:44:227:meter_ts, value=0000014c6654ede3, timestamp=1427645984229000)
=> (name=16:19:44:227:power_level, value=3fcbb9ab3ab23530, timestamp=1427645984229000)
=> (name=16:19:44:227:reading, value=41ca46388d800000, timestamp=1427645984229000)
=> (name=16:19:44:227:signal_strength, value=615f31, timestamp=1427645984229000)
=> (name=16:19:44:227:tampered, value=4e, timestamp=1427645984229000)
=> (name=16:19:44:227:temperature, value=40485bb94c9a73a3, timestamp=1427645984229000)
```

v3.0+ Moving away from Dynamo inspiration?

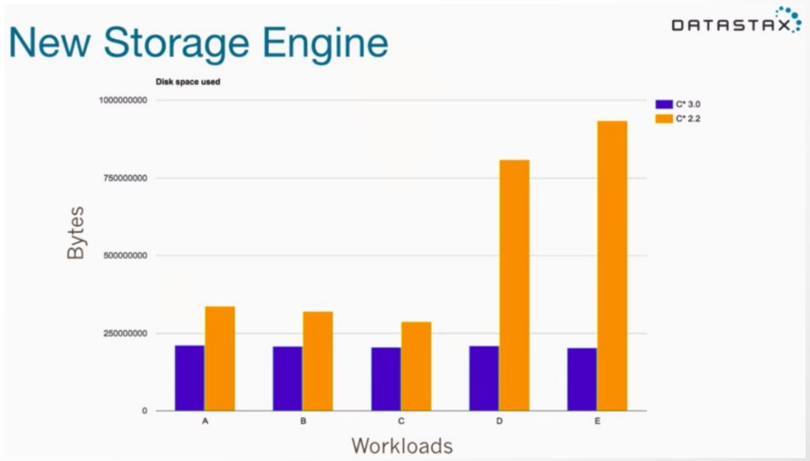
The motivation

- Make the storage engine more aware of the CQL structure. In practice, instead of having partitions be a simple iterable map of cells, it should be an iterable list of row (each being itself composed of per-column cells, though obviously not exactly the same kind of cell we have today)

- Make the engine more iterative. What I mean here is that in the read path, we end up reading all cells in memory (we put them in a ColumnFamily object), but there is really no reason to. If instead we were working with iterators all the way through, we could get to a point where we're basically transferring data from disk to the network, and we should be able to reduce GC substantially



- Partition header stores column names
- Row stores clustering values
- No duplicated values



v3.0+ Row structure

- For a given partition, the format simply serialize rows one after another (atoms in practice). For the on-disk format, this means that it is now rows that are indexed, not cells.
- The format uses a header that is written at the beginning of each partition for the on-wire format and is kept as a new sstable Component for sstables. The following are the differences compared to the current format:
  - Clustering values are only serialized once per row (we even skip serializing the number of elements since that is fixed for a given table)
  - Column names are not written for every row. Instead they are written once in the header. For a given row, we support two small variant: dense and sparse.
  - When dense, cells come in the order the columns have in the header, meaning that if a row doesn't have a particular column, this column will still use a byte.
  - When sparse, we don't have anything if the column doesn't have a cell, but each cell has an additional 2 bytes which points into the header (so with vint it should rarely take more than 1 byte in practice).
  - The variant used is automatically decided based on stats on how many columns set a row has on average for the source we serialize.

- Values for fixed-width cell values are serialized without a size.
- If a cell has the same timestamp than its row, that timestamp is not repeated for the cell. Same for the ttl (if applicable).
- Timestamps, ttls and local deletion times are delta encoded so that they are ripe for vint encoding. The current version of the patch does not yet activate vint encoding however (neither for on-wire or on-disk).

https://github.com/apache/cassandra/blob/trunk/guide_8099.md#storage-format-on-disk-and-on-wire
Good way to see the structure is by using SSTableDump utility

Let's watch this

- https://www.youtube.com/watch?v=daN7V-lT7M4

This Week In Cassandra: 3.0 Storage Engine Deep Dive 3/11/2016

Inspecting the structure!

- Create a keyspace - stockdb
- Create a table users
- Insert a few records
- https://github.com/tolbertam/sstable-tools
  - Git clone and build using mvn
  - $ java -jar sstable-tools-3.7.0-alpha7-SNAPSHOT.jar cqlsh
  - cqlsh>use /opt/cassandra-data/data/stockdb/user-d2af5a80eb7011e58737a736dc04913a/ma-1-big-Data.db
  - cqlsh>dump limit 10
  - describe sstables