Modern Database Systems

Column-family stores

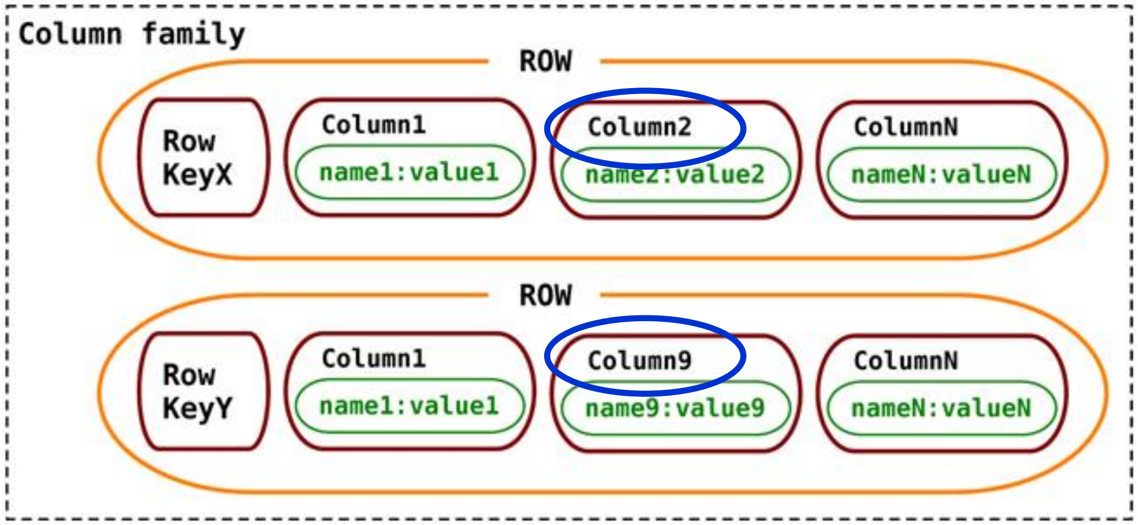
Doc. RNDr. Irena Holubova, Ph.D.
Irena.Holubova@matfyz.cuni.cz

Column-Family Stores

Basic Characteristics

Note: two different meanings of column-oriented!!

- Also “columnar” or “column-oriented”
- Column families = rows that have <u>many</u> columns associated with a row key
- Column families are groups of related data that is often accessed together
  - e.g., for a customer we access all profile information at the same time, but not orders



Second Meaning of Column-Oriented

- Stores data tables as columns rather than as rows
  - Maps data to rowIds = a closer structure to an index
  - More efficient query operations (some of them)
    - E.g. Find all people with name Jones
  - Retrieving a whole row is slower

RowId	EmpId	Lastname	Firstname	Salary
001	10	Smith	Joe	40000
002	12	Jones	Mary	50000
003	11	Johnson	Cathy	44000
004	22	Jones	Bob	550

⇨

```
001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
```

⇘

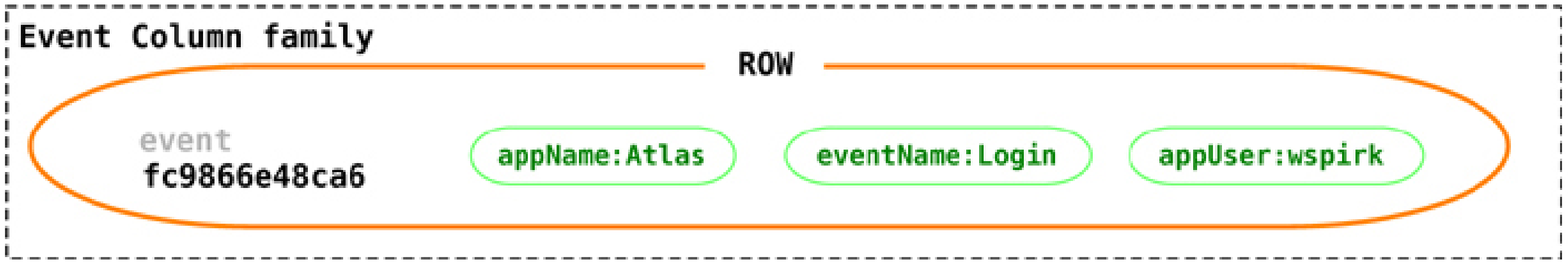
```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

```
Jones:002,004
```

⇔

Column-Family Stores

Suitable Use Cases



Event Logging
- Ability to store any data structures → good choice to store event information

Content Management Systems, Blogging Platforms
- We 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
- Blog users and the actual blogs can be put into different column families

Column-Family Stores

When Not to Use

Systems that Require ACID Transactions

- Column-family stores are <u>not</u> just a special kind of RDBMSs with variable set of columns!

Aggregation of the Data Using Queries

- (such as SUM or AVG)
- Have to be (usually) done on the client side

For Early Prototypes

- We are not sure how the query patterns may change
- As the query patterns change, we have to change the column family design

Column-Family Stores
Representatives

Google's BigTable

H·BASE

Cassandra ✓

HYPERTABLE

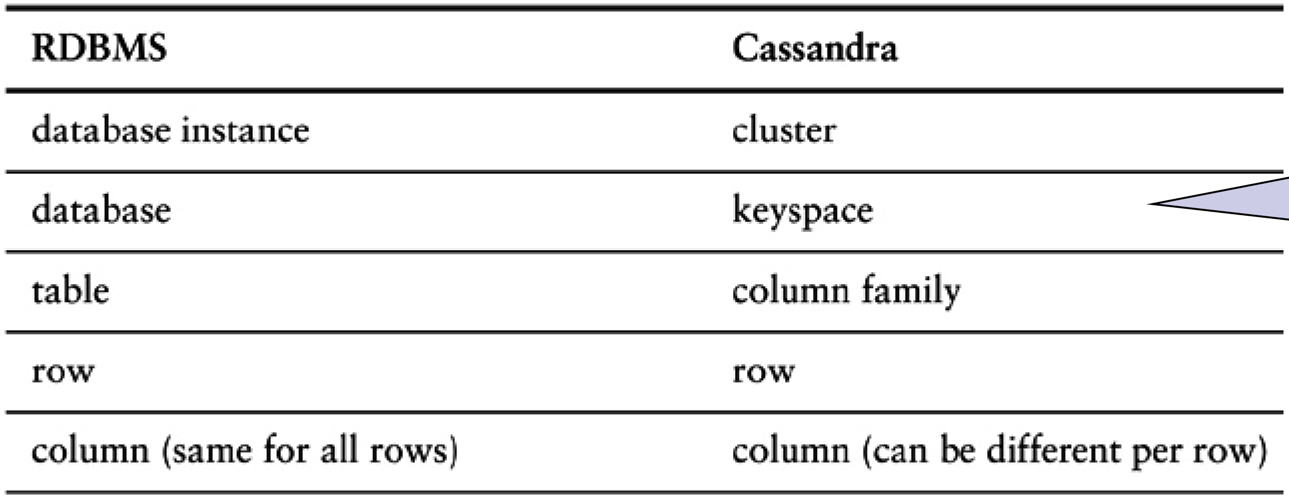
SimpleDB

Apache Cassandra

- Developed at Facebook
- Initial release: 2008
- Stable release: 2013
  - Apache Licence
- Written in: Java
- OS: cross-platform
- Operations:
  - CQL (Cassandra Query Language)
  - MapReduce support
    - Can cooperate with Hadoop (data storage instead of HDFS)

http://cassandra.apache.org/

Cassandra
Terminology

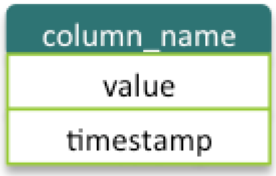


RDBMS	Cassandra
database instance	cluster
database	keyspace
table	column family
row	row
column (same for all rows)	column (can be different per row)

Usually one per application

- Column = basic unit, consists of a **name-value** pair
  - Name serves as a key
  - Stored with a **timestamp** (expired data, resolving conflicts, …)
- Row = a collection of columns attached or linked to a key
- Column family = a collection of <u>similar</u> rows
  - Rows do not have to have the same columns

3-tuple



Cassandra

Data Model – Example

```
{ "pramod-sadalage" : {
    firstName: "Pramod",
    lastName: "Sadalage",
    lastVisit: "2012/12/12" }
  "martin-fowler" : {
    firstName: "Martin",
    lastName: "Fowler",
    location: "Boston" } }
```

- pramod-sadalage row and martin-fowler row with different columns; both rows are a part of a column family

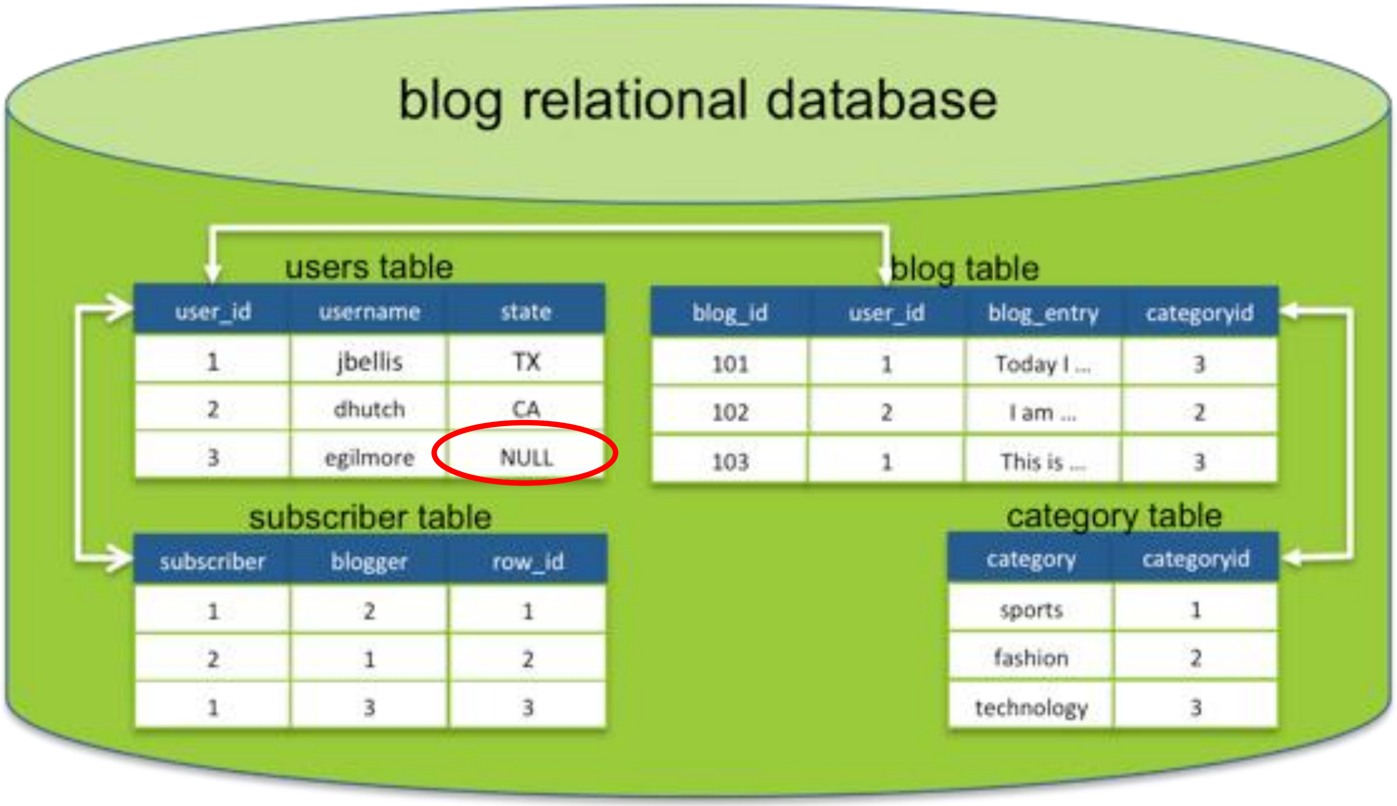
```
{ name: "firstName",
  value: "Martin",
  timestamp: 12345667890 }
```

- Column key of firstName and the value of Martin

Cassandra

Column-families vs. Relations

- We do not need to model all of the columns up front
  - Each row is not required to have the same set of columns
  - Usually we assume similar sets of columns
    - Related data
    - Can be extended when needed
- No formal foreign keys
  - Joining column families at query time is usually not supported
  - We need to pre-compute the query / use a secondary index



users table

user_id	username	state
1	jbellis	TX
2	dhutch	CA
3	egilmore	NULL

blog table

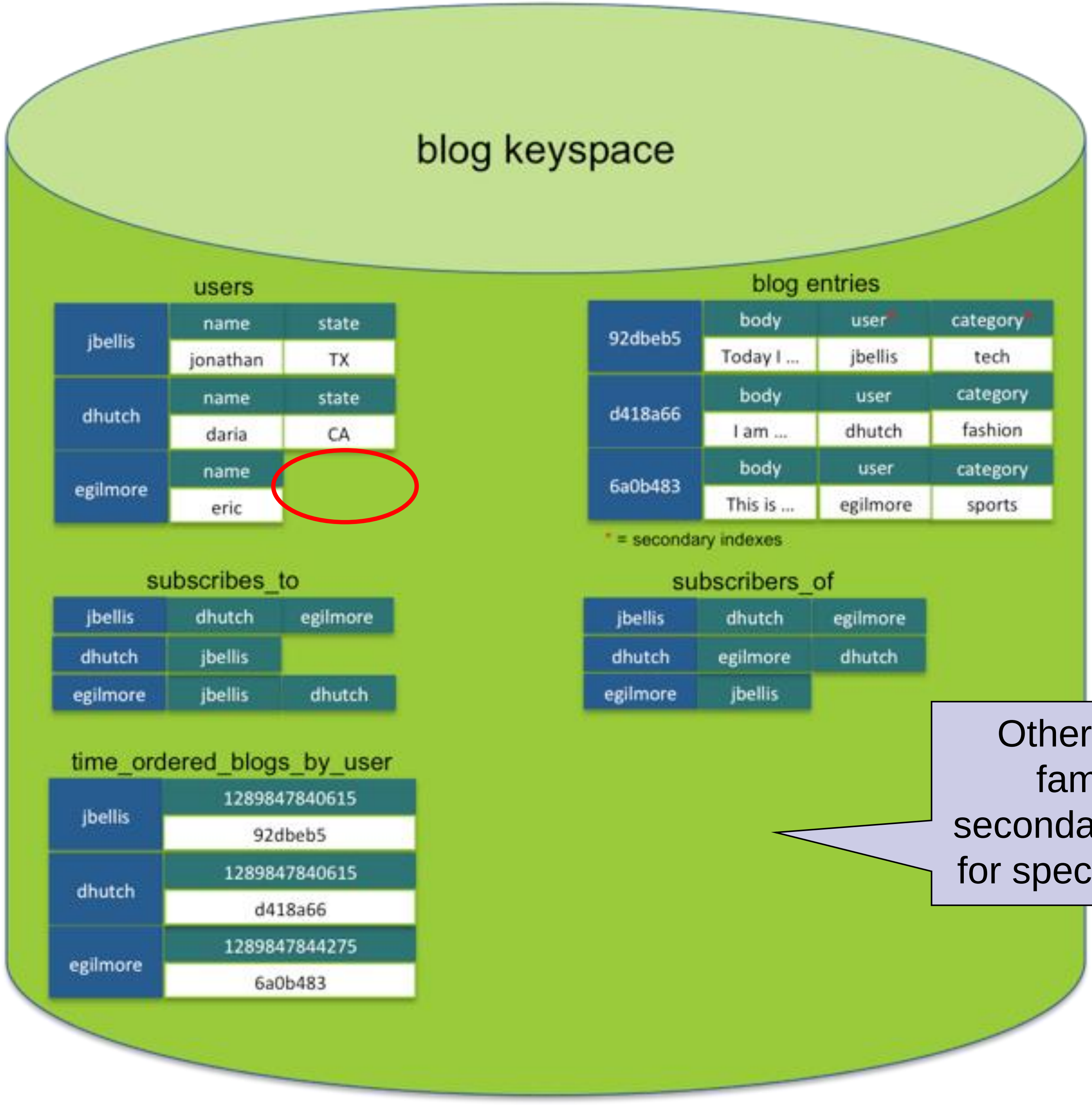
blog_id	user_id	blog_entry	categoryid
101	1	Today I ...	3
102	2	I am ...	2
103	1	This is ...	3

subscriber table

subscriber	blogger	row_id
1	2	1
2	1	2
1	3	3

category table

category	categoryid
sports	1
fashion	2
technology	3



users

jbellis	name	state
	jonathan	TX
dhutch	name	state
	daria	CA
egilmore	name	
	eric	

blog entries

92dbeb5	body	user*	category*
	Today I ...	jbellis	tech
d418a66	body	user	category
	I am ...	dhutch	fashion
6a0b483	body	user	category
	This is ...	egilmore	sports

* = secondary indexes

subscribes_to

jbellis	dhutch	egilmore
dhutch	jbellis	
egilmore	jbellis	dhutch

subscribers_of

jbellis	dhutch	egilmore
dhutch	egilmore	dhutch
egilmore	jbellis	

time_ordered_blogs_by_user

jbellis	1289847840615
	92dbeb5
dhutch	1289847840615
	d418a66
egilmore	1289847844275
	6a0b483

Other column families / secondary indexes for special queries

Cassandra

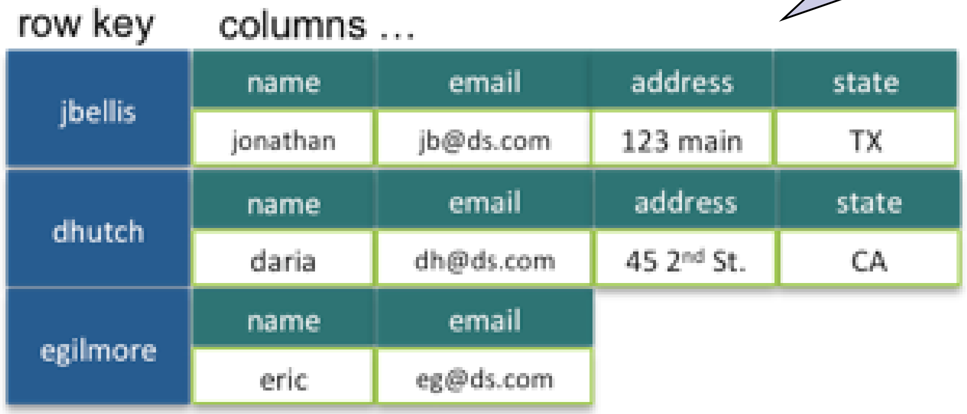
Column-families

- Can define metadata about columns
  - Actual columns of a row are determined by client application
  - Each row can have a different set of columns
- Static – similar to a relational database table
  - Rows have the same set of columns
  - Not required to have all of the columns defined
- Dynamic – takes advantage of Cassandra's ability to use arbitrary application-supplied column names
  - Pre-computed result sets
  - Stored in a single row for efficient data retrieval
  - Row = a snapshot of data that satisfy a given query
    - Like a materialized view

Cassandra

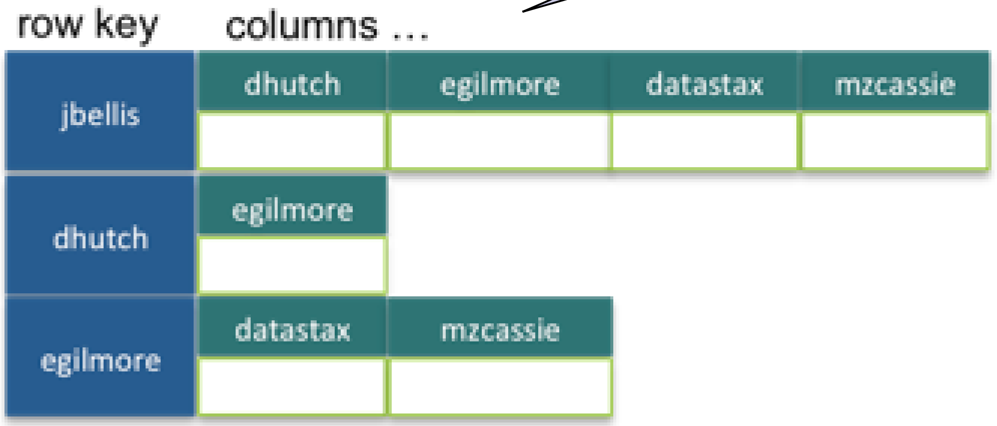
Column-families

static



row key	columns …			
jbellis	name	email	address	state
	jonathan	jb@ds.com	123 main	TX
dhutch	name	email	address	state
	daria	dh@ds.com	45 2^{nd} St.	CA
egilmore	name	email		
	eric	eg@ds.com		

dynamic

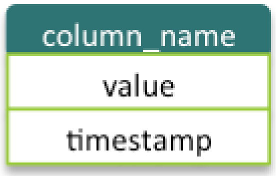


row key	columns …			
jbellis	dhutch	egilmore	datastax	mzcassie
dhutch	egilmore			
egilmore	datastax	mzcassie		

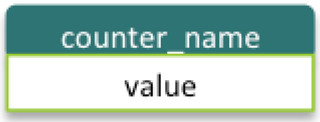
Users that subscribe to a particular user's blog

Cassandra

Columns

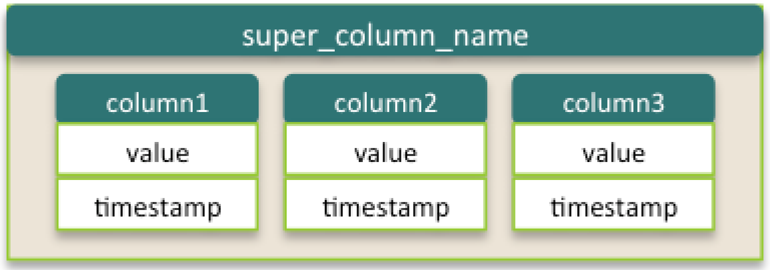


- Column is the smallest increment of data
  - Name + value + timestamp
  - Value can be empty (e.g., materialized views)
- Can be indexed on their name
  - Using a secondary index
  - Primary index = row key
    - Ensure uniqueness, speeds up access, can influence storage order
- Types:
  - Expiring – with optional expiration date called TTL
    - Can be queried
  - Counter – to store a number that incrementally counts the occurrences of a particular event or process
    - E.g., to count the number of times a page is viewed
    - Operation increment/decrement with a specified value
    - Internally ensures consistency across all replicas
  - Super – add another level of nesting
    - To group multiple columns based on a common lookup value

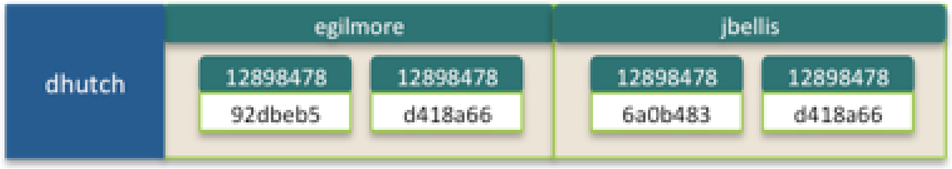


Cassandra
Super columns

```
{   name: "book:978-0767905923",
    value: { author: "Mitch Albon",
             title: "Tuesdays with Morrie",
             isbn: "978-0767905923"   } }
```



- super column – a column consisting of a map of columns
  - It has a name and value involving the map of columns
- super column family – a column family consisting of super columns
  - vs. standard column family



Cassandra

Column Families

- A key <u>must</u> be specified
- Data types for columns <u>can</u> be specified
- Options <u>can</u> be specified

```
CREATE COLUMNFAMILY Fish (key blob PRIMARY KEY);
CREATE COLUMNFAMILY FastFoodEatings (user text PRIMARY KEY)
    WITH comparator=timestamp AND default_validation=int;
CREATE COLUMNFAMILY MonkeyTypes (
    KEY uuid PRIMARY KEY,
    species text,
    alias text,
    population varint
) WITH comment='Important biological records'
  AND read_repair_chance = 1.0;
```

Cassandra
Column Families

- Comparator = data type for a column name
- Validator = data type for a column (or row key) value
- Data types do not need to be defined
  - Default: `BytesType`, i.e. arbitrary hexadecimal bytes

- Basic operations: GET, SET, DEL

We will focus on it later

- From new versions of Cassandra and CQL: new strategy
  - But the capabilities remain the same
    - i.e., we can still create tables with arbitrary columns

Column-Family Stores

Column Families – Example

different syntax (different version)

```
CREATE COLUMNFAMILY users
with key_validation_class = 'UTF8Type'
 and comparator = 'UTF8Type'
 and column_metadata = [
       {column_name : 'name', validation_class : UTF8Type},
       {column_name : 'birth_year', validation_class : Int32Type}];

SET users['jbellis']['name'] = 'Jonathan Ellis';
SET users['jbellis']['birth_year'] = 1976;
SET users['jbellis']['home'] = long(1112223333);
SET users['jbellis']['work'] = long(2223334444);

GET users['jbellis'];
GET users['jbellis']['home'];

DEL users['jbellis']['home'];
DEL users['jbellis'];
```

Column-Family Stores

Column Families – Best Practice

- Comparators
  - Within a row, columns are stored in sorted order by their column name
  - Static column families:
    - Typically strings
    - Order unimportant
  - Dynamic column families
    - Order is usually important (e.g. timestamps)
- Validators
  - Define a default row key validator using property `key_validation_class`
  - Static column families:
    - Define each column and its associated type
  - Dynamic column families
    - Column names are not known ahead
    - Specify `default_validation_class`

Cassandra

CQL – New Approach

- Cassandra query language
- SQL-like commands
  - CREATE, ALTER, UPDATE, DROP, DELETE, TRUNCATE, INSERT, …
- Much simpler than SQL
  - Does not allow joins or subqueries
  - Where clauses are simple
  - …
- Different approach than column families (since CQL 3 called tables)
  - More general
  - Closer to key/value and document databases

Cassandra

CQL Data Types

`ascii`	ASCII character string
`bigint`	64-bit signed long
`blob`	Arbitrary bytes (no validation)
`boolean`	true or false
`counter`	Counter column (64-bit long)
`decimal`	Variable-precision decimal
`double`	64-bit IEEE-754 floating point
`float`	32-bit IEEE-754 floating point
`int`	32-bit signed int
`text`	UTF8 encoded string
`timestamp`	A timestamp
`uuid`	A UUID in standard UUID format
`varchar`	UTF8 encoded string
`varint`	Arbitrary-precision integer
…	…

Universally Unique Identifier

Cassandra
Working with a Key Space

One replication factor vs. `NetworkTopologyStrategy` = individual replication factor for each data center

```
CREATE KEYSPACE Excelsior
WITH replication = {'class': 'SimpleStrategy',
                    'replication_factor' : 3};
```

- Create a key space with a specified replication strategy and parameters

```
USE Excelsior;
```

- Set a keyspace as the current working keyspace

```
ALTER KEYSPACE Excelsior
WITH replication = {'class': 'SimpleStrategy',
                    'replication_factor' : 4};
```

- Alter the properties of an existing keyspace

```
DROP KEYSPACE Excelsior;
```

- Drop a keyspace

Cassandra

Working with a Table – Primary Key

```
CREATE TABLE timeline (
    userid uuid,
    posted_month int,
    posted_time uuid,
    body text,
    posted_by text,
    PRIMARY KEY (userid, posted_month, posted_time) )
WITH compaction = { 'class' : 'LeveledCompactionStrategy' };
```

Explained in the internals part

- Creating a table with name, columns and other options

- Primary key is compulsory
  - Partition key = the first column (or a set of columns if parenthesised)
    - Records are stored on the same node
  - Clustering columns
    - Determine per-partition clustering, i.e., the order for physical storing of rows in a partition!!

Cassandra

Working with a Table – Column Expiration

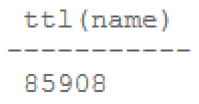
```
CREATE TABLE excelsior.clicks (
  userid uuid,
  url text,
  date timestamp,
  name text,
  PRIMARY KEY (userid, url) );

INSERT INTO excelsior.clicks (userid, url, date, name)
VALUES (3715e600-2eb0-11e2-81c1-0800200c9a66,
   'http://apache.org', '2013-10-09', 'Mary')
USING TTL 86400;
```

- When the data will expire

```
SELECT TTL (name) from excelsior.clicks
  WHERE url = 'http://apache.org' ALLOW FILTERING;
```

- Determine how much longer the data has to live



```
 ttl(name)
------------
 85908
```

Cassandra

Working with a Table – Collections

- Collection types:
  - **set** – a set of unique values
    - Returned in alphabetical order, when queried
  - **list** – ordered list of elements
    - Can store the same value multiple times
    - Returned sorted according to index value in the list
  - **map** – name + value pairs
  - Each element is internally stored as one Cassandra column

  => Each element can have an individual time-to-live

Cassandra

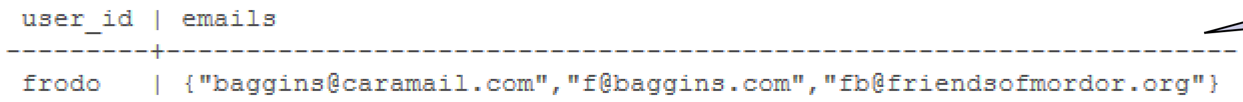
Working with a Table – Set

```
CREATE TABLE users (
  user_id text PRIMARY KEY,
  first_name text,
  last_name text,
  emails set<text> );

INSERT INTO users (user_id, first_name, last_name, emails)
VALUES('frodo', 'Frodo', 'Baggins', {'f@baggins.com', 'baggins@gmail.com'});

UPDATE users SET emails = emails + {'fb@friendsofmordor.org'}
WHERE user_id = 'frodo';

SELECT user_id, emails FROM users WHERE user_id = 'frodo';
```



```
 user_id | emails
---------+-------------------------------------------------------------------
 frodo   | {"baggins@caramail.com","f@baggins.com","fb@friendsofmordor.org"}
```

order

```
UPDATE users SET emails = emails - {'fb@friendsofmordor.org'}
WHERE user_id = 'frodo';

UPDATE users SET emails = {} WHERE user_id = 'frodo';
```

Cassandra

Working with a Table – List

```
ALTER TABLE users ADD top_places list<text>;

UPDATE users SET top_places = [ 'rivendell', 'rohan' ]
WHERE user_id = 'frodo';

UPDATE users SET top_places = [ 'the shire' ] + top_places
WHERE user_id = 'frodo';

UPDATE users SET top_places = top_places + [ 'mordor' ]
WHERE user_id = 'frodo';

UPDATE users SET top_places[2] = 'riddermark'
WHERE user_id = 'frodo';

DELETE top_places[3] FROM users WHERE user_id = 'frodo';

UPDATE users SET top_places = top_places - ['riddermark']
WHERE user_id = 'frodo';
```

Cassandra

Working with a Table – Map

```
ALTER TABLE users ADD todo map<timestamp, text>;

UPDATE users SET todo = { '2012-9-24' : 'enter mordor',
'2012-10-2 12:00' : 'throw ring into mount doom' }
WHERE user_id = 'frodo';

UPDATE users SET todo['2012-10-2 12:00'] =
'throw my precious into mount doom'
WHERE user_id = 'frodo';

INSERT INTO users (user_id, todo) VALUES ('frodo', {
'2013-9-22 12:01' : 'birthday wishes to Bilbo',
'2013-10-1 18:00' : 'Check into Inn of Prancing Pony' });

DELETE todo['2012-9-24'] FROM users
WHERE user_id = 'frodo';
```

Cassandra
Working with a Table

```
DROP TABLE timeline;
```
- Delete a table including all data

```
TRUNCATE timeline;
```
- Remove all data from a table

```
CREATE INDEX userIndex ON timeline (posted_by);
```
- Create a (secondary) index
- Allow efficient querying of other columns than key

```
DROP INDEX userIndex;
```
- Drop an index

Cassandra
Querying

- Remember: no joins, just simple conditions
  - For simple data reads

```
SELECT * FROM users
WHERE firstname = 'jane' and lastname='smith'
ALLOW FILTERING;
```

- Filtering (`WHERE`)

```
SELECT * FROM emp
WHERE empID IN (130,104)
ORDER BY deptID DESC;
```

- Ordering (`ORDER BY`)

Cassandra

Querying

optional

```
SELECT select_expression
FROM keyspace_name.table_name
WHERE relation AND relation ...
GROUP BY columns
ORDER BY ( clustering_key ( ASC | DESC )...)
LIMIT n
ALLOW FILTERING
```

- `select_expression`:
  - List of columns
  - `DISTINCT`
  - `COUNT`
  - Aliases (`AS`)
  - `TTL(column_name)`
  - `WRITETIME(column_name)`

Cassandra
Querying

- `relation`:
  - column_name (= | < | > | <= | >=) key_value
  - column_name IN ((key_value,...))
  - TOKEN (column_name, ...) (= | < | > | <= | >=) (term | TOKEN (term, ...))

hash

- `term`:
  - constant
  - set/list/map

Cassandra

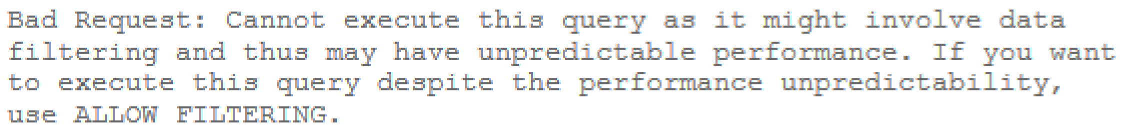
Querying – `GROUP BY`

- Groups rows of a table according to certain columns
- Only groupings induced by primary key columns are allowed!
- Aggregate functions
  - `COUNT, MIN, MAX, SUM, AVG`
  - User-defined
  - When a non-grouping column is selected without an aggregate function, the first value encounter is always returned

Cassandra

Querying – `ALLOW FILTERING`

- Non-filtering queries
  - Queries where we know that all records read will be returned (maybe partly) in the result set
  - Have predictable performance
- Attempt a potentially expensive (i.e., filtering) query
- `ALLOW FILTERING`
  - “We know what we are doing”
  - Usually together with `LIMIT n`



```
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.
```

Cassandra

Querying – `ALLOW FILTERING`

```
CREATE TABLE users (
    username text PRIMARY KEY,
    firstname text,
    lastname text,
    birth_year int,
    country text
);
CREATE INDEX ON users(birth_year);
```

query performance proportional to the amount of data returned

```
SELECT * FROM users;

SELECT firstname, lastname FROM users
WHERE birth_year = 1981;
```

Cassandra
Querying – `ALLOW FILTERING`

```
SELECT firstname, lastname
FROM users
WHERE birth_year = 1981 AND country = 'FR';
```

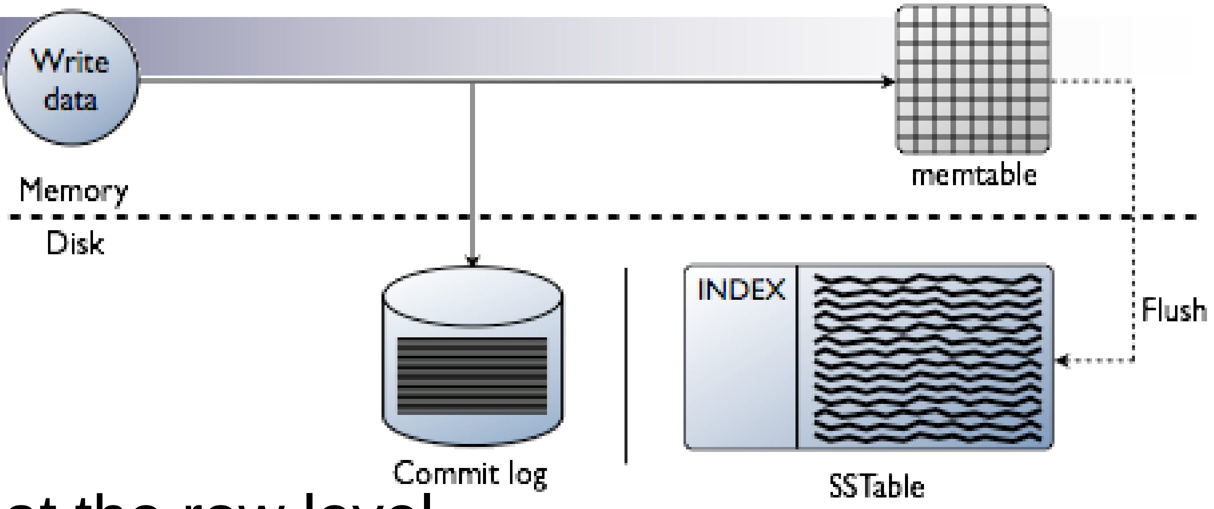
No guarantee that Cassandra won't have to scan large amount of data even if the result is small

```
SELECT firstname, lastname
FROM users
WHERE birth_year = 1981 AND country = 'FR'
ALLOW FILTERING;
```

More on Internals

Cassandra

Writes



- A write is atomic at the row level

1. When a write occurs:
   a. The data are stored in memory (memtable)
   b. Writes are appended to commit log on disk
      - Durability after HW failure
2. The more a table is used, the larger its memtable needs to be
   - Size > (configurable) threshold ⇒ the data is put in a queue to be flushed to disk
3. The memtable data is flushed to SSTables on disk
   - Sorted string table
4. Data in the commit log is purged after its corresponding data in the memtable is flushed to the SSTable

Cassandra
Writes

- Memtable and SSTables are maintained per table
- SSTables are immutable
  - ⇒ A row is typically stored across multiple SSTable files
  - ⇒ Read must combine row fragments from SSTables and un-flushed Memtables
- Memory structures for each SSTable:
  - Partition index – a list of primary keys and the start position of rows in the data file
  - Partition summary – a subset of the partition index
    - By default 1 primary key out of every 128 is sampled
    - To speed up searching

Cassandra

Writes

Write example:

```
write (k1, c1:v1)
write (k2, c1:v1 c2:v2)
write (k1, c1:v4 c3:v3 c2:v2)
```

Memtable:

```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```

- Data is sorted
- Column names are not repeated

Commit log:

```
k1, c1:v1
k2, c1:v1 c2:v2
k1, c1:v4 c3:v3 c2:v2
```

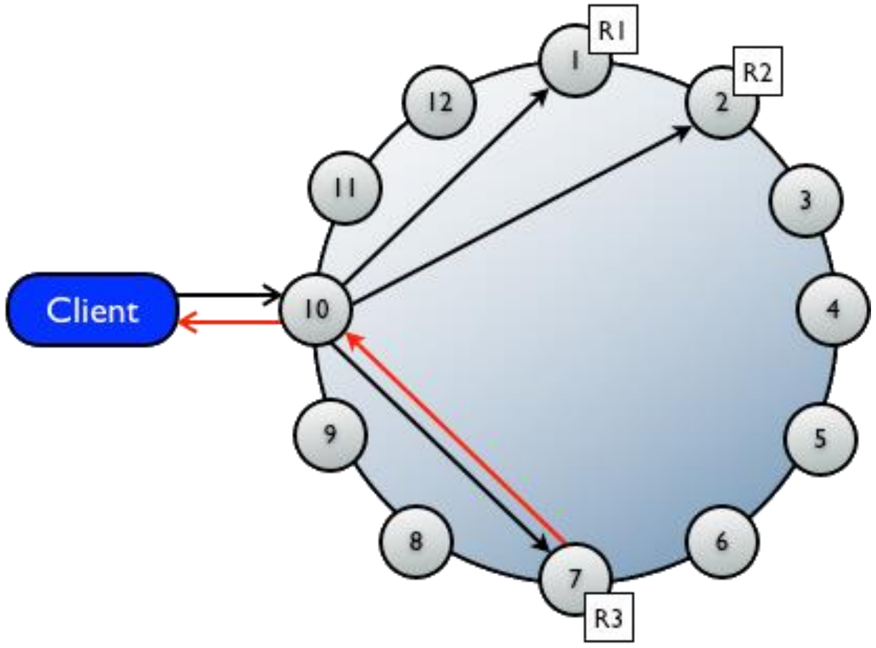
SSTable:

```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```

After flushing memtable on disk

Cassandra
Write Request

- Goes to any node (coordinator)
  - A proxy between the client application and the nodes
- Sends a write request to all replicas that own the row being written
  - Write consistency level = how many replicas must respond with success
    - Success = the data was written to commit log and memtable
- Example:
  - 12 node cluster, replication factor = 3, write consistency level = ONE
  - The first node to complete the write responds back to coordinator
  - Coordinator proxies the success message back to the client



Cassandra

Reads

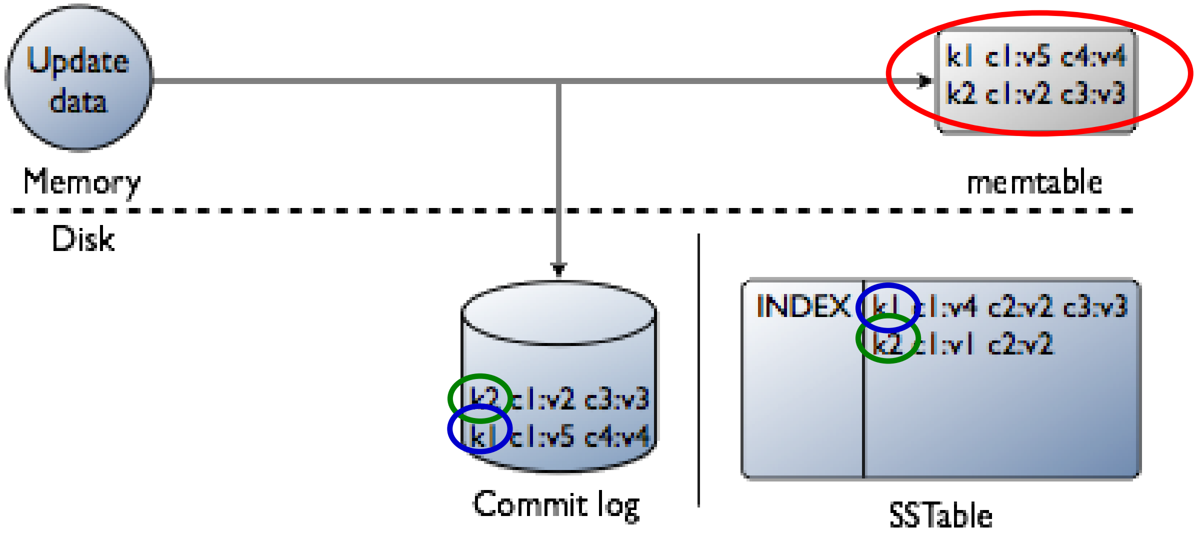
- Types of read requests a coordinator can send to a replica:
  - Direct read request – limited by the read consistency level
  - Background read repair request
- Steps:
  1. The coordinator contacts replicas specified by the read consistency level
     - Sends requests to those that currently respond fastest
  2. Data from replicas are compared to see if they are consistent
     - The most recent data (based on timestamp) is used
  3. Read repair: The coordinator contacts and compares the data from all the remaining replicas that own the row in the background
     - If the replicas are inconsistent, the coordinator issues writes

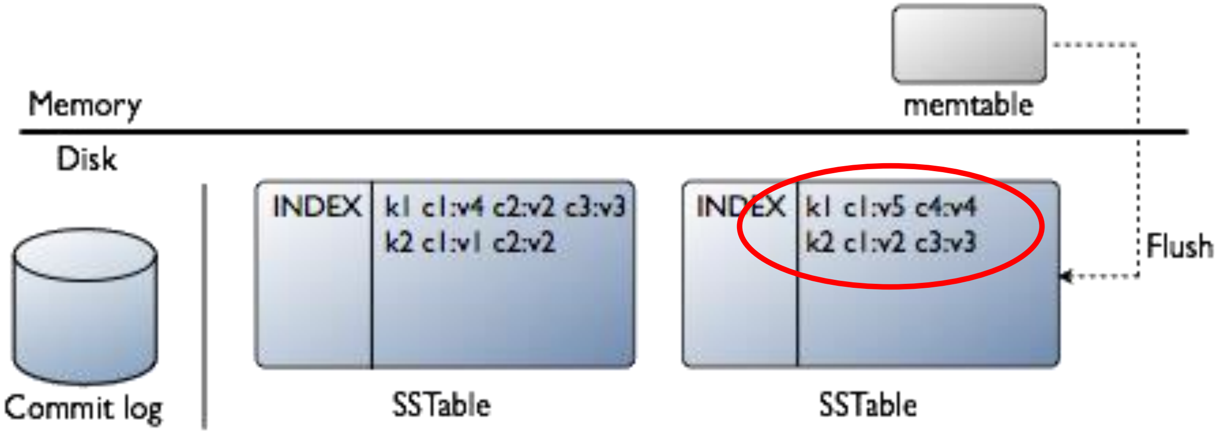
Cassandra
Updates

- Insert and update operations are identical
- Any number of columns can be inserted/updated at the same time
- Cassandra does not overwrite the rows
  - It groups inserts/updates in the memtable
  - See the example for writes
- Upsert = insert or update depending on the (non)existence of the data
  - Columns are overwritten only if the timestamp in the new version is more recent
    - Timestamp is provided by the client $\Rightarrow$ the clients should be synchronized
  - Otherwise the updates are stored into a new SSTable
    - Merged periodically on background using compaction process

Cassandra

Updates





Cassandra

Deletes

- Delete of a row = a delete of its columns
- After an SSTable is written, it is immutable

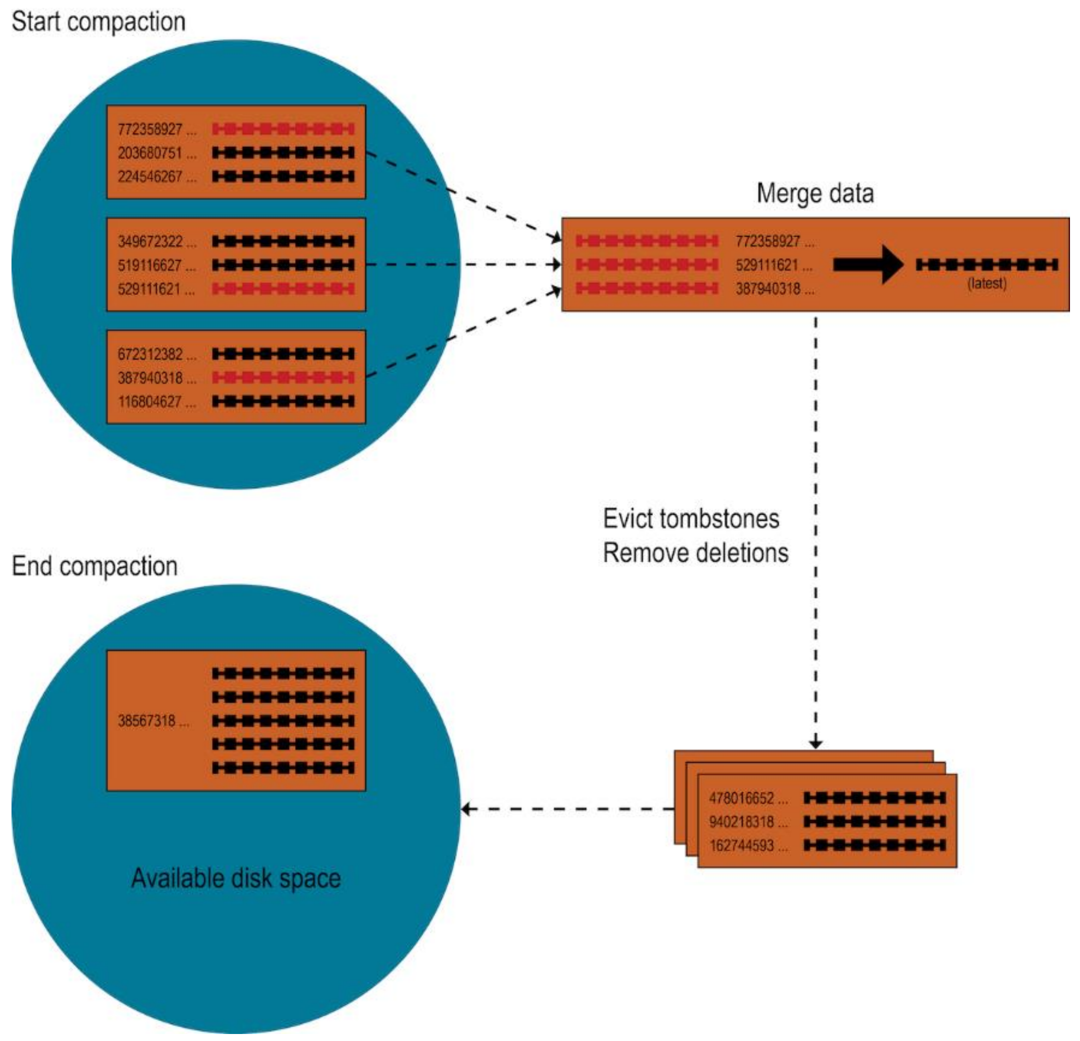
  ⇒ a deleted column is not removed immediately
- A tombstone is written
  - A marker in a row that indicates a column was deleted
  - Signals Cassandra to retry sending a delete request to a replica that was down at the time of delete
- Columns marked with a tombstone exist for a (configurable) grace period
  - Defined per table
  - When expires, the compaction process permanently deletes the column
    - The same process that merges multiple SSTables
- If a node is down longer, the node can possibly miss the delete ⇒ deleted data comes back up again
  - Administrators must run regular node repair

Synchronizes and corrects all replicas

Cassandra
Compaction Process

- Cassandra does not insert/update/delete in place
  - Inserts/updates = new timestamped version of the inserted/updated data in another SSTable
  - Delete = tombstone mark for data
- From time to time compaction has to be done
- Compaction steps:
  1. Merging the data in each SSTable data by partition key
     - Selecting the latest data for storage based on its timestamp
       - We need synchronization!
     - Remember: SSTables are <u>sorted</u> $\rightarrow$ random access is not needed
  2. Evicting tombstones and removing deleted data
  3. Consolidation of SSTables into a single file
  4. Deleting old SSTable files
     - As soon as any pending reads finish using the files

Cassandra
Compaction Process



Cassandra
Compaction Process

- Different strategies (specified per table)
- Simple: trigger compaction when there are more than min_threshold SSTables for a column family
  - SizeTieredCompactionStrategy (default) – creates similar sized SSTables
    - For write-intensive workloads
  - DateTieredCompactionStrategy – stores data written within a certain period of time in the same SSTable
    - For time-series and expiring data
- Complex: LeveledCompactionStrategy
  - Small fixed-sized (5MB by default) SSTables are organized into levels
  - SSTables do not overlap within a level (= immediate compaction)
  - When a level is filled up, another level is created
    - Each new level is 10x larger
  - For read-intensive workloads
    - 90% of all reads are satisfied from a single SSTable
      - Assuming row sizes are nearly uniform
    - In the worst case we read from all levels

Cassandra

Architecture

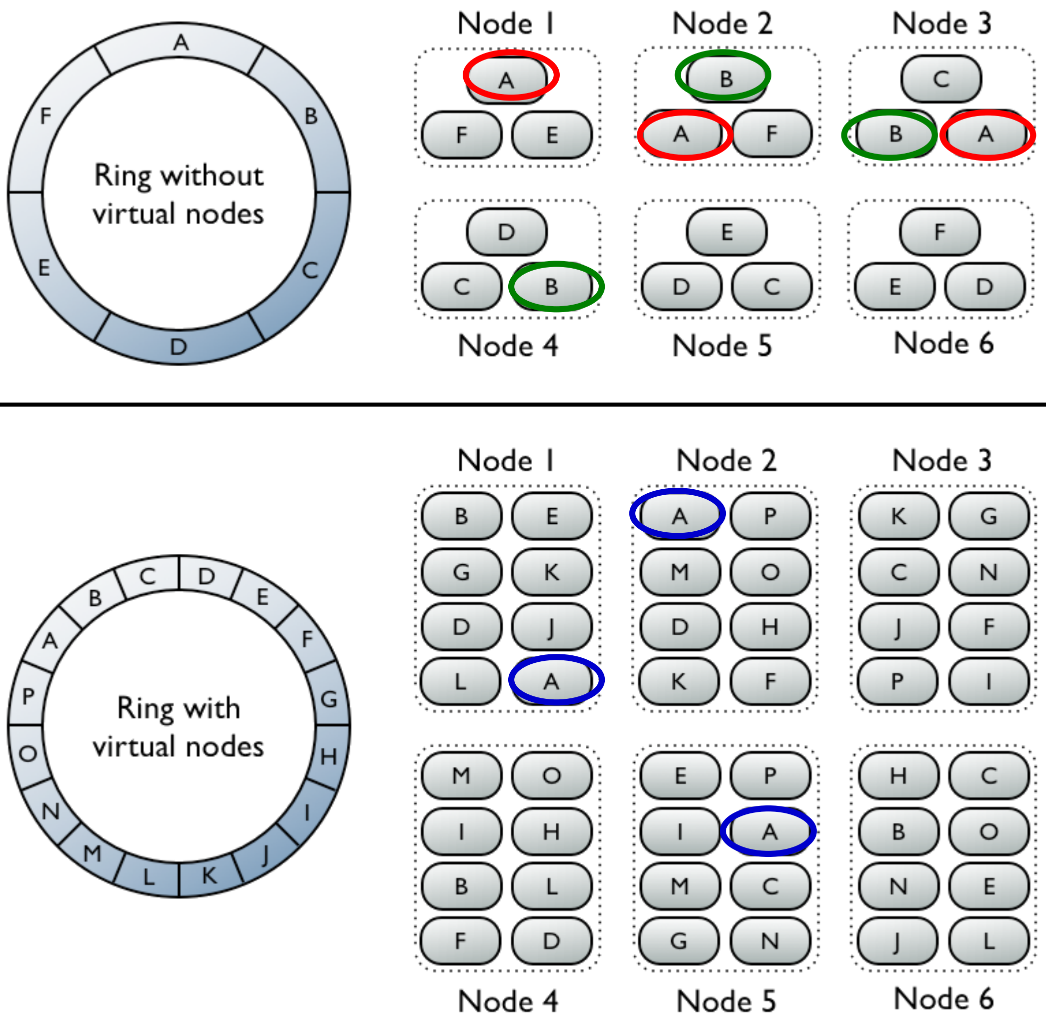
- Peer-to-peer distributed system
  - Assumption: System and hardware failures can and do occur
  - Coordinator = any node responsible for a particular client operation
- Key components:
  - Virtual nodes – assign data ownership to physical nodes
  - Gossip – exchanging information across the cluster
  - Partitioner – determines how to distribute the data across the nodes
  - Replica placement strategy – determines which nodes to place replicas on
- Cluster – stores data partitions of a Cassandra ring

Cassandra

Virtual Nodes

- Allow each node to own a large number of small partition ranges
  - Easier for adding/removing nodes – the small partition ranges are simply transferred
- Still use consistent hashing to distribute data

Example: replication factor = 3



Cassandra

Gossip

- Gossip process
  - Runs every second
  - Exchanges state messages with up to 3 other nodes in the cluster
  - Enables to detect failures
- Gossiped message:
  - Information about a gossiping node + other nodes that it knows about
  - Acquired:
    - Directly = by direct communication
    - Indirectly = second hand, third hand, …
  - Has a version
    - Older information is overwritten with the most current state

Cassandra

Partitioner

- Determines how data is distributed across the nodes
  - Including replicas
- Hash function for computing the token (hash) of a row key
- Types of partitioners:
  - Murmur3Partitioner (default) – uniformly distributes data across the cluster based on MurmurHash hash values
    - Non-cryptographic hash function
    - Values from -2^{63} to $+2^{63}$
  - RandomPartitioner (default for previous versions) – uniformly distributes data across the cluster based on MD5 hash values
    - Values is from 0 to 2^{127} -1
  - ByteOrderedPartitioner – orders rows lexically by key bytes
    - “Hash” = hexadecimal representation of the leading character(s) in key
    - Allows ordered scans by primary key
    - Can have problems with load balancing

Cassandra

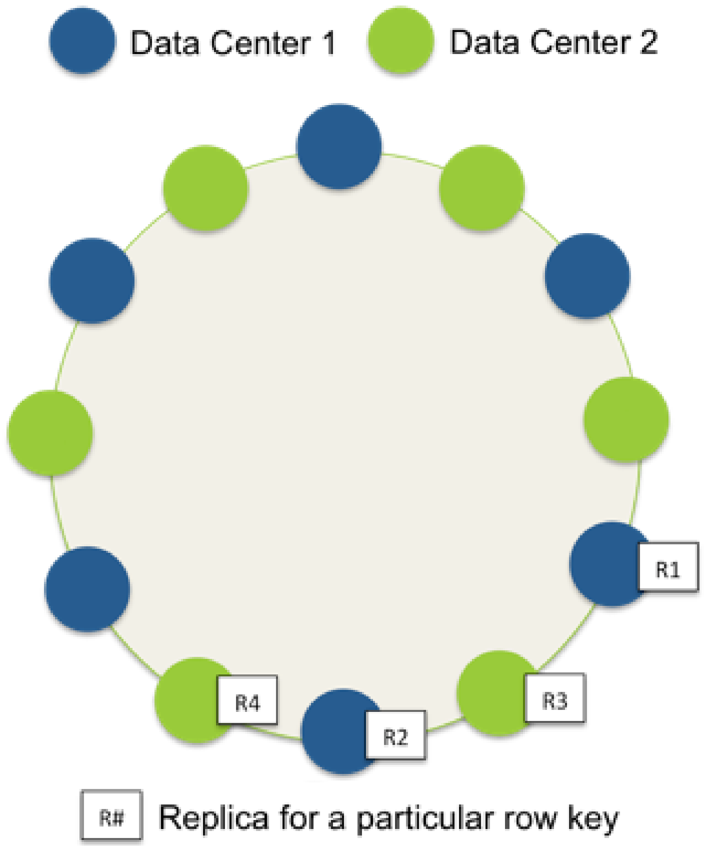
Replication

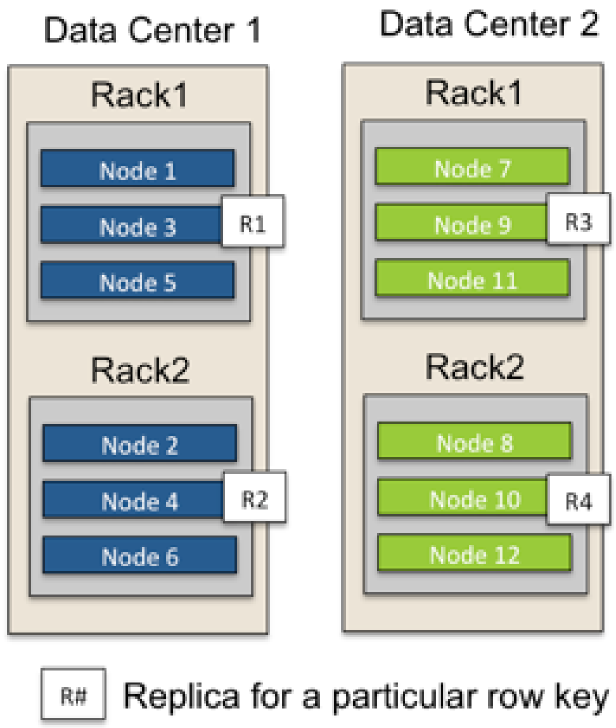
- All replicas are equally important
  - There is no primary or master replica
- When replication factor exceeds the number of nodes, writes are rejected
  - Reads are served as long as the desired consistency level can be met
- Replica placement strategies:
  1. SimpleStrategy
     - Places the first replica on a node determined by the partitioner
     - Additional replicas are placed on the next nodes clockwise in the ring
     - For a single data center only
       - We can divide the nodes into (optional **racks** forming) **data centers**
         - Collection of related nodes, physical or virtual

Cassandra
Replication

2. NetworkTopologyStrategy
   - Places replicas within a data center
     - We set number of replicas per a data center
   1. The first replica is placed according to the partitioner
   2. Additional replicas are placed by walking the ring clockwise until a node in a different rack is found
      - Motivation: nodes in the same rack often fail at the same
        - e.g., power, cooling, or network issue
   3. If no such node exists, additional replicas are placed in different nodes in the same rack

Cassandra
Replication – Examples





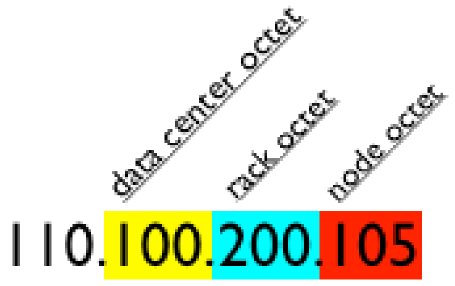
Replicas assigned to different racks

Cassandra
Replication

- How many replicas to configure in each data center?
  - Compromise between:
    1. Need for being able to satisfy reads locally
       - Without cross data-center latency
    2. Failure scenarios
  - Most commonly: 2-3 replicas in each data center
  - Can be asymmetric (= different replication factors for different data centers)

Cassandra

Replication – Snitch



- Informs about the network topology
  - Determines which data centers and racks are written to and read from
- All nodes must have exactly the same snitch configuration
- Various types:
  - SimpleSnitch – does not recognize data centers/racks
  - RackInferringSnitch – racks and data centers are assumed to correspond to the 3rd and 2nd octet of the node's IP address
  - PropertyFileSnitch – uses a user-defined description of the network
  - Dynamic snitching – monitors performance of reads, chooses the best replica based on this history
    - Special case: optimization of read requests
  - …

References

- Eric Redmond – Jim R. Wilson: **Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement**
- Pramod J. Sadalage – Martin Fowler: **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**
- Cassandra:
  - Getting Started: https://cassandra.apache.org/doc/latest/getting_started/
  - CQL: https://cassandra.apache.org/doc/latest/cql/index.html