Big Data Systems

Djellel Difallah

Spring 2023

Lecture 5 (cont.) – Apache Cassandra

Some content is courtesy of Benoit Perroud
CTO at Sqooba and Apache Committer

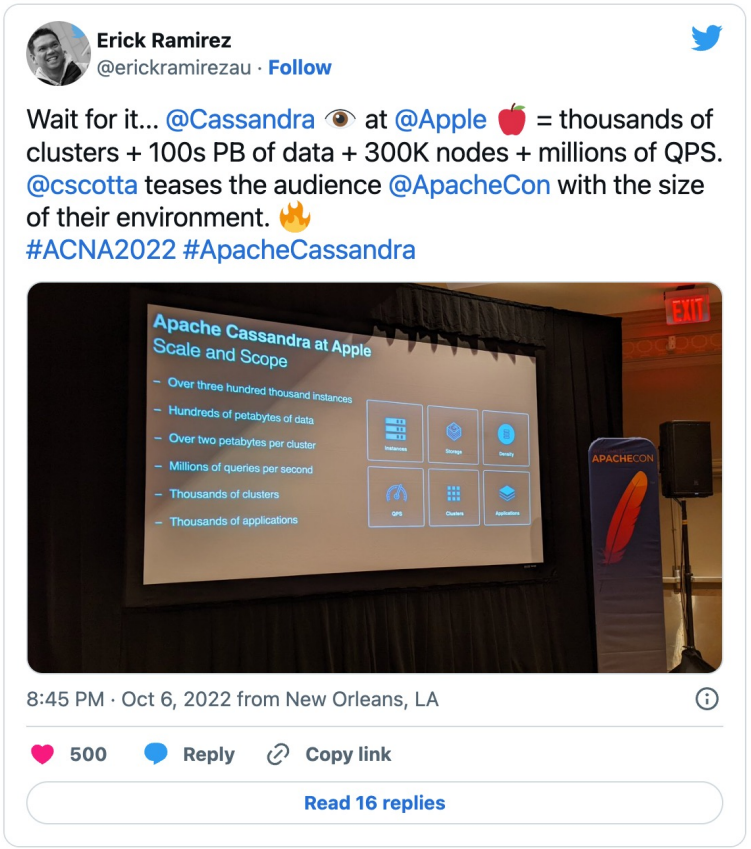
Agenda

- Apache Cassandra Overview
- Design principles
- Data Model
- Query Model
- Common Use Cases

Apache Cassandra

- Apache Cassandra could be simplified as a **scalable, distributed, sparse and eventually consistent hash map**. *But it's actually way more*.
- Originally developed by Facebook, hit ASF incubator early 2008, version 1.0 in 2010
- Inspired from Amazon Dynamo and Google BigTable
- Current version 4.1
- Now developed by many companies: Datastax, Apple, Netflix, Twitter, …

Some numbers

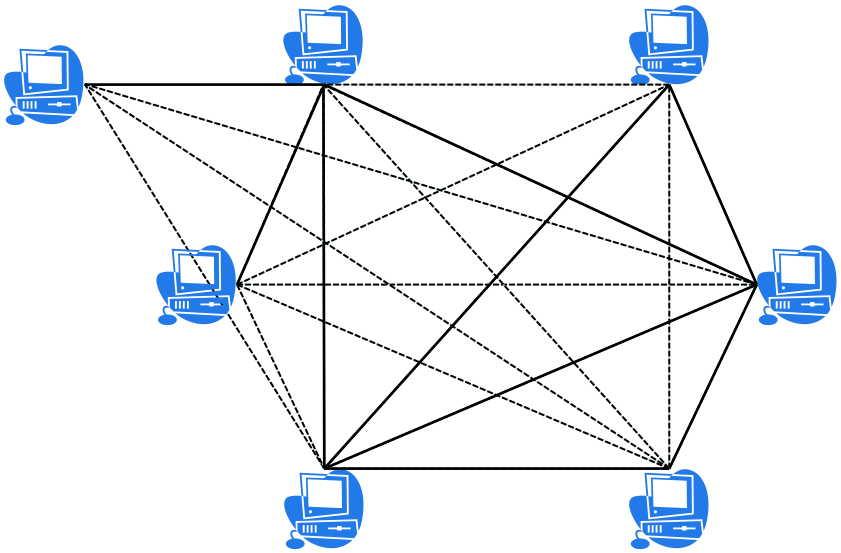


Apache Cassandra Gossip Protocol

- Cassandra is a decentralized system (Dynamo)
  - Nodes can be added and removed easily (highly scalable!)
- Nodes need to exchange state information (heartbeats), and metadata about the database.
- Node discovery is done using a **gossip protocol**
  - Think about word of mouth
- The gossip protocol enables decentralized communication among nodes in a network, without relying on a central point of control.
  - Used in peer-to-peer systems or distributed
  - Help to prevent network congestion and balance network load

Gossip Protocol

- Information is exchanged between subsets of nodes: Think about word of mouth.
- Each node maintains a list of other nodes that it communicates with directly.
- Periodically, a node selects random nodes from its list to send messages.

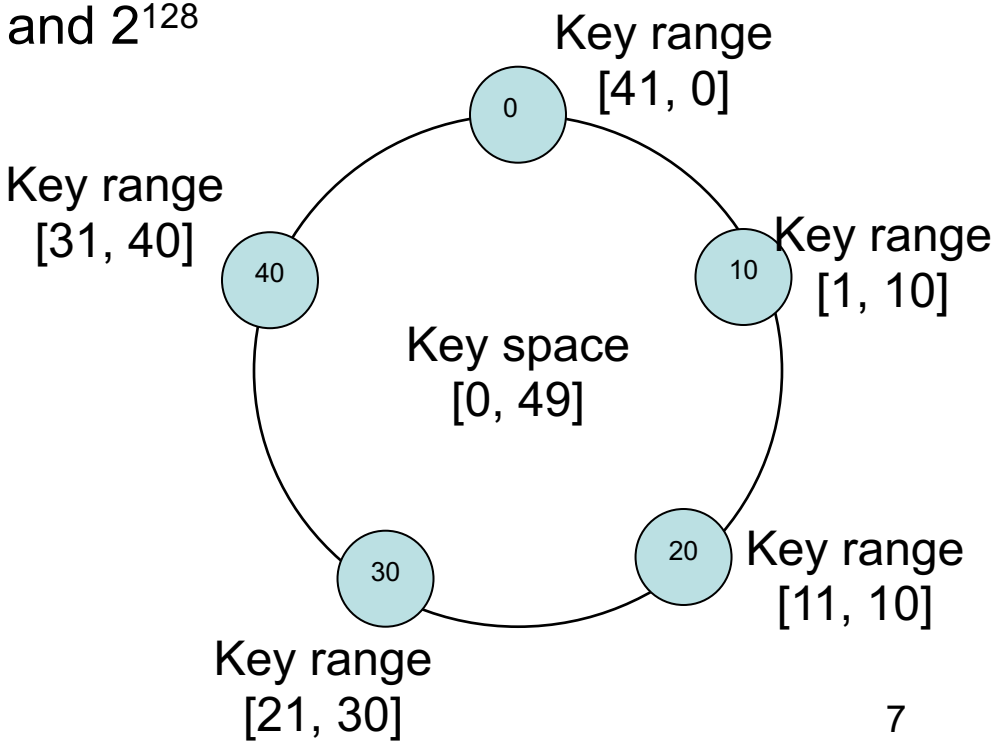


No single point of failure!

Scalable: $O(\log_m(N))$
where m is the number of connection
and N is the size of the cluster.

Apache Cassandra is a scalable **distributed**, sparse, eventually consistent hash map

- Key distribution based on **consistent hashing**
  - Nodes responsible for key range and replica sets
- Hash function return numbers on 128 bits (Murmur3)
  - i.e. a number between 0 and 2^{128}



No single point of failure!

Apache Cassandra is a scalable distributed, sparse, **eventually consistent** hash map

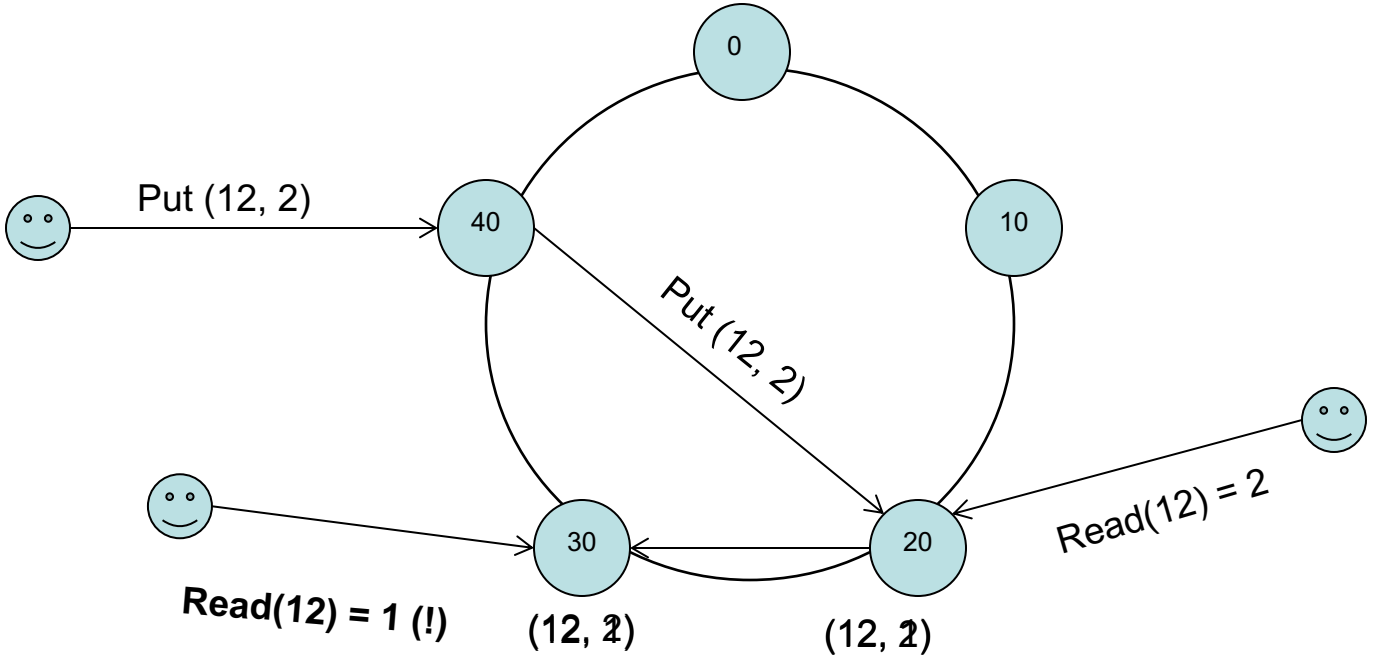
- *"A **quorum** is the minimum number of votes that a distributed transaction has to obtain in order to be allowed to perform an operation in a distributed system. A quorum-based technique is implemented to enforce consistent operation in a distributed system." [Wikipedia]*
- In Cassandra, the default consistency level's is **ONE** for all write and read operations.
- **Quorum** : R + W > N *(Tune the consistency level)*
  - N : number of replica, R : number of node read, W : number of node written.
  - R = 1, W = N
  - R = N, W = 1
  - R = N/2, W = N/2 (+1 if N is even)

N = 3
R = 1, W = 3
R = 2, W = 2

N = 5
R = 1, W = 5
R = 2, W = 4
R = 3, W = 3
…

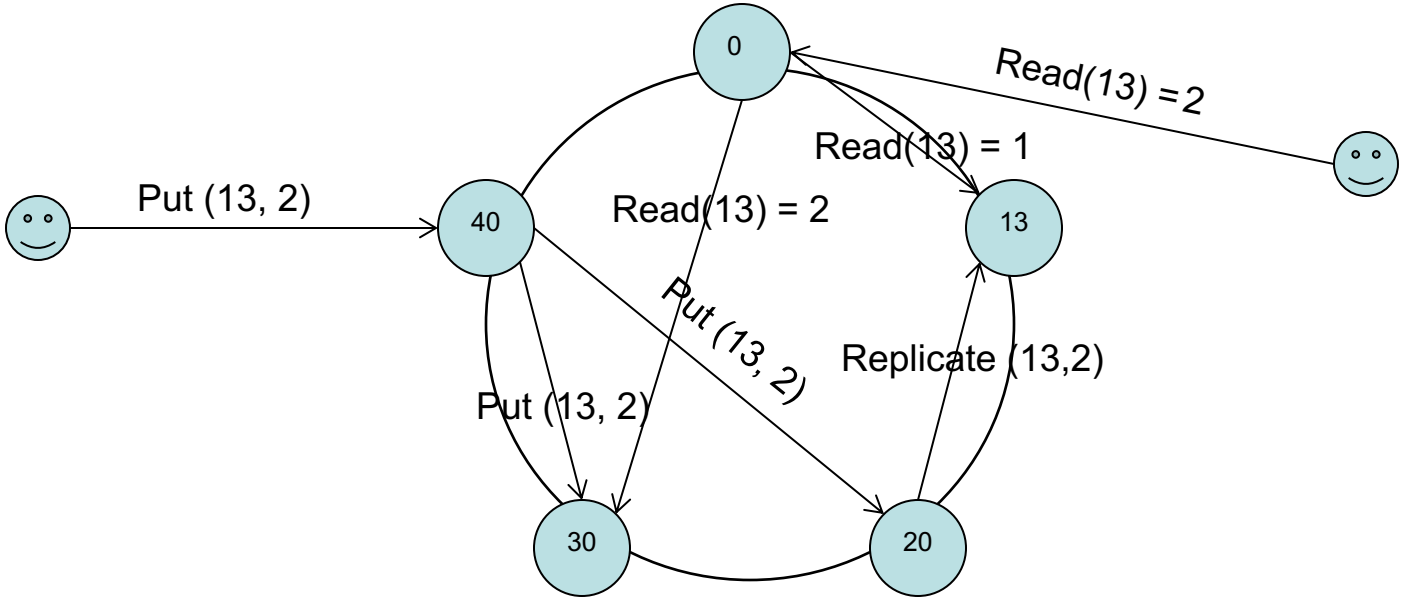
Apache Cassandra is a scalable distributed, sparse, **eventually consistent** hash map

- Key space [0,49]
- PUT(Key:12, Value:1)
- Replication factor 2
- Consistency Level : ONE (R = 1, W = 1)



Apache Cassandra is a scalable distributed, sparse, **eventually consistent** hash map

- Key space [0,49], previously put(13, 1)
- Replication factor 3
- Consistency : QUORUM (R = 2, W = 2)



DATA MODEL

Schema

- ~~Schemaless~~ Flexible Schema

ObjectA { id, name, color, anotherattribute, list<timestamp>, set<string>, map<string, long>, … }

A schema (metadata) is necessary

It might contains generic fields like map<text, text>

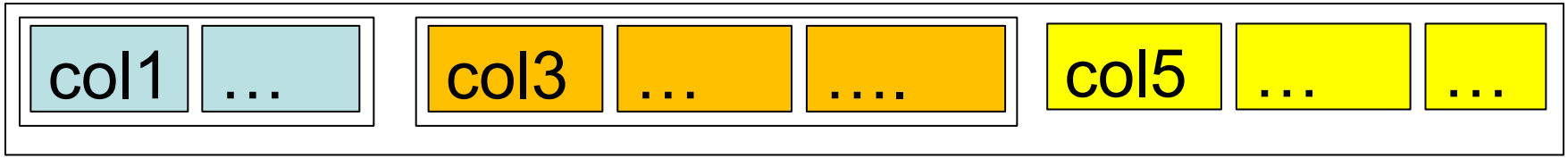
Data Model

- Can be seen as a multilevel hash map : Hash of Hash Map
  - 2 levels of keys.

- Keyspace > Table > row > column name = value
  - # use Keyspace1;
  - # set Table1['key1']['columName1'] = 'value1';
  - # get Table1['key1']['columName1’];

Map<RowKey, SortedMap<ColumnKey, ColumnValue>>

Data Model



Partition Keys Clustering Keys Payload

```
CREATE TABLE IF NOT EXISTS Student (
  sid         int,
  name        TEXT,
  email       TEXT,
  PRIMARY KEY ((email, sid), name)
);
```

Relational vs. Cassandra

Relational	Cassandra Model
Database	Keyspace
Table	Column Family
Primary Key	Row Key
Column Name	Column name / Key
Column value	Column value (cell)

Data Model : Keyspace

- Equivalent to database name in SQL world
- Define replication factor and network topology
  - Network topology include multi datacenters topology
  - Replication factor can be defined per datacenters

```
create keyspace bds with replication = {'class':
'NetworkTopologyStrategy', 'replication_factor':
‘5’};
```

  - Replication strategy:
    - SimpleStrategy (1 Datacenter)
    - NetworkTopologyStrategy (More)

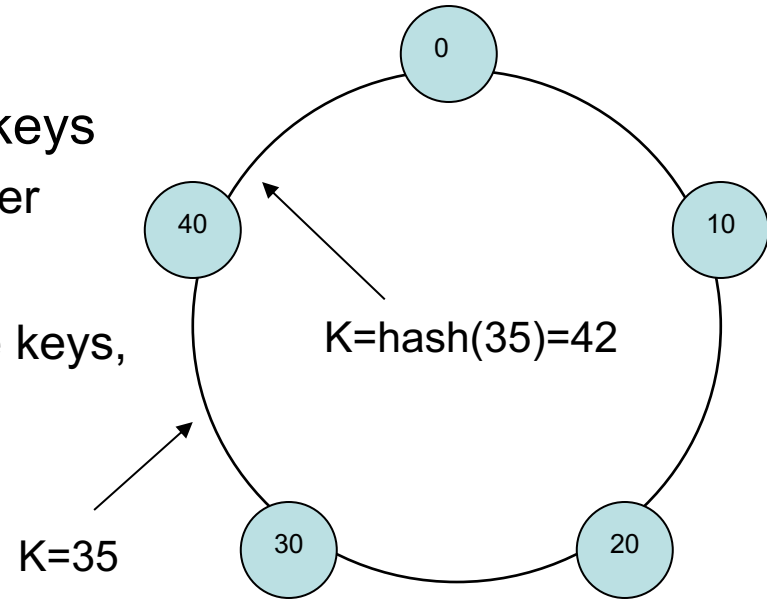
Data Model : Table

- Also called *Column Family* for historical reason
- Really equivalent to tables in SQL world

- Define
  - Type of the keys
  - Column name comparator
  - Additional metadata (types of certain known columns)

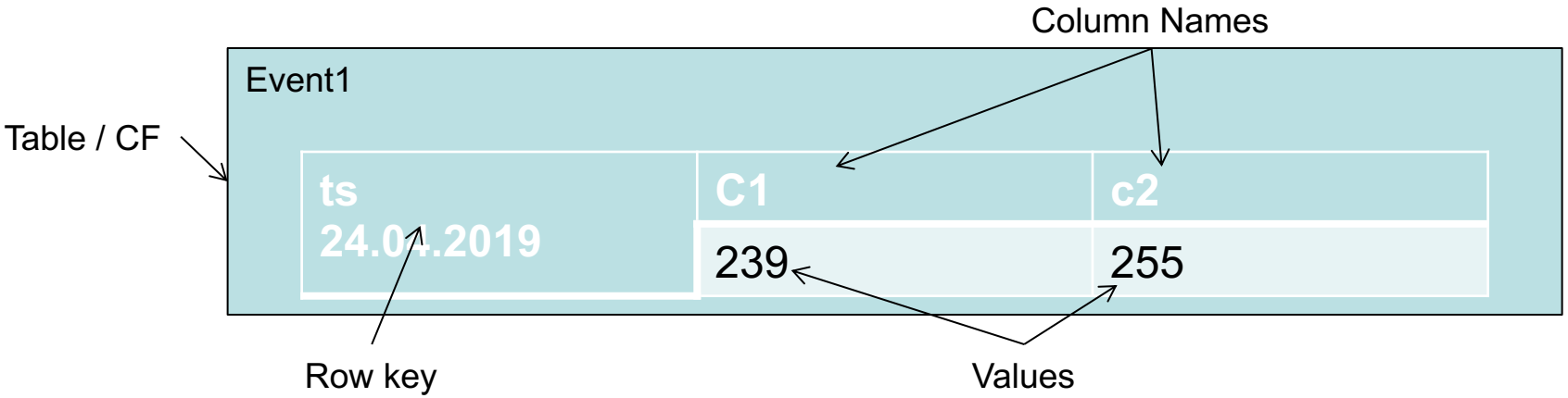
Data Model : Row

- Uniquely defined by the primary key.
  - Eventually stored to a node and its replicas
- Keys are typed
- 2 strategies of key partitioner (hash function) on the key space
  - Random partitioner evenly distribute keys
    - Murmur3Partitioner, RandomPartitioner
  - ByteOrderedPartitioner :
    - Keep order while iterating through the keys, may lead to hot spots



Data Model : Column Name

- Could be seen as column in SQL world



Data Model : Value

- Can be types, seen as array of bytes otherwise
- Existing types include
  - Bytes (blob)
  - Strings (ASCII or UTF-8 strings)
  - Integer, Long, Float, Double, Decimal
  - UUID, dates
  - Counters (of long)
  - UDT (User Defined Type, i.e. struct)
- Can expire (TTL: time to live)
- No foreign keys / no joins!

http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cql_data_types_c.html

Query Model

- Interact with Cassandra
  - CQL (Cassandra Query Language)
    - SQL-like interface to query the data

Query Model

- Cassandra is more than a key/value store.
  - Get
  - Put
  - Delete
  - Update
  - But also various **range queries**
    - Key range
    - Column range (slice)
  - Secondary indexes

Model Illustration

- create TABLE model
  (k text, r text, v1 int, v2 int, v3 int, v4 int, PRIMARY KEY (k, r));

Query Model : Get

- Get single key
  - Give me key ‘a’
- Get multiple keys
  - Give me rows for keys ‘a’, ‘c’, ‘d’ and ‘f’

Select * from model where
k = ‘a’

Select * from model where
k IN (‘a’, ‘c’, ‘d’, ‘f’)

Ordered regarding column name comparator →

RandomPartitionner ↓

	‘1’	‘2’	‘3’	‘4’	‘5’
‘c’	8	9	10		11
‘e’		12	13		14
‘f’	15			16	17
‘a’		18			
‘b’	19	20			20
‘d’	22	23	24	25	26

Query Model : Get Range

- Range
  - Query for a range of key
    - Give me all rows with keys between ‘c’ and ‘f’.
    - Mind the partitioner.

Select * from model where token(k) >= token(‘c’) and token(k) <= token(‘f’)

	‘1’	‘2’	‘3’	‘4’	‘5’
‘c’	8	9	10		11
‘e’		12	13		14
‘f’	15			16	17
‘a’		18			
‘b’	19	20			20
‘d’	22	23	24	25	26

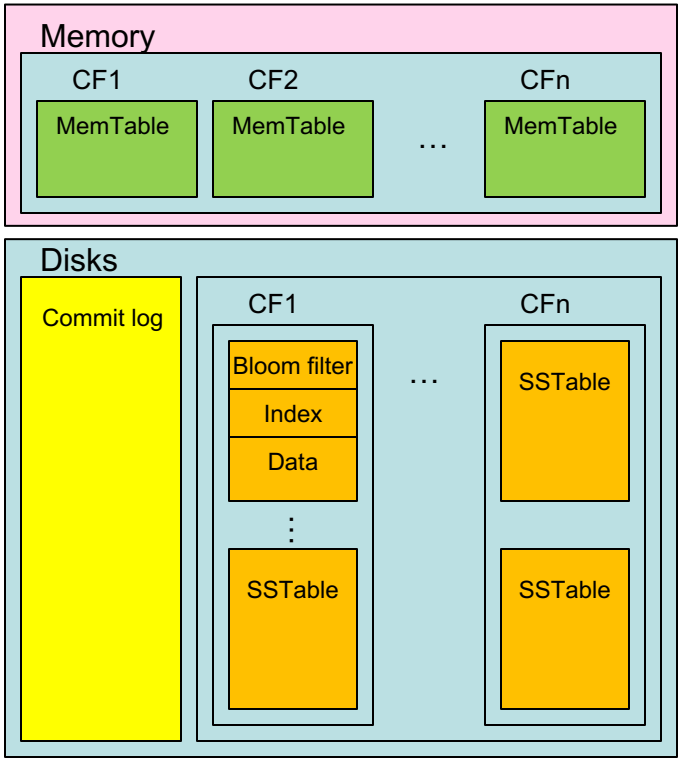
Query Model : Secondary Index

- Secondary Index
  - Give me all rows where value for column ‘2’ is ‘12’

	‘1’	‘2’	‘3’	‘4’	‘5’
‘a’	8	9	10		11
‘b’		12	13		14
‘c’	15			16	17
‘d’		18			
‘e’	19	20			20
‘f’	22	23	24	25	26

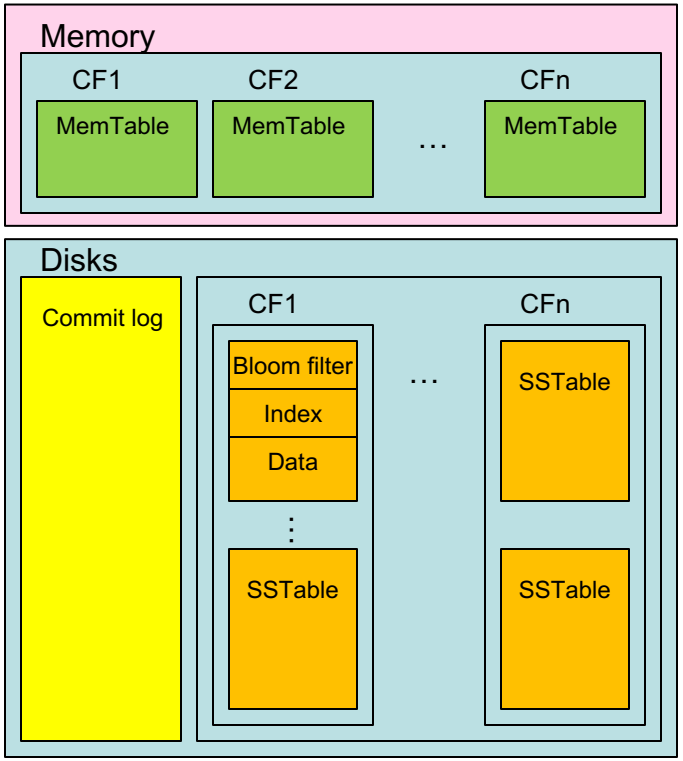
Write path

1. Write to commit log
2. Update MemTable
3. Acknowledge the client
4. When MemTable reaches a threshold, flush to disk as SSTable



Read path

- Versions of the same column can be spread at the same time
  - In the MemTable
  - In the MemTable being flushed
  - In one or multiple SSTable
- All versions read, and resolved / merged using timestamp
  - Keys and Rows cache
  - Bloom filters allow to skip reading unnecessary SSTables
  - SSTables are indexed
  - Compaction keep things reasonable



Uncovered Advanced Features

- Lightweight transactions
- Materialized view
- Secondary Index
- Seamless integration with other computation framework, such as Spark
- Bulk Loading
- Compression
- Compaction
  - Size-tiered vs. Leveled vs. Date-tiered compaction
- Multi tenancy
- Data center awareness

Use case: Time Series

```
CREATE TABLE counterT (
id text,
ts timestamp,
c counter,
PRIMARY KEY (id, ts)) WITH CLUSTERING ORDER BY (ts ASC);
```

- Example

```
counterT['sensor1'][2019-06-14 18:30:00]
counterT['sensor1'][2019-06-14 18:30:05]
counterT['sensor1'][2019-06-14 18:30:10]
…
```

Query per entity
number of hits for ‘sensor1’
between 18:30:00 and 19:00:00

```
counterT[2019-06-14 18:30:00]['sensor1']
counterT[2019-06-14 18:30:00]['sensor2']
counterT[2019-06-14 18:30:00]['sensor3']
…
counterT[2019-06-14 18:30:05]['sensor1']
```

Query per date range
all entities being hit between
18:30:00 and 19:00:00
! need complete date enumeration

Conclusion

- Cassandra is **not** a general purpose solution
- But Cassandra is doing a **really good job** if used accordingly
  - Really good scalability
    - Netflix's 1M w/s on AWS
    - Low operational cost
    - Admin friendly, no SPoF, Vnodes, snapshot, …
  - *Advanced* data and query model

Installation

- Installa Cassandra
  - On MacOS: brew install Cassandra
  - Start the client: `cqlsh`

- More info:
  - http://cassandra.apache.org