CS 649 Big Data: Tools and Methods
Spring Semester, 2022

Doc 22 NoSQL, Cassandra
Apr 14, 2022

CAP Theorem

CAP theorem says in a distributed system you can not have all three

- Consistency
- Availability
- Tolerance to network Partitions

Consistency

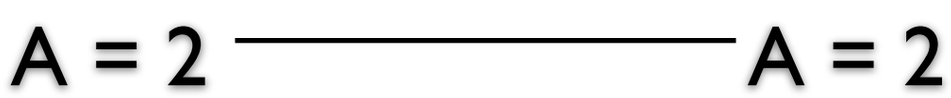
Machine 1 Machine 2

A = 2 —————————— A = 2

Not Consistent A = 2 A = 3

Partition

Machine 1 Machine 2



Partitioned

Machine 1 cannot

talk to machine 2

But how does machine 1 tell the difference between no connection and a very slow connection or busy machine 2?

Latency

Latency

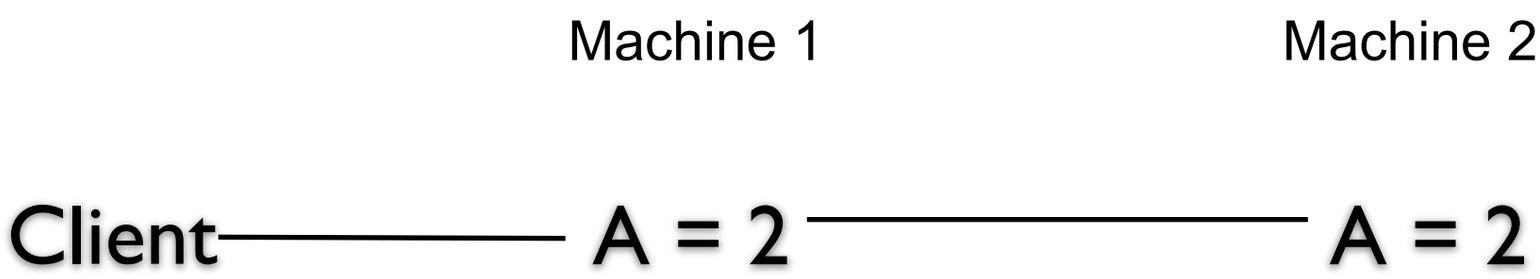
Time between making a request and getting a response

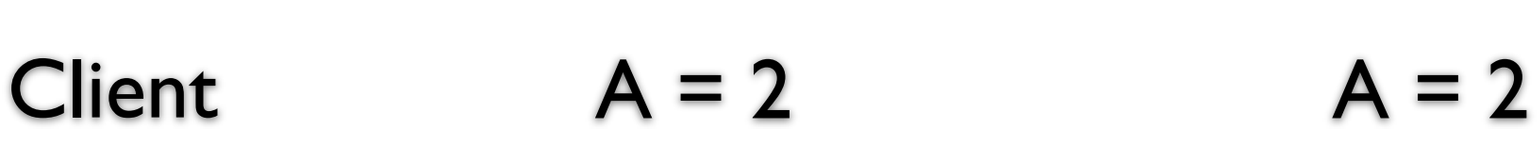
Distributed systems always have latency

In practice detect a partition by latency

When no response in a given time frame assume we are partitioned

Available





Client can not access value of A

What does not available mean?

- No connection
- Slow connection
- What is the difference?

Some say high available - meaning low latency

In practice available and latency are related

Consistency over Latency

Machine 1 Machine 2

Set A to 3 A = 2 A = 2

Set A to 3 A = 2 Lock A A = 2

Write requests queued until unlocked

Set A to 3 A = 2 Set A to 3 A = 2

Increased latency
System still available

Set A to 3 A = 3 Unlock A A = 3

A = 3 A = 3

Latency over Consistency

Machine 1

Machine 2

Set A to 3

A = 2

A = 2

A = 3

A = 2

Write requests accepted

Set A to 3

Low latency

A = 3

A = 2

System inconsistent

A = 3

A = 3

Latency over Consistency - Write Conflicts

Machine 1

Machine 2

Set A to 3

A = 2

A = 2

Subtract 1 from A

A = 3

A = 2

Set A to 3

A = 3

A = 1

Subtract 1 from A

A = ?

A = ?

Need policy to make system consistent

Partition

Machine 1

Machine 2

A = 2

A = 2

A = 2

A = 2

Set A to 3

A = 3

Subtract 1 from A

A = 1

A = ?

A = ?

Need policy to make system consistent

CAP Theorem

Not a theorem

Too simplistic
- What is availability
- What is a partition of the network

Misleading

Intent of CAP was to focus designers attention on the tradeoffs in distributed systems

- How to handle partitions in the network
- Consistency
- Latency
- Availability

CAP & S3

S3 favors latency over consistency

Big Data 3-5 V’s

Volume
  Large datasets

Clusters - Spark

Velocity
  Real time or near-real time streams of data

Kafka

Variety
  Different formats
  Structured, Numeric, Unstructured, images, email, etc.

NoSQL
Cassandra

Variability
  Data flows can be inconsistent

Veracity
  Accuracy

Complexity

Databases

Relational database

1970 E.F. Codd introduce relational databases - 12 relational rules

1979 First Commercial Relational Database - Oracle

firstname	lastname	phone	code
John	Smith	555-9876	2000
Ben	Oker	555-1212	9500
Mary	Jones	555-3412	9900

NoSQL Databases

1965 MultiValue from TRW

1998 NoSQL coined by Carlo Strozzi

2003 Memcached

2005 CouchDB

2006 Google's BigTable paper

2007 MongoDB - Document based

2008 Facebook open sources Cassandra

2009

- Eric Evans popularized NoSQL, gave current meaning
- Redis - Key-value store
- HBase - BigTable clone for Hadoop

Why NoSQL Databases

Scaling to clusters

More flexible

- Easier to deal with change in data's structure

Less mismatch between objects and database data

Faster on some operations

Disadvantages of NoSQL

No SQL

No ad hoc joins

May not support ACID (Atomicity, Consistency, Isolation, Durability)

Consistency is relaxed

Some Common Types

Wide Column

Document

Key-Value

Graph

Examples

Type	Examples
Key-Value Cache	Apache Ignite, Couchbase, Coherence, Memcached, Redis, Velocity
Key-Value Store	ArangoDB, Aerospike, Couchbase, Redis
Key-Value Store (Eventually-Consistent)	Oracle NoSQL Database, Dynamo, Riak, Voldemort
Key-Value Store (Ordered)	FoundationDB, InfinityDB, LMDB, MemcacheDB
Tuple Store	Apache River, GigaSpaces
Object Database	Objectivity/DB, Perst, ZopeDB
Document Store	ArangoDB, Clusterpoint, Couchbase, CouchDB, DocumentDB, MarkLogic, MongoDB, Elasticsearch
Wide Column Store	Amazon DynamoDB, Bigtable, Cassandra, HBase

Source: https://en.wikipedia.org/wiki/NoSQL

Document-Oriented

At a key store a document

Key
    Identifier, URI, path

Document
    JSON, XML, YAML

```
{
    "FirstName": "Bob",
    "Address": "5 Oak St.",
    "Hobby": "sailing"
}
```

Key-Value Store

Document-Oriented database are special case of Key-Value store

- Value is a document

Key-Value store

- Different databases support different values

Redis Values can be

- Strings
- Sets of strings
- Lists of strings
- hash tables where table keys & values are strings
- HyperLogLogs
- Geospatial data

```
>>> import redis
>>> r = redis.Redis(host='localhost', port=6379, db=0)
>>> r.set('foo', 'bar')
True
>>> r.get('foo')
'bar'
```

Apache Cassandra

Open source

Distributed

Decentralized

Elastically scalable

Highly available

Fault-tolerant

Tuneably consistent

Row-oriented database (Wide Column)

Distribution design based on amazon’s dynamo

Data model based on Google’s BigTable

Cassandra Users

Apple

100,000 Cassandra nodes

Netflix

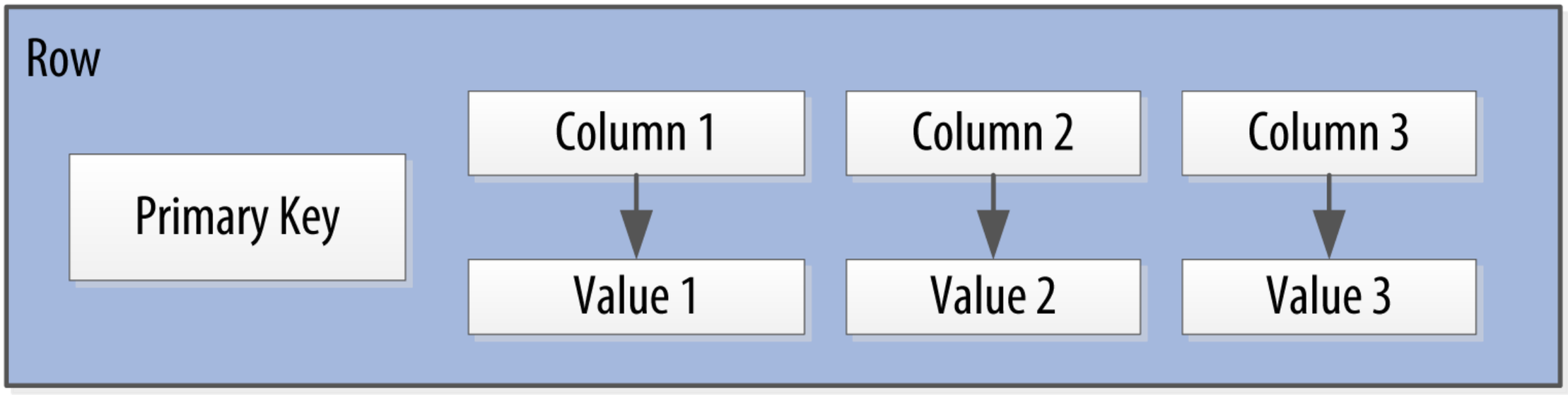
Cassandra as their back-end database for their streaming services

Uber

Million writes per second (2016)

Driver & Rider app update location every 30 seconds

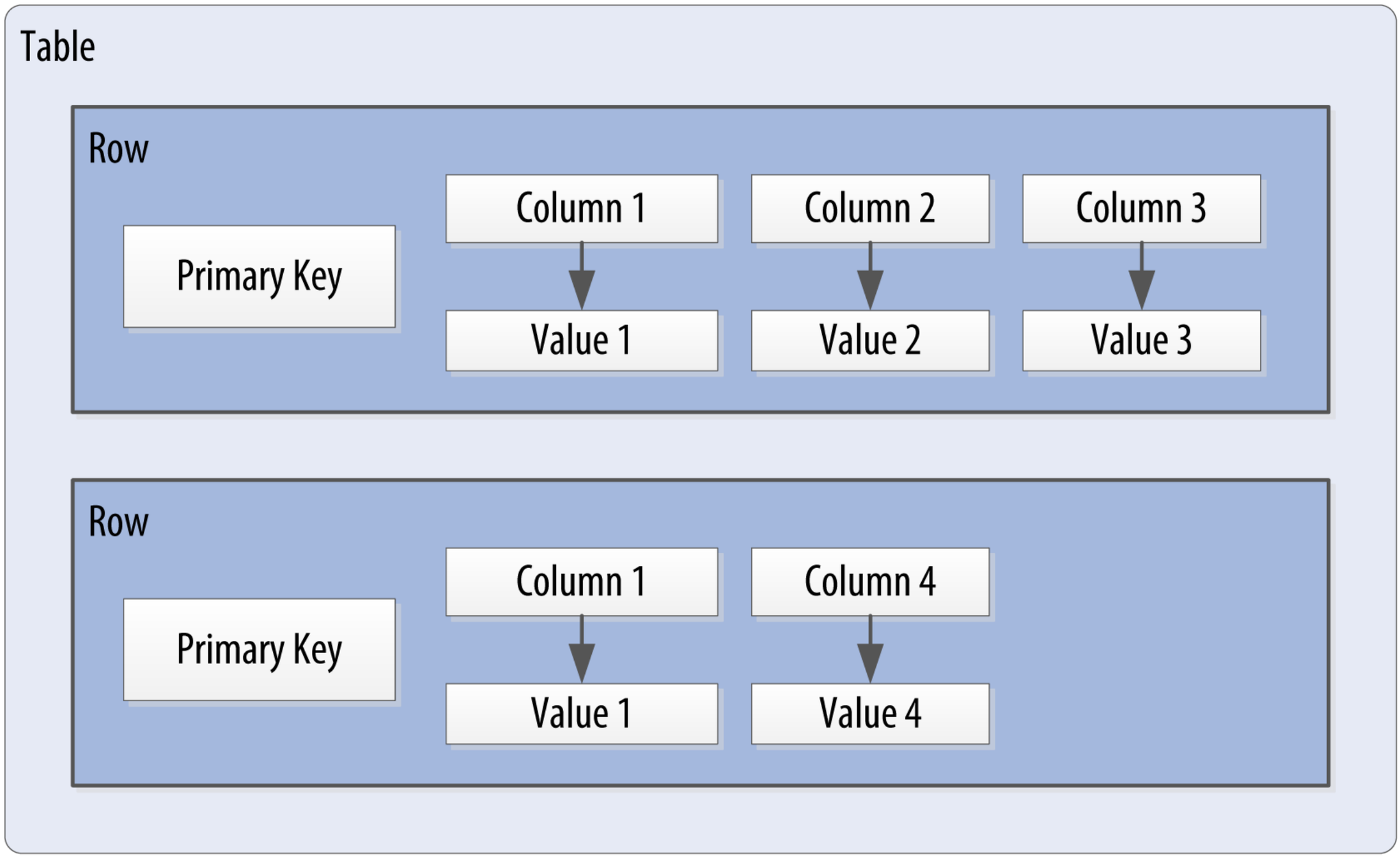
Cassandra Rows & Columns



Source for Diagrams

Cassandra: The Definitive Guide, 2nd Ed (3rd ed is coming)
Carpenter, Hewitt
O'Reilly Media, Inc, 2016

Cassandra Table



Wide Rows

Row with many columns

- Thousands or millions

Partition Key

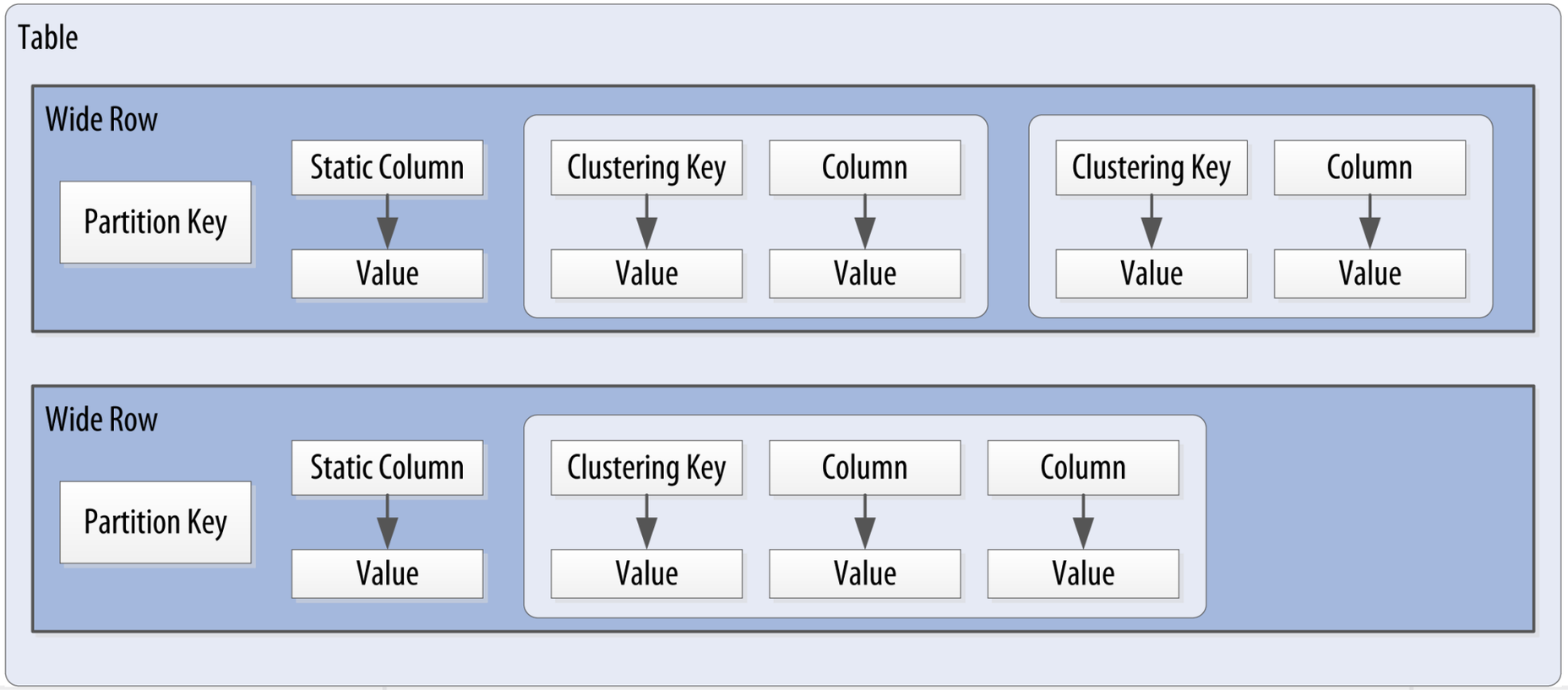
- Determines which node(s) used to store the row

Clustering Columns

- How data is sorted

Primary Key

- Partition Key +
- Clustering columns



Terms

Column	name/value pair
Row	Container for columns reference by primary key
Table	Container for Rows
Keyspace	Container for Tables
Cluster (Ring)	Container for Keyspaces

Keyspace, table, column names are stored as lowercase

Cassandra Clients

cqlsh

- Command line client
- Part of Cassandra distribution

DataStax Cassandra clients

- C++
- C#
- Java
- Node.js
- Python

DataStax

- Sells proprietary version of Cassandra

Sample Interaction

->cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.6 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.

cqlsh> CREATE KEYSPACE class_demo WITH
replication = {'class': 'SimpleStrategy', 'replication_factor': 1};

cqlsh> USE class_demo;

cqlsh:class_demo> DESCRIBE KEYSPACE class_demo;

CREATE KEYSPACE class_demo WITH
replication = {'class': 'SimpleStrategy', 'replication_factor': '1'}
AND durable_writes = true;

cqlsh:class_demo> DESCRIBE TABLE USER;

```
CREATE TABLE class_demo.user (
    first_name text PRIMARY KEY,
    last_name text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class':
'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';
```

```
cqlsh:class_demo> INSERT INTO user (first_name, last_name) VALUES ('Roger', 'Whitney');

cqlsh:class_demo> SELECT COUNT (*) FROM user;

 count
-------
     1

(1 rows)

Warnings :
Aggregation query used without partition key
```

cqlsh:class_demo> select * from user where first_name = 'Roger';

```
 first_name | last_name
------------+-----------
      Roger |   Whitney

(1 rows)
```

cqlsh:class_demo> select * from user where last_name = 'Whitney';

InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"

cqlsh:class_demo> select * from user where last_name = 'Whitney' ALLOW FILTERING ;

```
 first_name | last_name
------------+------------
      Roger |   Whitney

(1 rows)
```

```
cqlsh:class_demo> ALTER TABLE user ADD title text;

cqlsh:class_demo> INSERT INTO USER (first_name, last_name, title)
                  VALUES ('Wei', 'Wang', 'Dr');

cqlsh:class_demo> SELECT * FROM user;

 first_name | last_name | title
------------+-----------+-------
        Wei |      Wang |    Dr
      Roger |   Whitney |  null

(2 rows)
```

Timestamps

Each write to a column produces a timestamp for that value

Timestamps are used to resolve write conflicts

cqlsh:class_demo> select first_name, last_name, writetime(last_name) FROM user;

```
 first_name | last_name | writetime(last_name)
------------+-----------+----------------------
        Wei |      Wang |     1586738550697351
      Roger |   Whitney |     1586738178218687

(2 rows)
```

Time to Live (TTL)

Each column value can have a time to live value

```
cqlsh:class_demo> INSERT INTO USER (first_name, last_name, title)
                  VALUES ('Bill', 'Temp', 'Lecturer') using TTL 600;

cqlsh:class_demo> SELECT first_name, TTL(last_name), TTL(title)
                  FROM user WHERE first_name = 'Bill';

 first_name | ttl(last_name) | ttl(title)
------------+----------------+------------
       Bill |            502 |        502

(1 rows)
```

Time to Live (TTL)

cqlsh:class_demo> **SELECT * FROM user;**

```
 first_name | last_name | title
------------+-----------+----------
        Wei |      Wang |        Dr
      Roger |   Whitney |      null
       Bill |      Temp | Lecturer
```

10 minutes later

cqlsh:class_demo> **SELECT * FROM user;**

```
 first_name | last_name | title
------------+-----------+-------
        Wei |      Wang |    Dr
      Roger |   Whitney |  null
```

Data Types

Numeric

tinyint 8-bit
smallint 16-bit
int 32-bit
bigit 64-bit
varint (java.math.BigInteger)
float 32-bit
double 64-bit
decimal

Text

text, varchar - UTF-8
ascii

Time

timestamp
date, time
uuid
timeuuid

Other

boolean
blob
int
counter

Collections

set
list
map

Set

```
cqlsh:class_demo> ALTER TABLE user ADD emails set<text>;

cqlsh:class_demo> UPDATE user SET emails = { 'rwhitney@sdsu.edu'}
                                WHERE first_name = 'Roger';

cqlsh:class_demo> UPDATE user SET emails = emails + { 'whitney@sdsu.edu'}
                                WHERE first_name = 'Roger';

cqlsh:class_demo> SELECT emails from user WHERE first_name = 'Roger';

 emails
------------------------------------------
 {'rwhitney@sdsu.edu', 'whitney@sdsu.edu'}

(1 rows)
```

List

```
cqlsh:class_demo> ALTER TABLE user ADD phone_numbers list<text>;

cqlsh:class_demo> UPDATE user SET phone_numbers = [ '619-594-3535']
                                WHERE first_name = 'Roger';

cqlsh:class_demo> SELECT * FROM user WHERE first_name = 'Roger';

 first_name | emails                                | last_name | phone_numbers     | title
------------+---------------------------------------+-----------+-------------------+-------
      Roger | {'rwhitney@sdsu.edu',
          'whitney@sdsu.edu'} |   Whitney | ['619-594-3535'] |  null

(1 rows)
```

Why Collections?

Relational Model - Uses separate tables

Where are the tables physically located?

user

id	first_name	last_name
1	Roger	Whitney
2	Wei	Wang

email_address

id	email	user_id
1	whitney@sdsu.edu	1
2	rwhitney@sdsu.edu	1
3	wwang@sdsu.edu	2

Why Collections?

In Cassandra we denormalize the data

Keep related data together for faster access

```
 first_name | emails                    | last_name | phone_numbers    | title
------------+---------------------------+-----------+------------------+-------
     Roger | {'rwhitney@sdsu.edu',
           'whitney@sdsu.edu'} |   Whitney | ['619-594-3535'] |  null
```

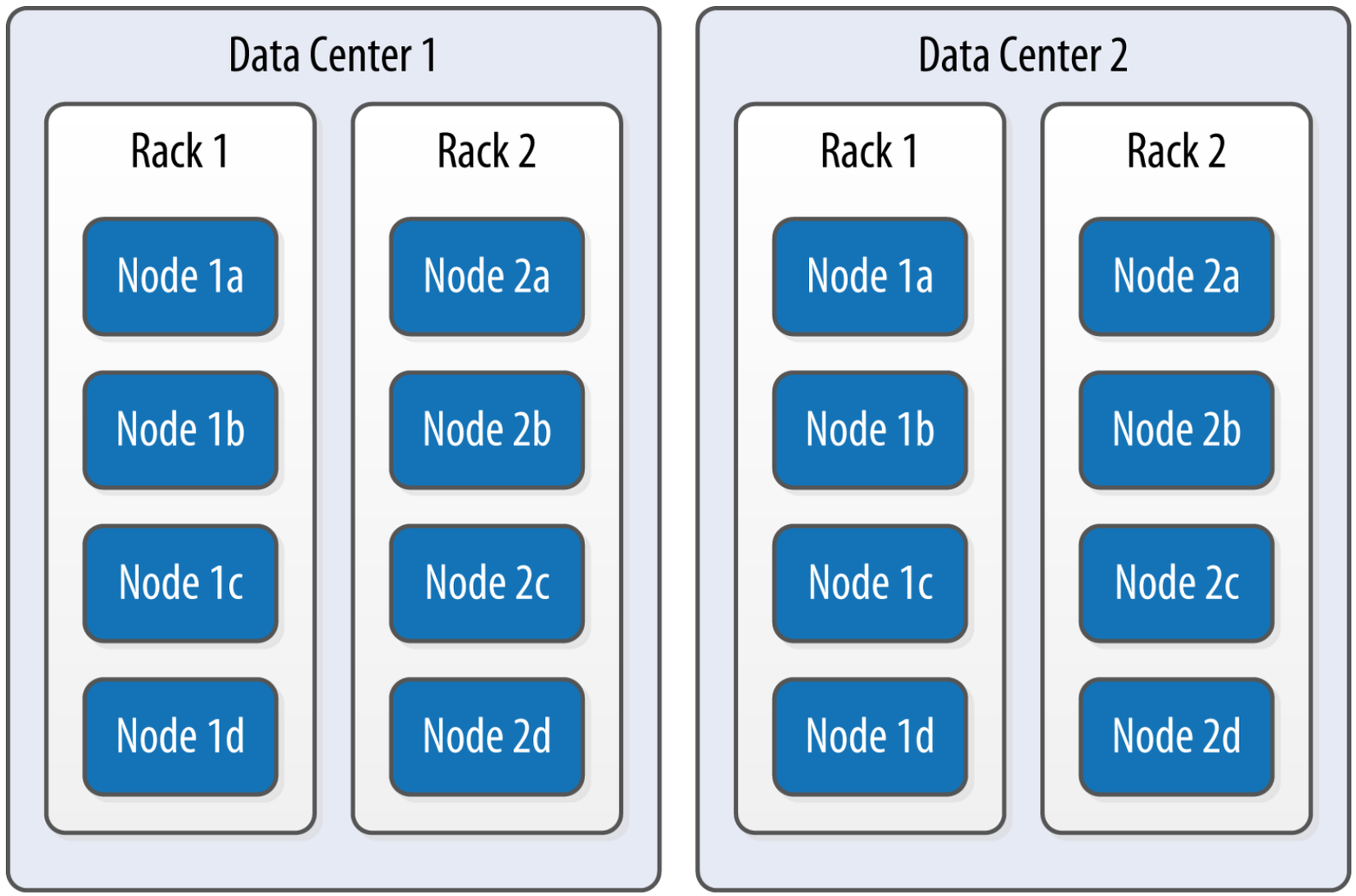
Cassandra Architecture

Two levels of grouping

- Rack
- Data Center

Default Configuration

- 1 data center
- 1 rack



Gossip & Snitches

Gossip protocol

- Determines likelihood that machine is dead

Snitches

- Determines network topology
- Which machines are close to each other

Rings & Tokens

Each Cassandra node is given a token - 64 bit integer

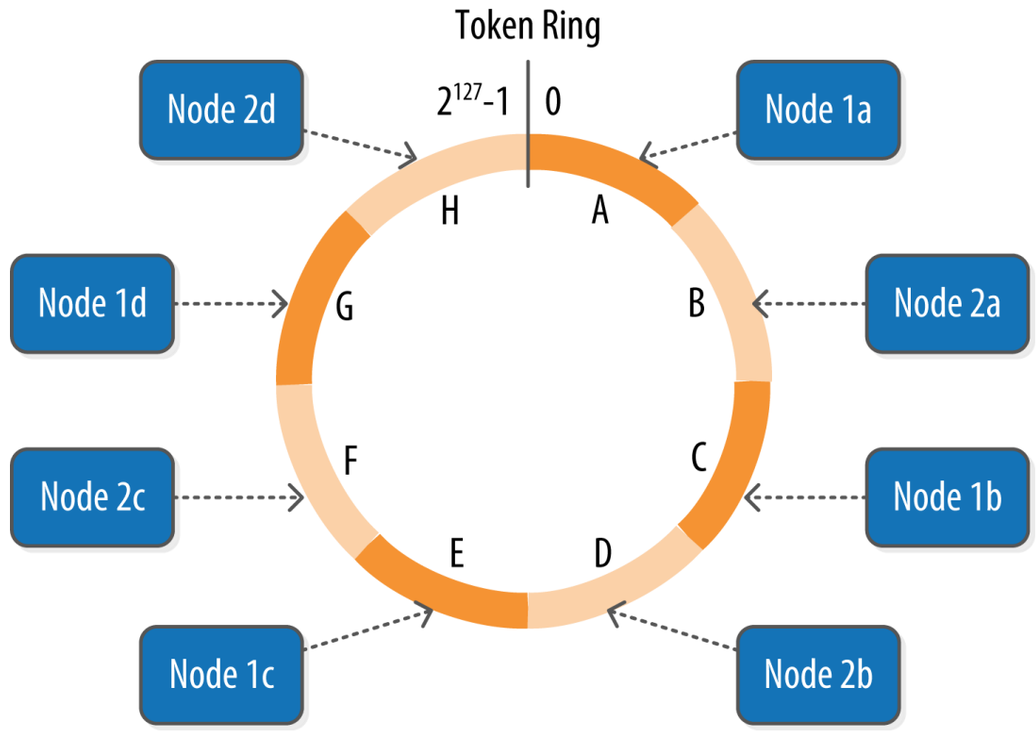
Nodes are arranged in a ring

Node owns all values

- Greater than the token of the previous node
- Less than or equal to it token

Virtual Nodes

- Physical node is treated as many virtual nodes
- Default is 256
- Helps
  - Rebalance
  - Deal with heterogeneous machines



Partitioners

Each wide row has a partition key

The partition key is hashed to 64 bit number

The hash value determines the node the row is stored on

```
CREATE TABLE t (
   id int,
   v text,
   PRIMARY KEY (id)
);
```

id - partition key

```
CREATE TABLE t (
   id int,
   c text,
   v text,
   PRIMARY KEY (id, c)
);
```

id - partition key

c - clustering key

Sort data in partition

```
CREATE TABLE t (
   id1 int,
   id2 int,
   c1 text,
   c2 text
    v text,
   PRIMARY KEY ((id1,id2), c1,c2)
);
```

(id1,id2) - partition key

c1, c2 - clustering key

Sort data in partition

Replication Strategy

```
CREATE KEYSPACE class_demo WITH
        replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

SimpleStrategy

Place replicas in next N nodes

NetworkTopologyStrategy

Different replication factor per data center

Replicas in different racks

Consistency Levels

ONE
TWO
THREE

QUORUM
ALL

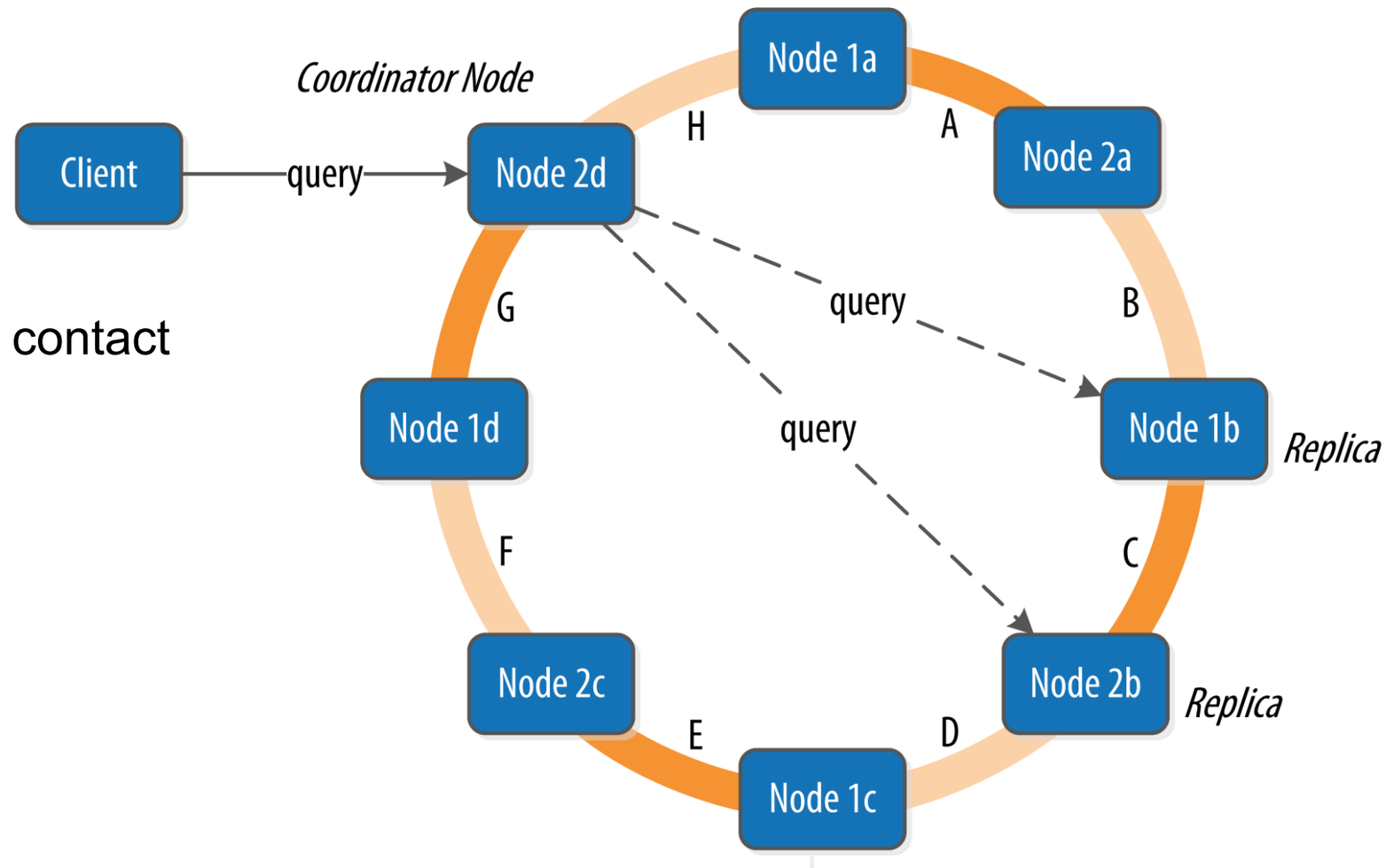
On write

How many nodes must confirm write

On read

How many nodes must return value

Client can contact any node



Issues on Write & Read

What if replica is not available when writing & then come back online?

CAP theorem - nodes will not have consistent values

How to handle reads when nodes return different values

Memtables, SSTables, and Commit Logs

On writes

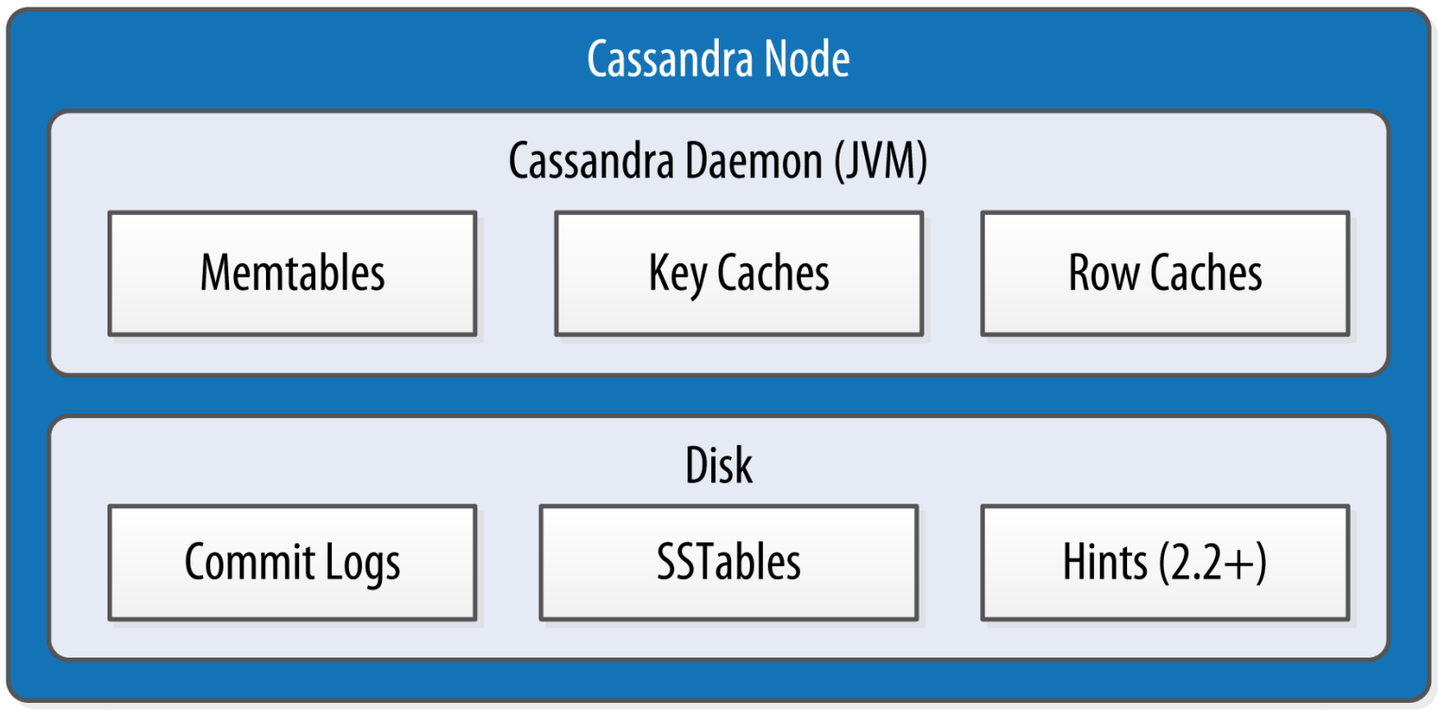
- Store command in log (Kafka)
- Add/modify data in Memtables

When memtables get too big

- Write to SSTable
- Clear memtable
- Mark command in log

On startup

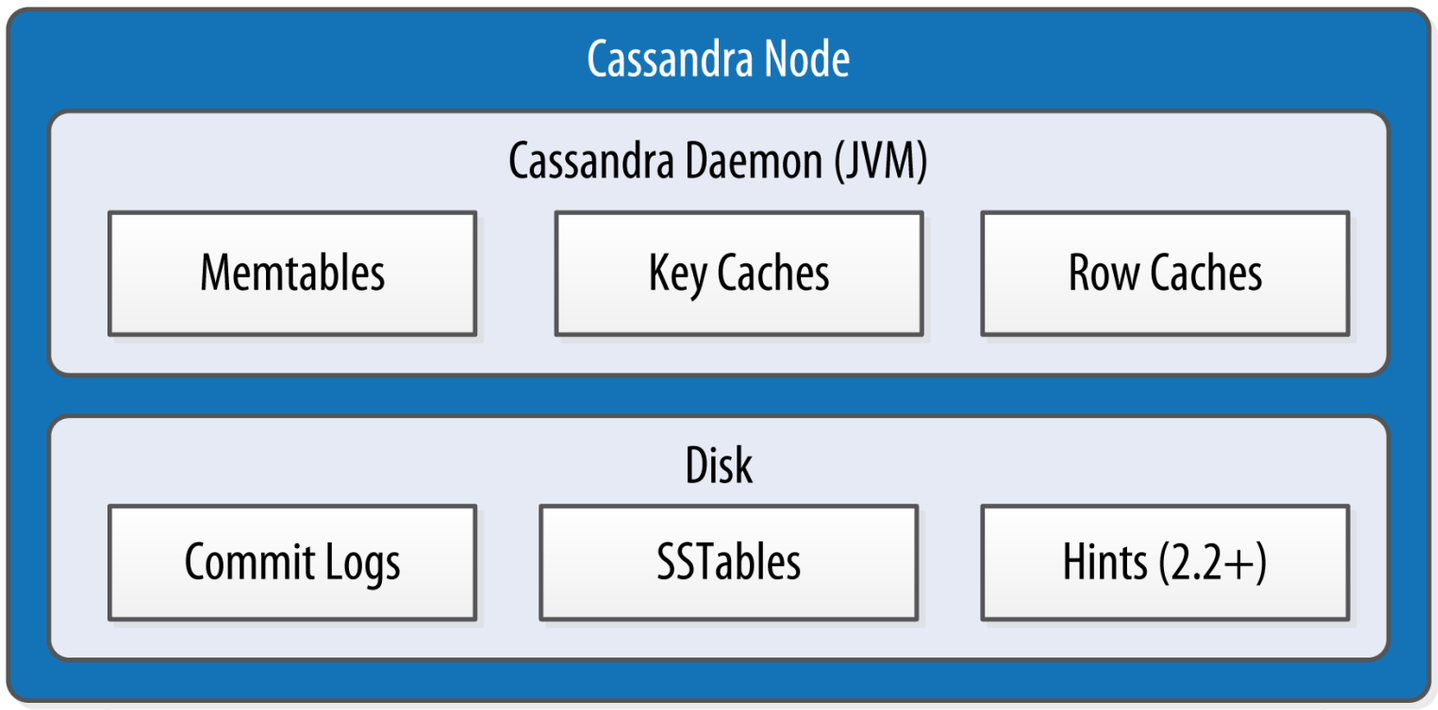
- Use logs to recreate meltable



Memtables, SSTables, and Commit Logs

On writes

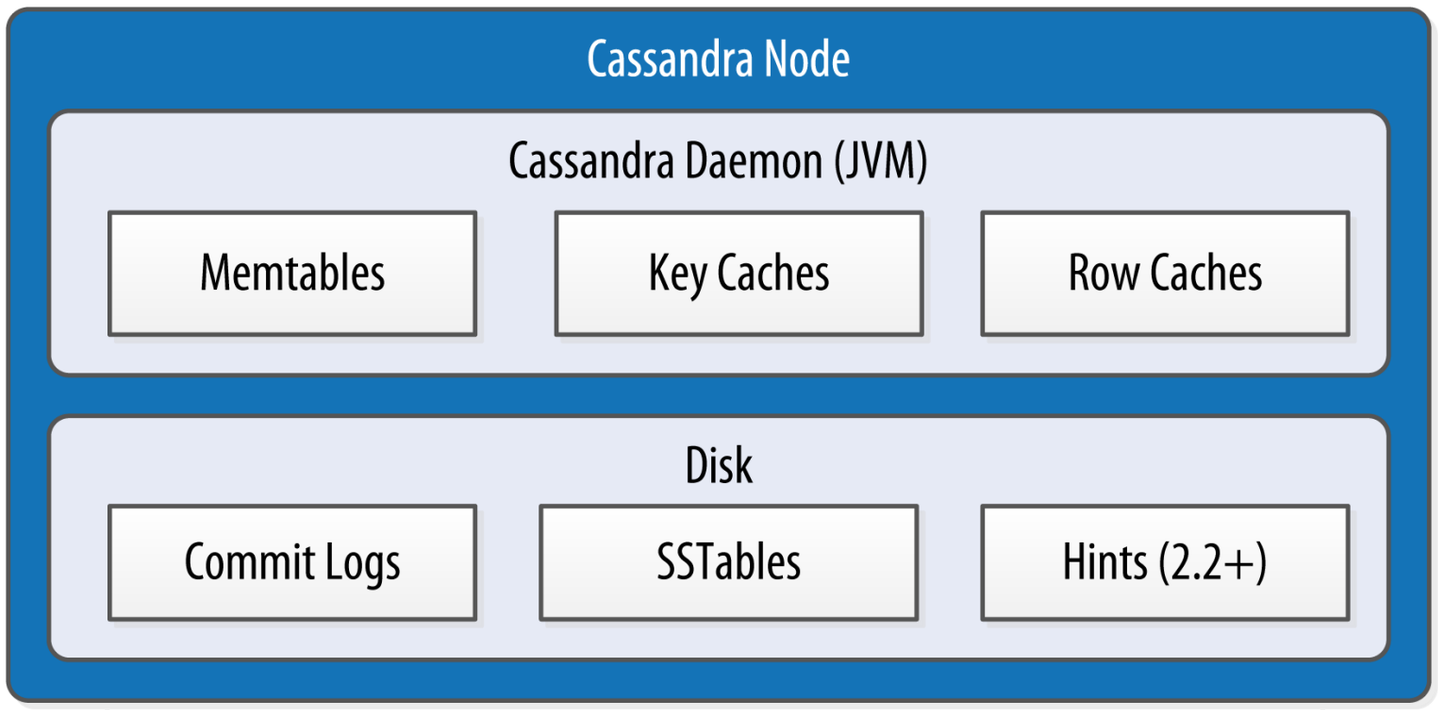
- If replica is not available
- Store command in Hints
- When replica becomes available send update



Read repair

On reads

- Contact required number of replicas
- Detect out of date values
- If needed update replicas before continuing
- Otherwise update out of date values after finish read



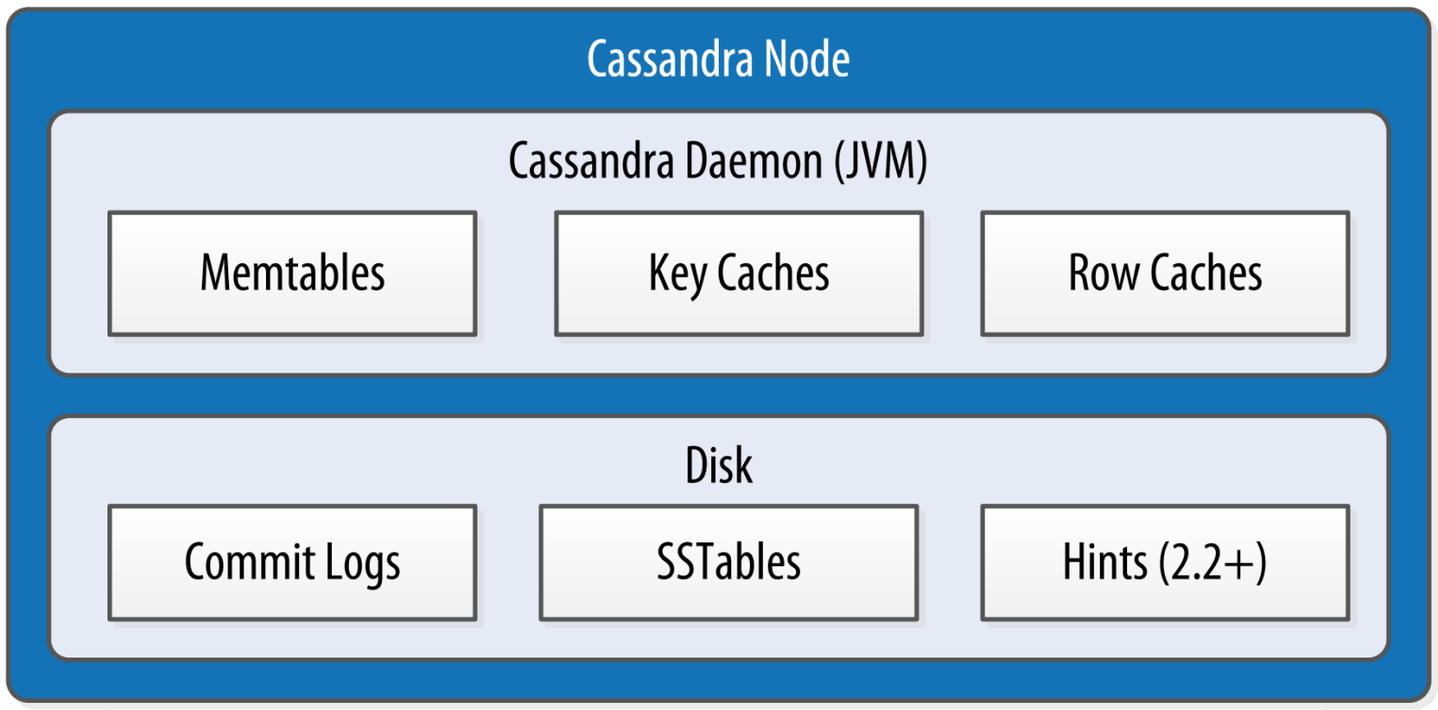
Read repair

On reads

Check memtable - if has value return it

Use Bloom filter to help determine which SSTable contains value

Other repair algorithm used: anti-entropy



Replication Factor Vs Consistency Level

Replication Factor

How many node have copy of the data (row)

Set when creating keyspace (database)

Consistency Level

How many replicas need to respond to a request for it to return result

Set when

Client connects to Cassandra

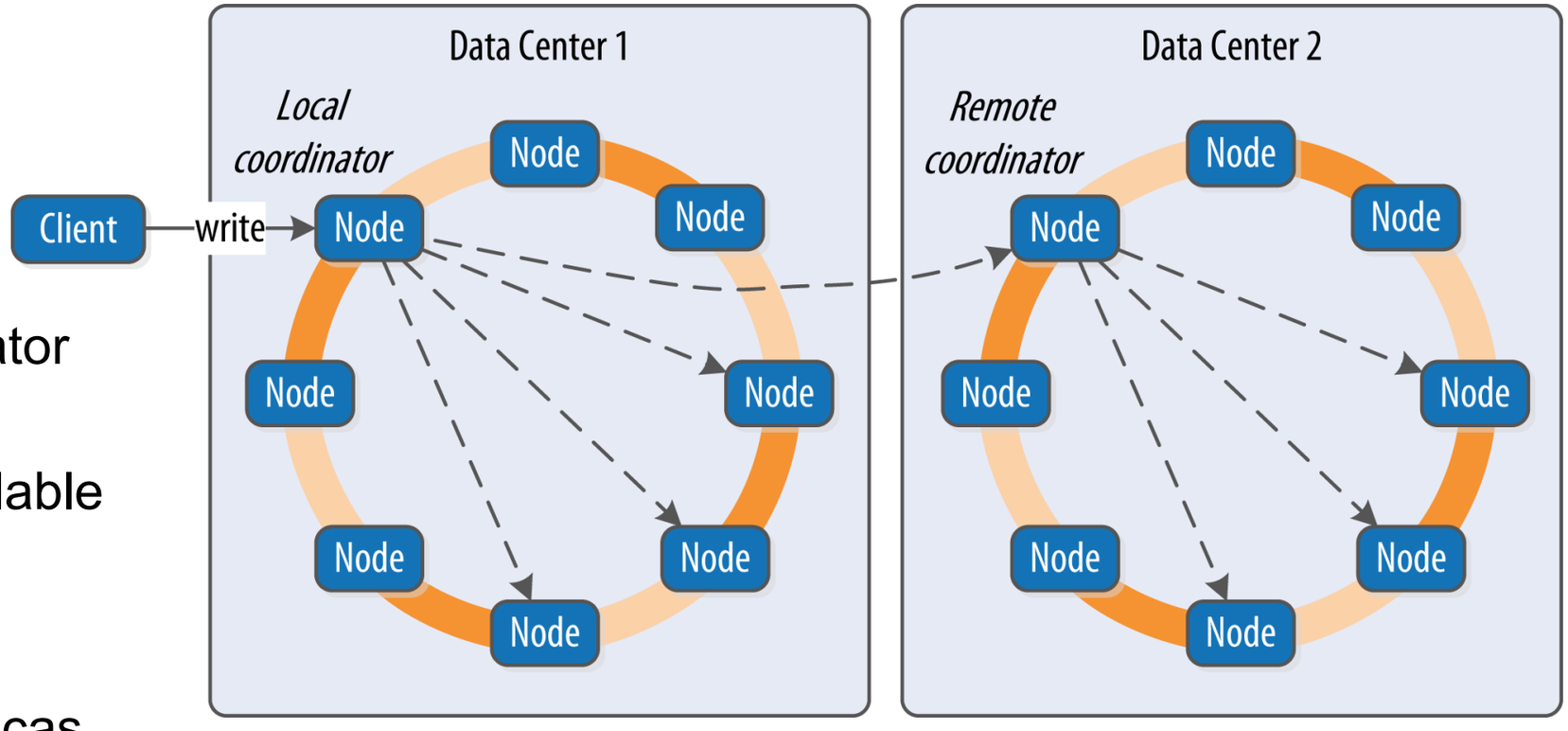
Per request

Write Details

Client contact a node
Which becomes local coordinator

If not enough replicas are available
return fail

Send write requests to all replicas

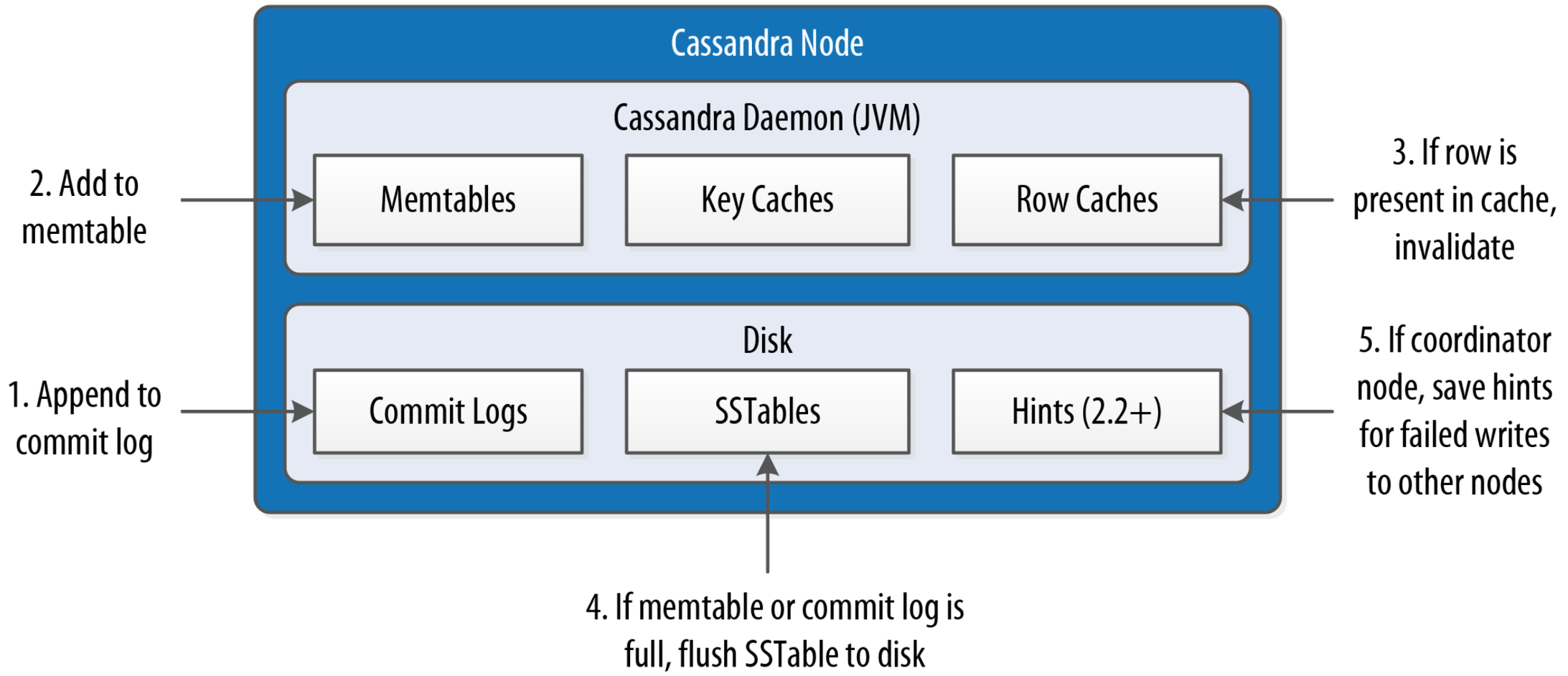


If spans data center local coordinator contact remote coordinators

Once a sufficient number of relics respond coordinator acknowledges write to client

Any replica that does not respond in timeout period
- Mark it as down
- Store hint for write with replica returns

Write Details in One Node



Kafka Like Strategy

Uses log files to store requests

Keep data in memory for fast access

Excess goes to disk (SSTable)

More Write Details - Lightweight Transactions

IF NOT EXISTS

Returns true if does not exist

```
cqlsh> INSERT INTO hotel.hotels (id, name, phone) VALUES (
 'AZ123', 'Super Hotel at WestWorld', '1-888-999-9999') IF NOT EXISTS;

 [applied]
-----------
      True
```

If exists return false and the values we tried to enter

```
cqlsh> INSERT INTO hotel.hotels (id, name, phone) VALUES (
 'AZ123', 'Super Hotel at WestWorld', '1-888-999-9999') IF NOT EXISTS;

 [applied] | id    | address | name                        | phone
-----------+-------+---------+-----------------------------+------------
     False | AZ123 |    null | Super Hotel at WestWorld | 1-888-999-9
```

DataStax Python Driver

https://docs.datastax.com/en/developer/python-driver/3.23/

```
from cassandra.cluster import Cluster

cluster = Cluster()
session = cluster.connect('class_demo')

rows = session.execute('select * from user')
rows
```

Run in Jupyter Notebook

```
<cassandra.cluster.ResultSet at 0x11b408bd0>
```

Result sets can be read once

```
rows = session.execute('select * from user')
for user_row in rows:
    print(user_row)
```

```
Row(first_name='Wei', emails=None, last_name='Wang', phone_numbers=None, title='Dr')
Row(first_name='Roger', emails=SortedSet(['rwhitney@sdsu.edu', 'whitney@sdsu.edu']),
last_name='Whitney', phone_numbers=['619-594-3535'], title=None)
```

```
rows = session.execute('select * from user')
for user_row in rows:
    print(user_row.first_name, user_row.last_name, user_row.phone_numbers)
```

```
Wei Wang None
Roger Whitney ['619-594-3535']
```

```
rows = session.execute('select * from user')
for user_row in rows:
    print(user_row[0], user_row[1], user_row[2])
```

```
Wei None Wang
Roger SortedSet(['rwhitney@sdsu.edu', 'whitney@sdsu.edu']) Whitney
```

Simple Statement

Allows setting information about the request

```
from cassandra.query import SimpleStatement
statement = SimpleStatement("SELECT * FROM user", fetch_size=10)
for user_row in session.execute(statement):
    print(user_row.first_name)
```

```
from cassandra.query import SimpleStatement
statement = SimpleStatement("SELECT * FROM user", consistency_level = 1)
for user_row in session.execute(statement):
    print(user_row.first_name)
```

Only had one node so could only set level to 1

Prepared Statements

Cassandra parses and saves prepared statement

Client sends only values of parameter to bind

Reduces

- Network traffic
- CPU usage on Cassandra - only parse once

```
user_lookup_stmt = session.prepare("SELECT * FROM user WHERE user_id=?")

users = []
for user_id in [1,21, 39, 99]:
    user = session.execute(user_lookup_stmt, [user_id])
    users.append(user)
```

Data Modeling

SQL

- Normalize tables to avoid duplicating data
- Then make queries to get the information you want

Cassandra

- First determine your queries - what questions you what answered
- Then create table that contain data for a given query
- Data will be duplicated

Batches

Combine multiple modifications in one statement

Atomic

Only INSERT, UPDATE, DELETE

```
cqlsh> BEGIN BATCH
  INSERT INTO hotel.hotels (id, name, phone)
    VALUES ('AZ123', 'Super Hotel at WestWorld', '1-888-999-9999');
 INSERT INTO hotel.hotels_by_poi (poi_name, id, name, phone)
    VALUES ('West World', 'AZ123', 'Super Hotel at WestWorld',
    '1-888-999-9999');
APPLY BATCH;
```