# A column-family data store

# Cassandra

- Used by more than 1500 companies
- Developed at Facebook
  - Initial release: 2008 (stable release: 2013)
  - now, Apache Software License
- Written in: Java
- Operating System: cross-platform
- Java, Scala, Ruby, C#, Python, Perl, PHP, C++ Cassandra clients

# Who uses Cassandra?

- Some of the largest production deployments:
  - Apple: over 75,000 nodes storing over 10 PB of data
  - Netflix: 500 nodes, 420 TB, over 1 trillion requests per day
  - eBay: over 100 nodes, 250 TB

# Who *else* uses Cassandra?

# Data model and interaction

# Cassandra data model

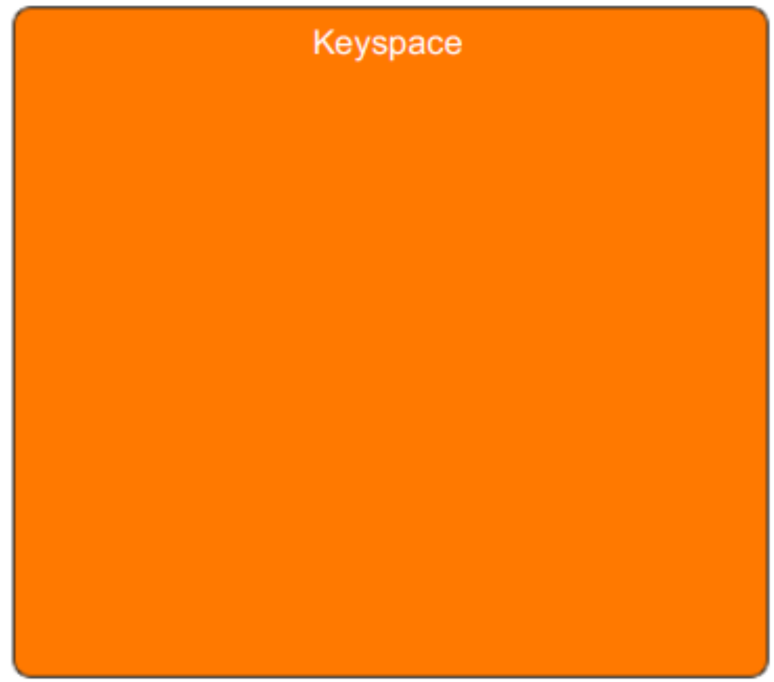
- The original Cassandra data model is similar to that discussed for generic column-family data stores, with a basic interaction protocol - Thrift
- More recently, a new high level data model (with primary keys) and related (SQL-like) language has been proposed
- We will discuss only the high level data model and language since the interaction protocol for the original data model is now deprecated

# Data model and interaction

- Concepts closer to a traditional relational database with tables, columns and rows
- Cassandra Query Language (CQL) – similar to SQL
  - Still the same data underneath, just abstracted away
  - reintroduction of schema so that you don't have to read code to understand the data model
  - CQL creates a common language so that details of your data can be easily communicated (but it is not a standard)
  - best-practice Cassandra interface that hides the details of the original data model
- CQL is one of the strongest reasons why Cassandra has become so widely used today

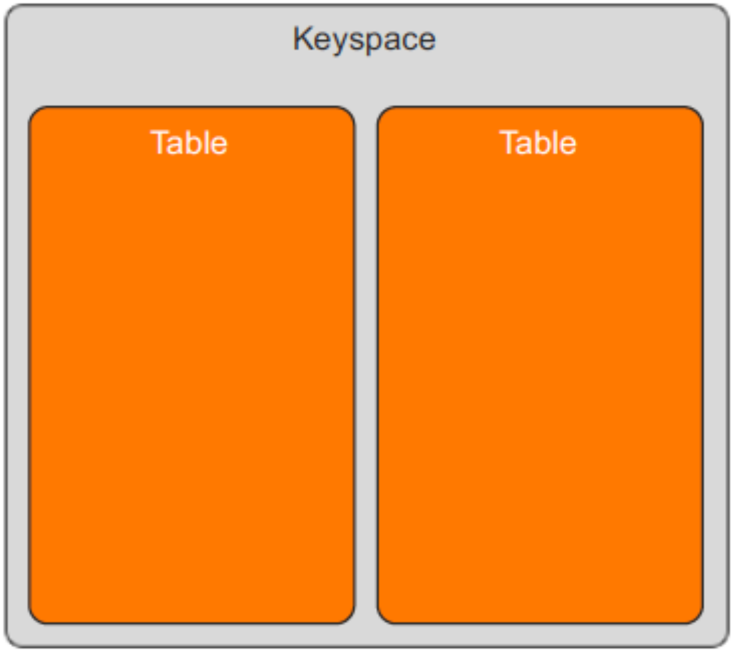
# Data model

- Keyspace – Database in an RDBMS



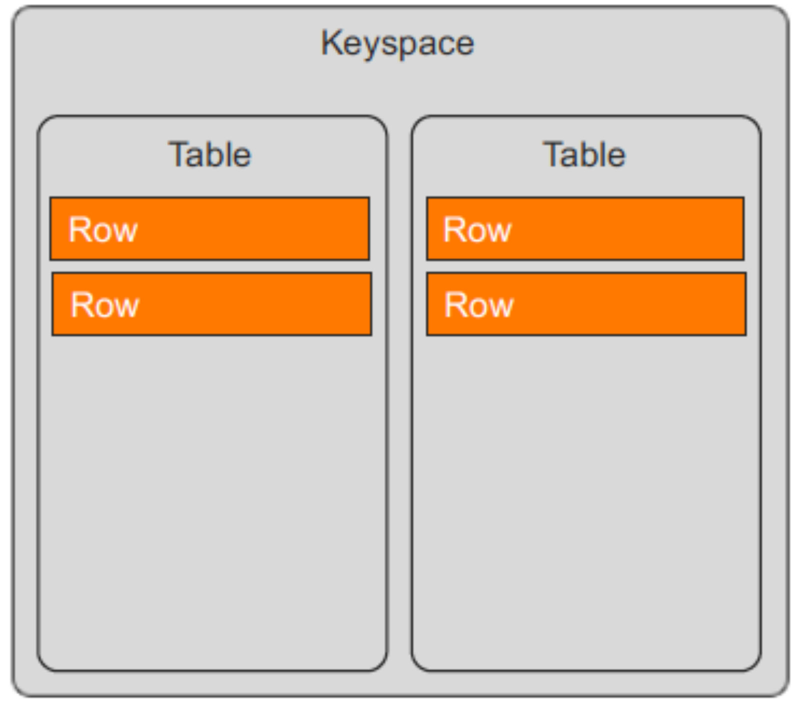
# Data model

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS



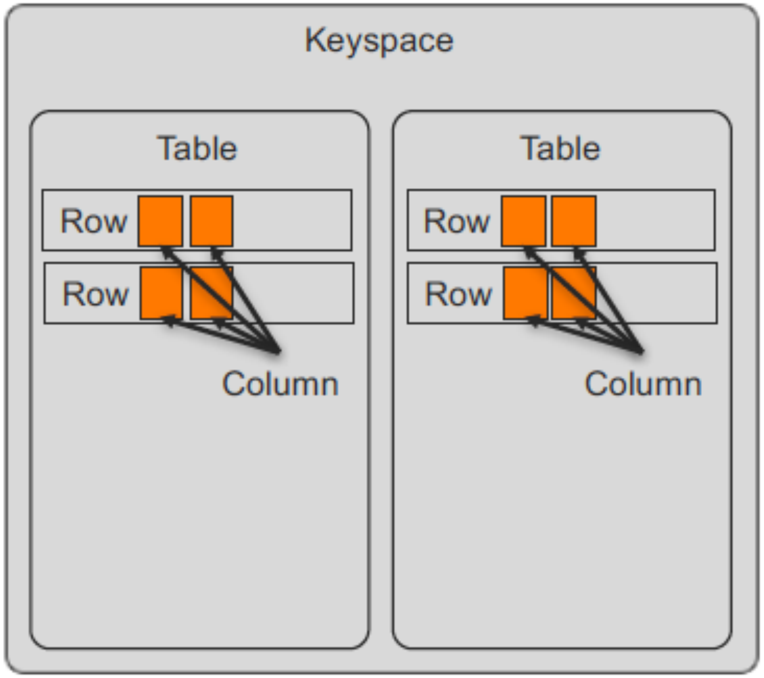
# Data model

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS
- Row – Row in an RDBMS



# Data model

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS
- Row – Row in an RDBMS
- Column – Column in an RDBMS



# CQL at the glance

```
CREATE TABLE clients (
   codCli int PRIMARY KEY,
   name text,
   surname text,
   birthdate date );
```

It looks like SQL but the original Cassandra model is still there, even if hidden by the CQL data model

```
INSERT INTO clients (codCli, name, surname, birthdate)
VALUES (3, 'john', 'black', '15/10/2000');

SELECT * FROM users;


   codCli| name | surname| birthdate
---------+-------+-------+-----------
       3 | john | black  | 15/10/2000
```

# CQL atomic data types

- Strings
  - Ascii, varchar, text
- Numbers
  - Bigint, decimal, double, float, int,…
- Time
  - Date, time, timestamp
- …

# CQL – CREATE TABLE and partition key

- Similar to SQL CREATE TABLE

```
CREATE TABLE clients (
    codCli int PRIMARY KEY,
    name text,
    surname text,
    birthdate date );
```

Partition key codCli
Primary key codCli

- By default, the partition key is the first column of a composite key (thus, partition key is not unique but do not forget the match with the old Cassandra model)

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, director));
```

Partition key title
Primary key (title, director)

Partition key (title, director)
Primary key (title, director)

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY ((title, director)));
```

# CQL – CREATE TABLE and partition key

Partition key =

first attribute of the primary key

or

a prefix of the primary key, specified inside ( )

# CQL – CREATE TABLE

- No foreign keys

```
CREATE TABLE movie (
   title text,
   director text,
   Year int,
   genre text,
   PRIMARY KEY ((title, director));

CREATE TABLE video (
   loc int PRIMARY KEY,
   type text,
   title text,
   director text);
```

There is no way to specify the foreign key

Side effect:
Table video can refer movies not included in table movie

# CQL - INSERT

- Syntax similar to SQL

```
INSERT INTO t2 (a, b, c, d) VALUES (0,0,0,0);
```

- No primary key violation: if you insert a row with the same primary key of another row in the table, the first will be simply updated
  - INSERT becomes an UPDATE!

# CQL – CREATE TABLE and partition key

- All the attributes that follow the partition key attributes in a primary key definition are called clustering columns

- In a node, all rows in a collection with the same partition key value are locally sorted with respect to clustering columns (default: ascending order)

- Consequences on how to query data

# Example

```
CREATE TABLE T
(       A int,
        B int,
        C int,
        D int,
        PRIMARY KY (A, B, C)
);

INSERT INTO T (A, B, C, D) VALUES (0,0,0,0);
INSERT INTO T (A, B, C, D) VALUES (0,1,2,2);
INSERT INTO T (A, B, C, D) VALUES (0,0,1,1);
INSERT INTO T (A, B, C, D) VALUES (0,1,3,3);
INSERT INTO T (A, B, C, D) VALUES (1,1,4,4);
```

Two partitions
(physical level)

**P1**

| A | B | C | D |
|---|---|---|---|
| 1 | 1 | 4 | 4 |

**P2**

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 2 | 2 |
| 0 | 1 | 3 | 3 |

# Example

Two partitions
(physical level)

P1

```
 A B C D
(1,1,4,4)
```

P2

```
 A B C D
(0,0,0,0)
(0,0,1,1)
(0,1,2,2)
(0,1,3,3)
```

If the partitions are physically stored on two nodes N1 and N2…

*Node N1*

```
A 1:4:D
1   4
```

*Node N2*

```
A 0:0:D 0:1:D 1:2:D. 1:3:D
0   0      1      2       3
```

If the partitions are physically stored on a single node N…

*Node N*

```
A 1:4:D
1   4
A 0:0:D 0:1:D 1:2:D. 1:3:D
0   0      1      2       3
```

# CQL - UPDATE

```
CREATE TABLE T
(       a int,
        b int,
        c int,
        d int,
        PRIMARY KEY (a, b, c)
);

INSERT INTO T (a, b, c, d) VALUES (0,0,0,0);
```

- UPDATE is similar to INSERT!

```
UPDATE T SET a = 0 AND b = 0 AND c = 0 AND d = 1;
```

- is equivalent to

```
INSERT INTO T (a, b, c, d) VALUES (0,0,0,1);
```

- It can contain a WHERE clause following the same rules of the WHERE clause in SELECT (see later)

# CQL - DELETE

- Marks data for removal from a table
- Data is removed later through a process called compaction
- Unlike SQL it MUST contain a WHERE clause, following the same rules than the WHERE clause in SELECT (see later)

```
DELETE FROM T
WHERE a = 0 AND b = 0 AND c = 0;
```

# CQL - SELECT

- Similar to SELECT statement in SQL

- Retrieve all the rows from a table

```
SELECT * FROM T;
```

- Project all the rows from a table

```
SELECT a FROM T;
```

- Select some rows from a table, with many limitations

```
SELECT * FROM T WHERE a = 0 AND b = 0 AND c = 1;
```

# CQL – SELECT, restrictions

- Many restrictions on what you can do in the WHERE clause
  - no join
  - restrictions on selection conditions

- Some queries cannot be executed

- Solutions
  - Query-based design
  - Indexes
  - Forcing the execution of non-admitted queries

# CQL – SELECT, restrictions: no join

- Queries must refer a single table
- Joins are not supported
    - As we know, joining (portions of) tables stored in different nodes leads to a high data communication
    - Impact on performance
- Two options to overcome the problem
    1. Go back to the logical design and restructure your data (query-based design)
    2. Execute joins by writing specific applications, accessing data stored in Cassandra
        - Store data in Cassandra and use MapReduce frameworks or Spark

# CQL – SELECT, restrictions: no join

- If you design your aggregates using the methodology, all wokload queries can be executed over your data without the need of join execution

- However, join execution might be needed for executing queries that do not belong to the workload

# Example

```
CREATE TABLE movie (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY ((title, director));
```

```
CREATE TABLE video (
    loc int PRIMARY KEY,
    type text,
    title text,
    director text);
```

```
SELECT year
FROM movie M, video V
WHERE M.title = V.title AND
M.director = V.director AND loc
= 1234;
```

Non admitted

# Example – approach 1

```
CREATE TABLE movie (
   title text,
   director text,
   year int,
   genre text,
   PRIMARY KEY ((title, director));
```

```
CREATE TABLE movie_video (
   title text,
   director text,
   year int,
   genre text,
   loc int PRIMARY KEY,
   type text);
```

```
CREATE TABLE video (
   loc int PRIMARY KEY,
   type text,
   title text,
   director text);
```

```
SELECT year
FROM movie_video
WHERE loc = 1234;
```

Admitted

Query-based design: go back to the design

# Example – approach 2

- Write a program that do the following:

1. executes

```
SELECT title, director
FROM videos
WHERE loc = 1234;
```

2. Let t represent the returned tuple
3. Execute

```
SELECT year
FROM movies
WHERE title = t.title AND director = t.director;
```

# CQL – SELECT, restrictions: selection conditions

- The WHERE clause must contain an equality-based selection condition on each attribute of the partition key
  - The system is able to identify the nodes contributing to the result
  - IN clause is also allowed (a set of nodes is identified)

- The WHERE clause can contain selection conditions on clustering columns, following the ordering provided in the primary key, with some additional restrictions
  - On the identified nodes, rows to be returned must be sequentially stored

# Example

```
SELECT*
FROM movies
WHERE title = 'pulp fiction'
```

Admitted

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, director);
```

```
SELECT*
FROM movies
WHERE  title = 'pulp fiction' AND
        director= 'quentin tarantino'
```

Admitted

Non admitted

```
SELECT*
FROM movies
WHERE director= 'quentin tarantino'
```

```
SELECT*
FROM movies
WHERE  title = 'pulp fiction' AND
        year = 2000
```

Non admitted

# CQL – SELECT, restrictions: selection conditions

- Data to be retrieved from a partition has to be sequentially stored

- Clustering columns support =, IN, >, >=, <=, <, CONTAINS operators
  - CONTAINS only for collection types, see later

- The WHERE clause can contain conditions over any prefix of the clustering column list, as defined in the primary key

- If more than one clustering column has been defined, range restrictions are allowed only on the last clustering column being restricted in the WHERE clause

# Example

```
CREATE TABLE movies (
   title text,
   director text,
   year int,
   genre text,
   PRIMARY KEY (title, director, year);
```

```
SELECT*
FROM movies
WHERE  title = 'pulp fiction' AND
       director= 'quentin tarantino' AND
       year > 2000
```

Admitted

```
SELECT*
FROM movies
WHERE  title = 'pulp fiction' AND
       year > 2000
```

Non admitted

# Example

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, year, director);
```

```
SELECT*
FROM movies
WHERE  title = 'pulp fiction' AND
        director= 'quentin tarantino' AND
        year > 2000
```

Non admitted

```
SELECT*
FROM movies
WHERE  title = 'pulp fiction' AND
        year > 2000
```

Admitted

# Indexes

- Exceptions to the previous rules are possible if indexes are created
- An index can be created on any column
  - B+-Tree indexes
  - Local at any node
- WHERE conditions on any attributes upon which an index has been created are admitted

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (   title,
                    director,
                    year);
```

```
CREATE INDEX ON movies(year);

SELECT*
FROM movies
WHERE  year > 2000;
```

Admitted

# Enforcing query execution

- Executing a non admitted query, you will get the following message

*Bad Request: 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.*

- The system tells you that the execution of your request might be inefficient

- You can force the query execution by using clause

`ALLOW FILTERING`

enforcing a full scan of data (quite inefficient)

# Example

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (   title,
                    director,
                    year);
```

```
CREATE INDEX ON movies(year);
```

```
SELECT*
FROM movies
WHERE  year > 2000;
```

Admitted

```
SELECT*
FROM movies
WHERE  year > 2000
ALLOW FILTERING ;
```

Admitted

# Other clauses

- Aggregate functions: COUNT, SUM, MAX, MIN,…
- GROUP BY
  - Over primary key columns in defined order as arguments
- ORDER BY
  - Depending on the clustering order

# CQL – Collection types

- Extension of basic features of a column-family data mode with nested atomic values: sets, lists, maps
- Sets hold list of unique elements that are visible to the system
- Lists hold ordered, possibly repeating elements
- Maps hold a list of key-value pairs
- Collection types cannot be nested
- Collection field cannot be used in primary keys

```
CREATE TABLE mytable(
    row text,
    Y text,
    myset set<text>,
    mylist list<int>,
    mymap map<text, text>,
    PRIMARY KEY (row)
);
```

# CQL – Collection types - INSERT

```
CREATE TABLE mytable(
        row text,
        Y text,
        myset set<text>,
        mylist list<text>,
        mymap map<int, text>,
        PRIMARY KEY (row)
);
```

- Inserting

INSERT INTO mytable (row, **myset**)
VALUES (123, **{ 'apple', 'banana'}**);

INSERT INTO mytable (row, **mylist**)
VALUES (123, **['apple','banana','apple']**);

INSERT INTO mytable (row, **mymap**)
VALUES (123, **{1:'apple',2:'banana'}**)

# CQL – Collection types - UPDATE

```
CREATE TABLE mytable(
      row text,
      Y text,
      myset set<text>,
      mylist list<text>,
      mymap map<text, text>,
      PRIMARY KEY (row)
);
```

- Updating

```
UPDATE mytable SET myset = myset + {'apple','banana'}
  WHERE row = 123;
UPDATE mytable SET myset = myset - { 'apple' }
  WHERE row = 123;

UPDATE mytable SET mylist = mylist + ['apple','banana']
  WHERE row = 123;
UPDATE mytable SET mylist = ['banana'] + mylist
  WHERE row = 123;

UPDATE mytable SET mymap['fruit'] = 'apple'
  WHERE row = 123
UPDATE mytable SET mymap = mymap + { 'fruit':'apple'}
  WHERE row = 123
```

# Example – collection types

```
CREATE TABLE clients (
codCli int PRIMARY KEY,
name text,
surname text,
birthdate date,
emails set<text>,
phones map<text, text>,
hobbies list<text>
);
```

```
SELECT * FROM clients
WHERE name = 'Benjamin';

SELECT * FROM clients
WHERE emails CONTAINS
            'Benjamin@oops.com';

SELECT * FROM clients
WHERE hobbies CONTAINS tennis';
```

```
CREATE INDEX ON clients (name);
CREATE INDEX ON clients (emails);
CREATE INDEX ON clients (hobbies);
CREATE INDEX ON clients (keys(phones));
CREATE INDEX ON clients (values(phones));
CREATE INDEX ON clients (entries(phones));
```

```
SELECT * FROM clients
WHERE phones CONTAINS KEY 'office';

SELECT * FROM clients
WHERE phones CONTAINS '0108567586';

SELECT * FROM clients
WHERE phones['office'] = '0108567586';
```

# CQL - User Defined Types

- User defined types allow a compact usage of complex data
- Nesting
- An alternative approach for dealing with associations and joins

# Example

client: { **codCli**,

name, surname, birthdate,
address: {city, street, streetNumber, postalCode},
movies: [ {comment, title, director,…}]
}

```
CREATE TYPE address_t(
city text,
street text,
streetNumber int,
postalCode int);
```

```
CREATE TYPE movie_t(
comment text,
title text,
director int,
…);
```

```
CREATE TABLE clients (
codCli int PRIMARY KEY,
name text,
surname text,
birthdate date,
address frozen<address_t>,
movies set<frozen<movie_t>>
);
```

Values for `address_t` treated as a blob (no way to select or update components, you should do that at the application level)

Values for `movie_t` treated as a blob (no way to select components,  you should do that at the application level)

# Frozen types

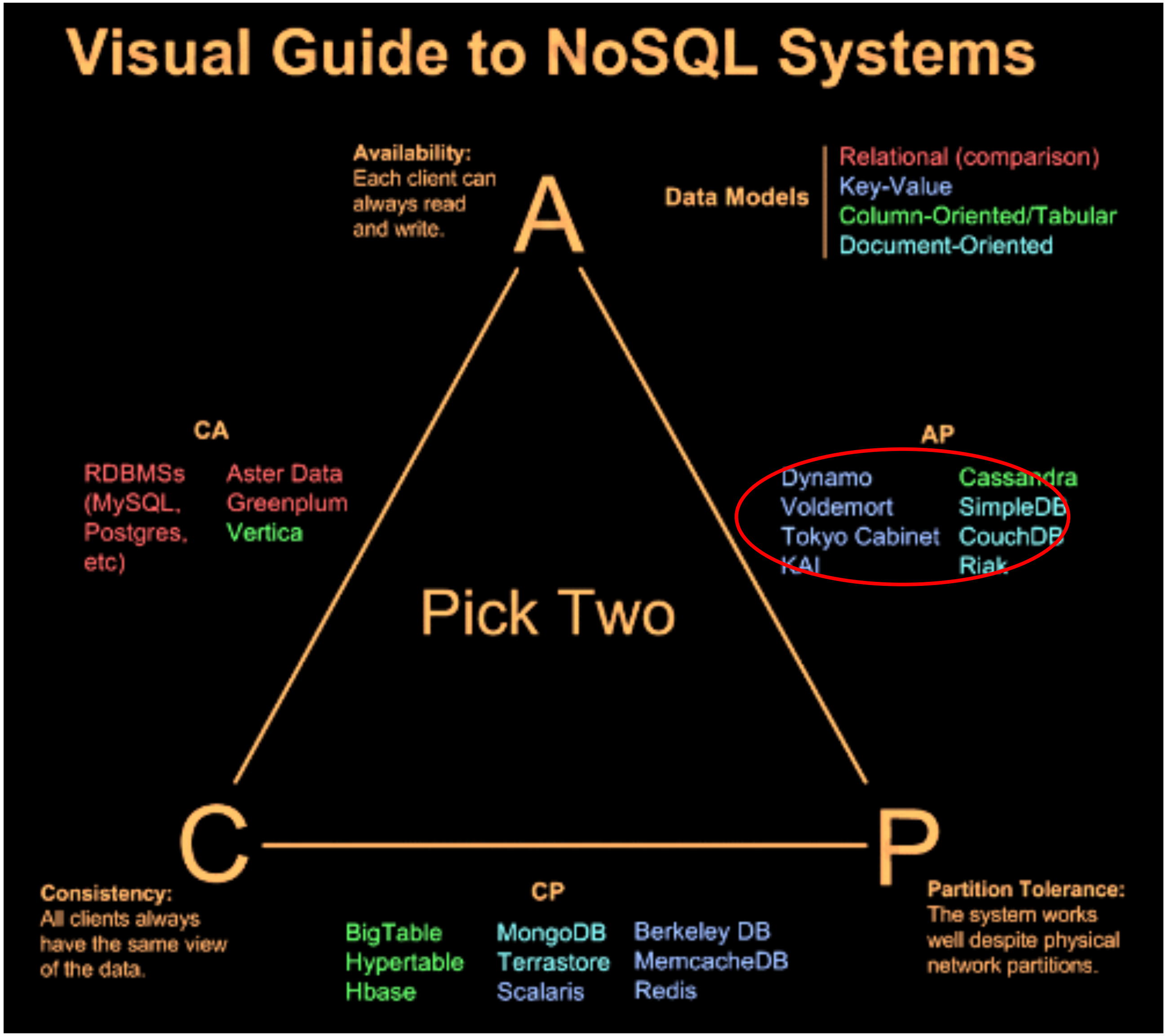
- Frozen <> makes a component opaque for the system
- Frozen<> can be used with both user defined types and collection types, it is mandatory in most situations
- With frozen <>, we can overcome most limitations related to collection types and UDTs
- Collection types and UDTs nesting is allowed by using frozen types
- Frozen types can be used inside either frozen or non frozen collection types
- Collection fields with frozen types can be used as primary key

# Architecture

# Cassandra in short

| Feature | In Cassandra |
|---|---|
| Model | Column-family |
| Architecture | P2P |
| Query language | Supported (CQL) |
| Reference scenarios | Transactional (read/write intensive) |
| Partitioning | Consistent hashing |
| Indexes | Local secondary index |
| Replication | Leader-less |
| Consistency | Eventual consistency |
| Availability | High, tunable |
| Fault tolerance | High (no master, P2P ring) |
| Transactions | No ACID transactions |
| CAP theorem | AP |
| Distributed by | Apache<br>third-parties |

# CAP theorem




Cassandra

AP: Highly available system

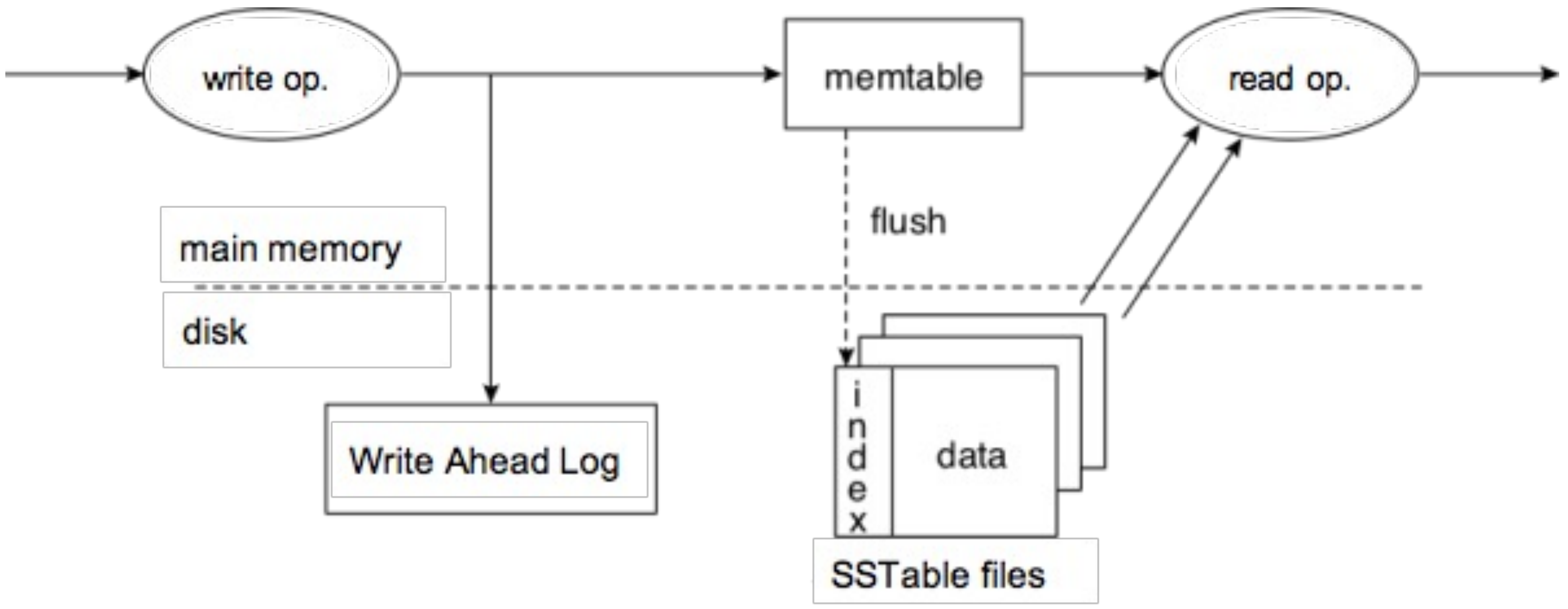
# Local Persistence

- Organization of local data store at nodes relies on local file system

- Write operations happens in three steps
  - Write to commit log in local disk of the node
  - Update in-memory data structure (write-back cache of data rows that can be looked up by key)
  - When the in-memory structure is full, it is written (flushed) on disk as an SSTable

- Read operation
  - Looks up in-memory data structures first before looking up files on disk

# Local persistence




Persistency + durability
but also high throughput of write operations

# Configurable read/write consistency

- A consistency level can be specified for a given query session or for each read and write query
- It allows to set parameters w and r in the formula w + r > n (n number of replicas)
- Common consistency levels

ALL – All nodes in the cluster must confirm (even non-local nodes)

QUORUM – A quorum of nodes (half the replication factor plus one) in the cluster must confirm

ONE / TWO / THREE – One, two or three nodes in the cluster must confirm

| WRITE | | |
|---|---|---|
| Level | Description | Usage |
| ALL | A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition. | Provides the highest consistency and the lowest availability of any other level. |
| QUORUM | A write must be written to the commit log and memtable on a quorum of replica nodes across *all* datacenters. | Use in single or multiple datacenter clusters to maintain strong consistency across the cluster. Use if you can tolerate some level of failure. |
| ONE | A write must be written to the commit log and memtable of at least one replica node. | Satisfies the needs of most users because consistency requirements are not stringent. |
| TWO | A write must be written to the commit log and memtable of at least two replica nodes. | Similar to ONE. |
| THREE | A write must be written to the commit log and memtable of at least three replica nodes. | Similar to TWO. |

| READ | | |
|---|---|---|
| Level | Description | Usage |
| ALL | Returns the record after all replicas have responded. The read operation will fail if a replica does not respond. | Provides the highest consistency of all levels and the lowest availability of all levels. |
| QUORUM | Returns the record after a quorum of replicas from all datacenters has responded. | Used in single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure. |
| ONE | Returns a response from the closest replica, as determined by the snitch. By default, a read repair runs in the background to make the other replicas consistent. | Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write. |
| TWO | Returns the most recent data from two of the closest replicas. | Similar to ONE. |
| THREE | Returns the most recent data from three of the closest replicas. | Similar to TWO. |

# Transactions

- No ACID transactions with rollback or locking mechanisms
- Cassandra offers Atomic, Isolated, and Durable (AID) transactions with eventual/tunable consistency (no classical consistency, no integrity constraints)

# Column-family stores use cases

# Suitable use cases

- *Applications with a high number of writes*

- Event Logging
  - Great choice to store event information

- Content Management Systems, Blogging Platforms
  - You can store blog entries with tags, categories, links, and trackbacks in different columns
  - Comments can be either stored in the same row or moved to a different keyspace
  - Similarly, blog users and the actual blogs can be put into different column families

- Counters
  - Often, in Web applications you need to count and categorize visitors of a page to calculate analytics
  - Counter can be maintained in Cassandra

# When not to use


- Systems that require ACID transactions for writes and reads
- Prototypes
  - during the early stages query patterns may change and this requires to change the column family design
  - in Cassandra, the cost may be higher for query change as compared to schema change

# Cassandra, MapReduce, Hadoop

- Integration with Hadoop MapReduce (MapReduce I/O from/to Cassandra)
- Integration with Pig (Pig I/O from/to Cassandra)