Data modelling with Apache Cassandra

Andrei Arion, LesFurets.com, tp-bigdata@lesfurets.com

Plan

- **Cassandra Query Language (CQL) by examples**
- Data modeling with Cassandra
- TP2: data modeling with Apache Cassandra

Recall: partitioner

- **hash function** that derives a token from the primary key of a row
- determines which node will receive the ***first replica***
- RandomPartitioner, **Murmur3Partitioner**, ByteOrdered

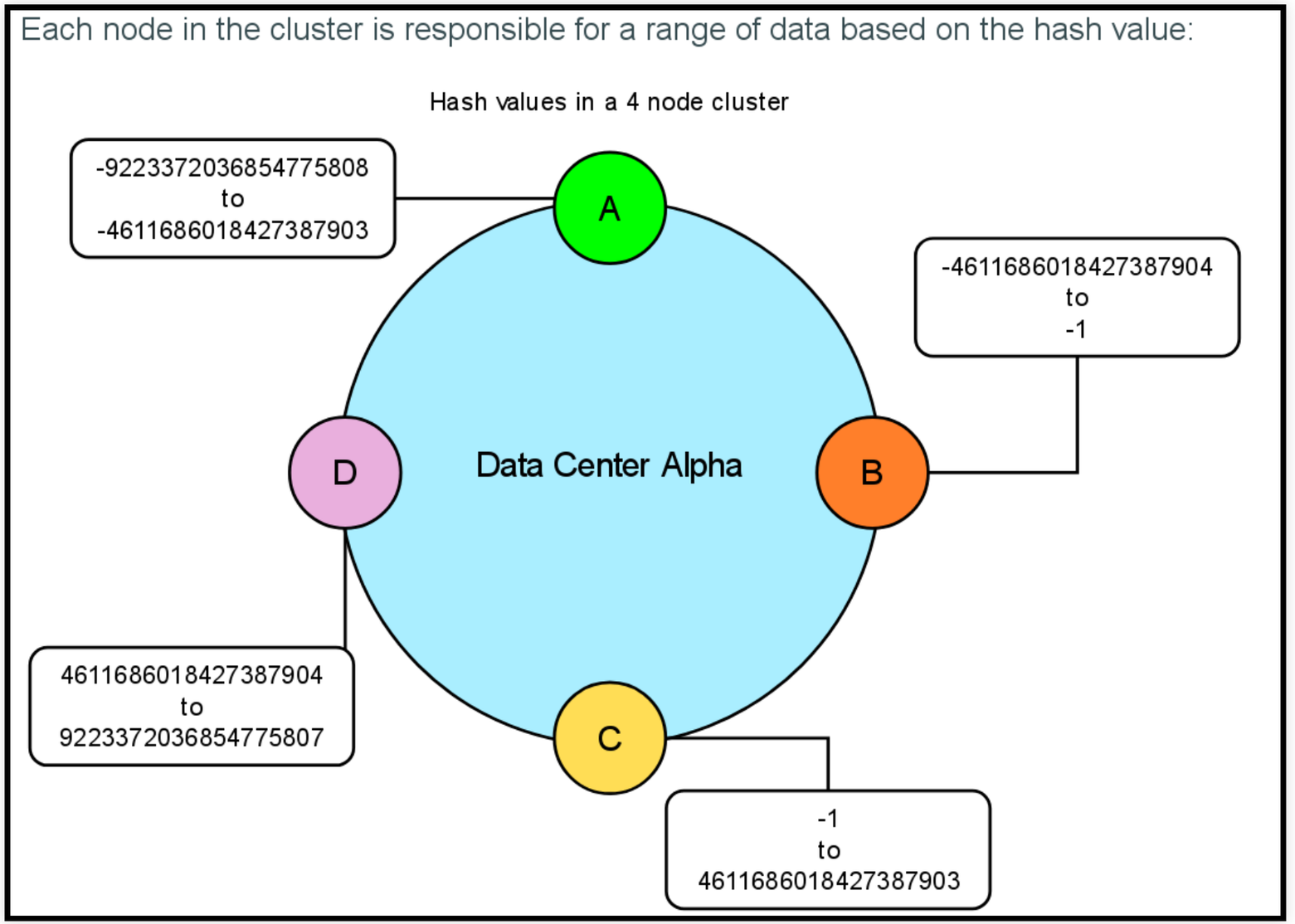
Recall: Murmur3Partitionner

name	age	car	gender
jim	36	camaro	M
carol	37	bmw	F
johnny	12		M
suzy	10		F

Cassandra assigns a hash value to each partition key:

Partition key	Murmur3 hash value
jim	-2245462676723223822
carol	7723358927203680754
johnny	-6723372854036780875
suzy	1168604627387940318

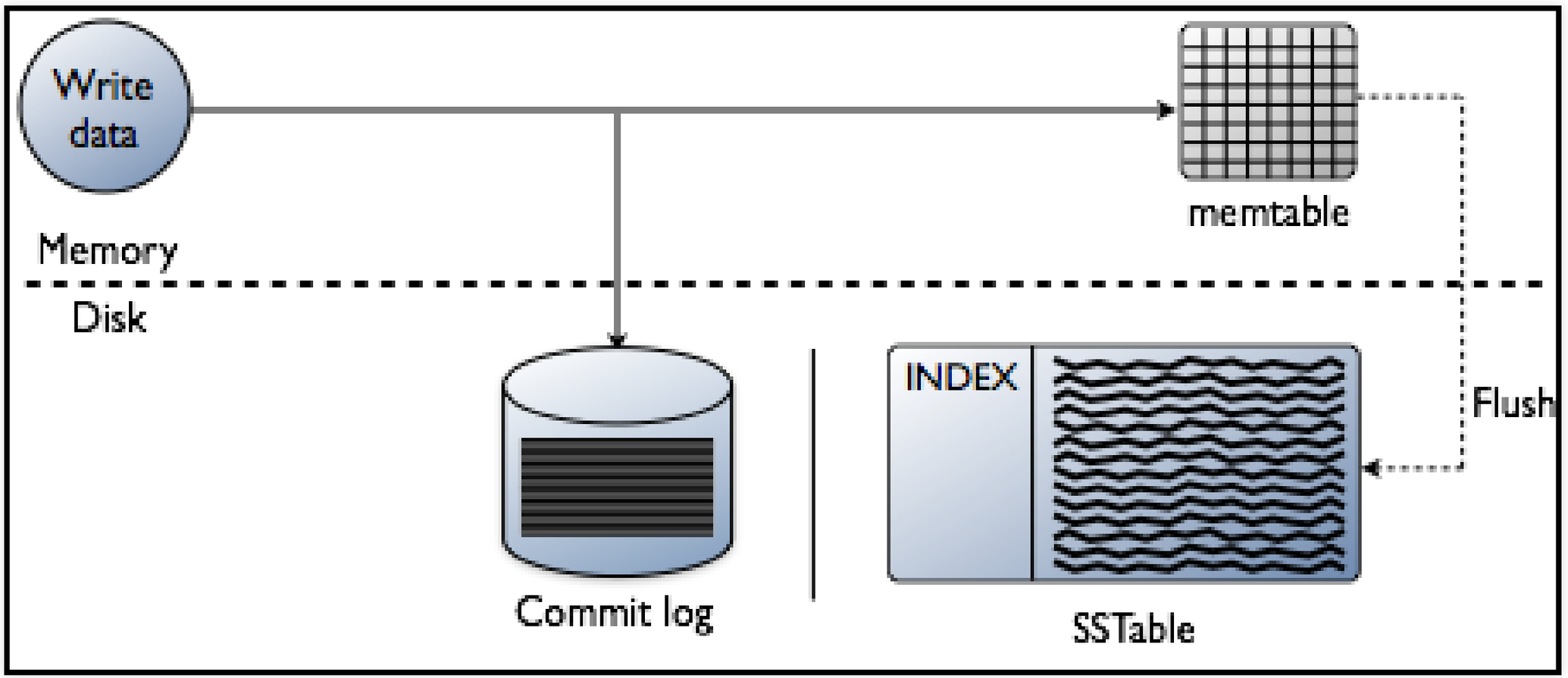
Recall: Consistent Hashing: mapping



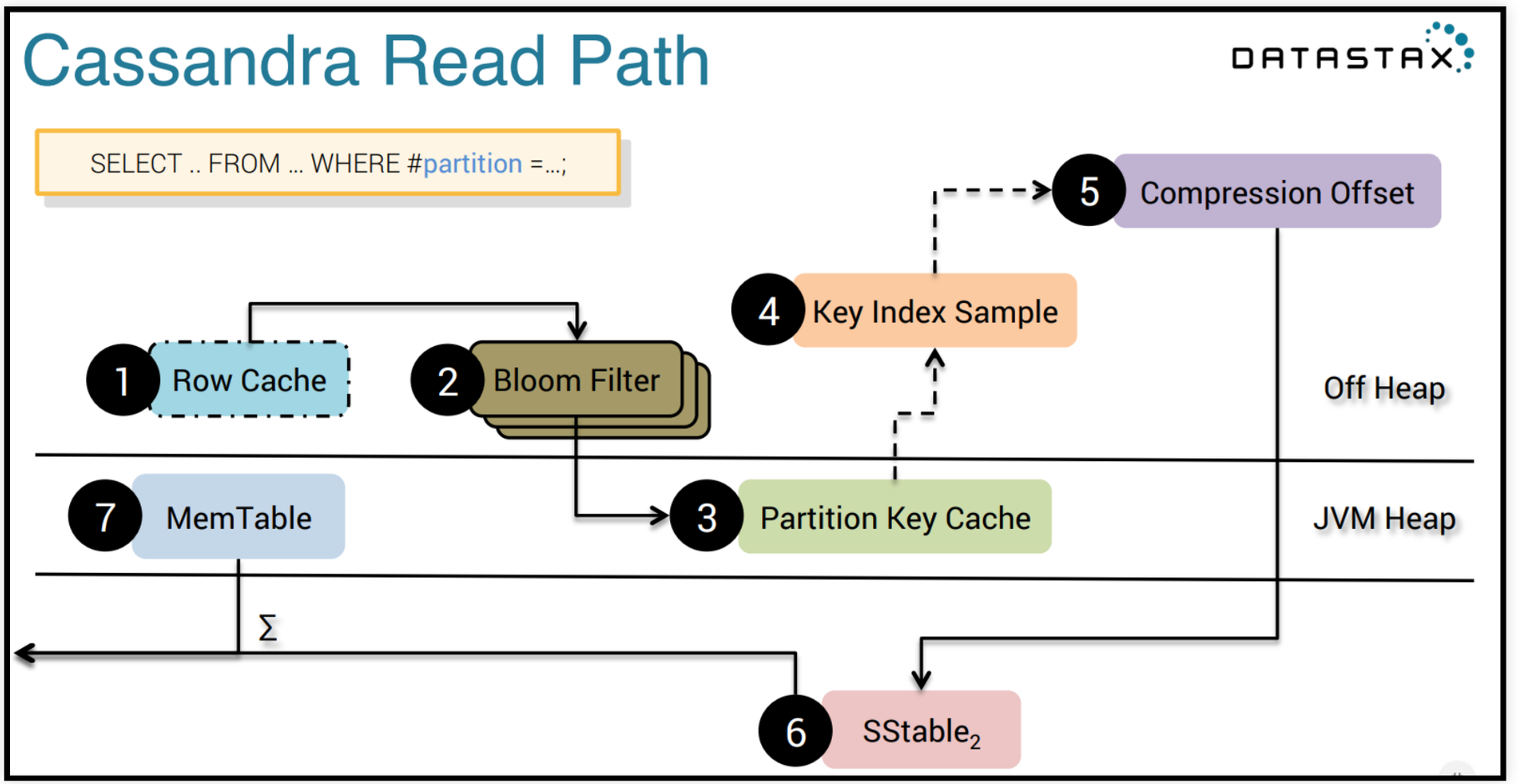
Recall: Consistent Hashing: mapping

Node	Start range	End range	Partition key	Hash value
A	-9223372036854775808	-4611686018427387903	johnny	-672337285403678075
B	-4611686018427387904	-1	jim	-22454626767232238 22
C	0	4611686018427387903	suzy	1168604627387940318
D	4611686018427387904	9223372036854775807	carol	7723358927203680754

Write path

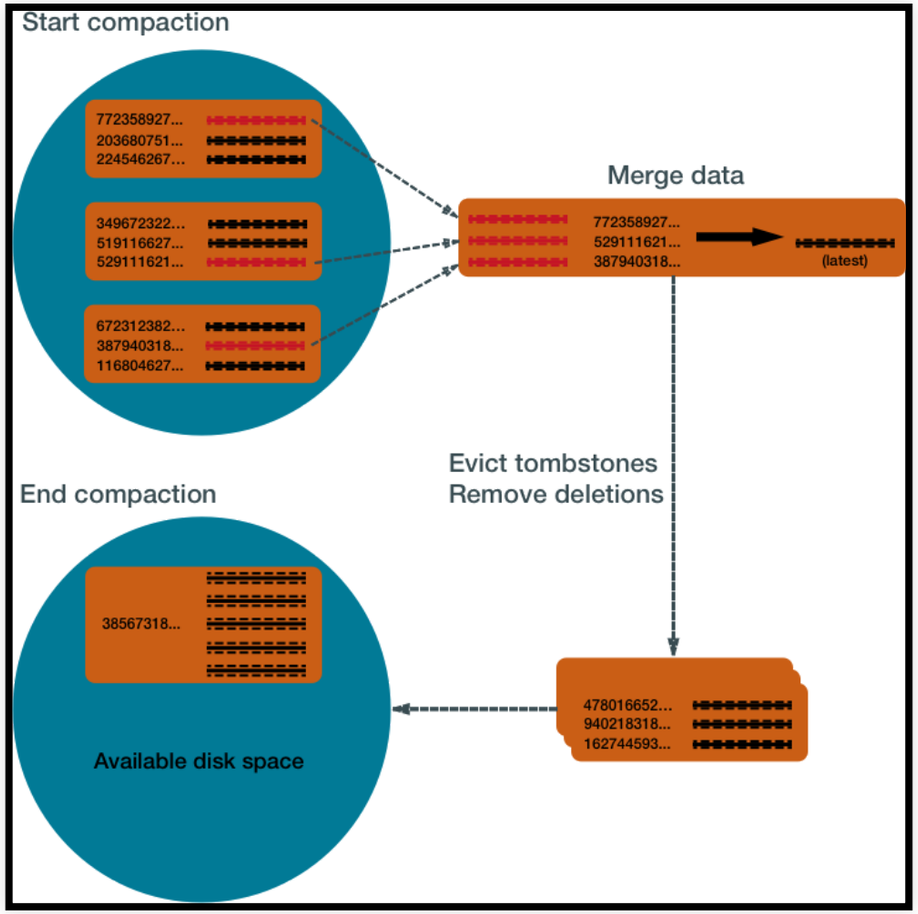

Write data
memtable
Memory
Disk
Flush
INDEX
Commit log
SSTable

Read path


Cassandra Read Path
DATASTAX
SELECT .. FROM ... WHERE #partition =...;
1 Row Cache
2 Bloom Filter
3 Partition Key Cache
4 Key Index Sample
5 Compression Offset
6 SStable2
7 MemTable
Σ
Off Heap
JVM Heap

Cassandra compactions

- collects all versions of each unique row
- assembles one complete row (up-to-date)


Start compaction
772358927...
203680751...
224546267...
349672322...
519116627...
529111621...
672312382...
387940318...
116804627...
Merge data
772358927...
529111621...
387940318...
(latest)
Evict tombstones
Remove deletions
End compaction
38567318...
Available disk space
478016652...
940218318...
162744593...

Cassandra versions

- *Latest version 3.11.5*
- ***Cassandra 3.0*** is supported until 6 months after 4.0 release (date TBD)
- Cassandra 2.2 is supported until 4.0 release
- Cassandra 2.1 is supported until 4.0 release

Cassandra Query Language

- CQL = SQL without:
  - Joins
  - ACID
  - Integrity constraints
  - Subqueries
  - Auto-increment columns
- Data model ⇒ query restrictions

Keyspace

- similar to a relational database schema

```
CREATE KEYSPACE movies
WITH replication = {
    'class' : 'SimpleStrategy',
    'replication_factor' : 2
};

DROP KEYSPACE movies;

USE movies
```

Modifying a keyspace

- altering a keyspace (eg. modify replication factor)

```
ALTER KEYSPACE movies
WITH REPLICATION =  {
     'class' : 'SimpleStrategy',
     'replication_factor' : 3
};
```

- needs a full repair on the keyspace to redistribute the data !

```
nodetool repair --full keyspace_name
```

CQL table

```
CREATE TABLE my_table (
   col1 int,
   col2 text,
   col3 int,
   col4 int,
   col5 int [STATIC],
   PRIMARY KEY ( (col1,col2),col3,col4))  1
   [WITH CLUSTERING ORDER BY (col3 DESC, col4 ASC)];
```

- *(col1,col2)* = (*composite*) **partition key**, first element of the PRIMARY KEY
  - **mandatory**, composed by one ore more columns
  - uniquely identifies a partition (group of columns that are stored/replicated together)
  - **hash function is applied to *col1:col2* to determine on which node to store the partition**

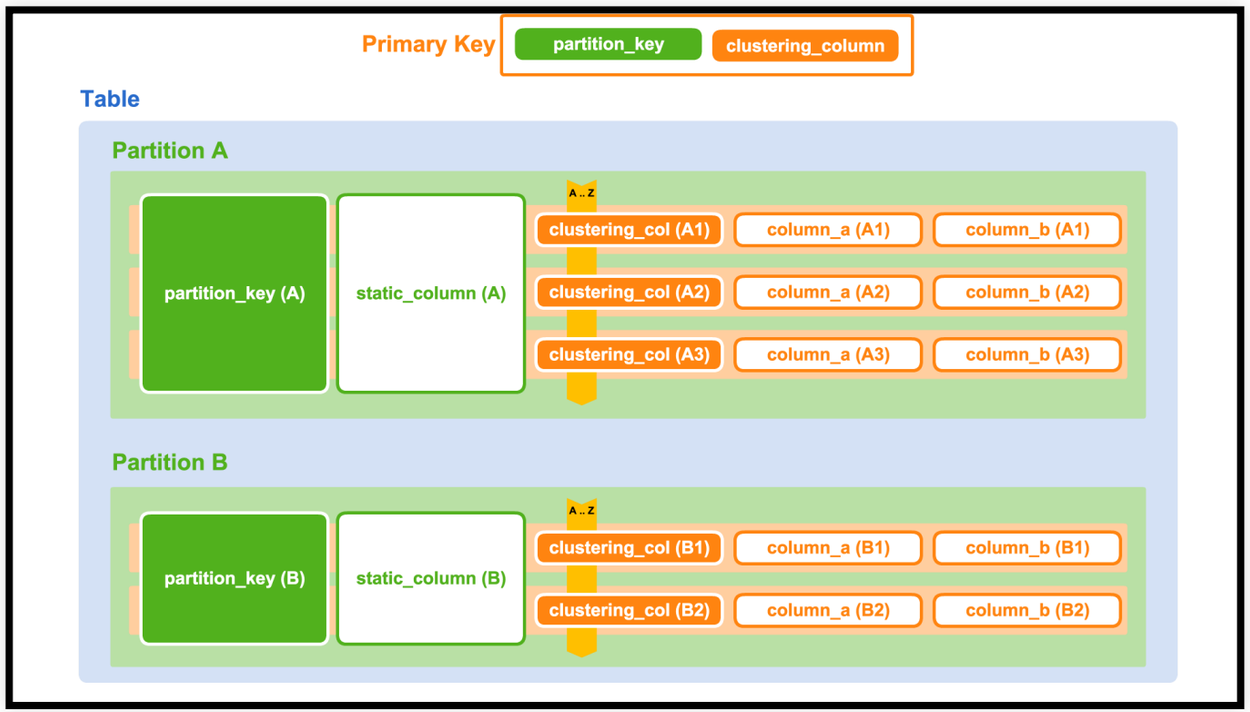
CQL table

```
CREATE TABLE my_table (
   col1 int,
   col2 text,
   col3 int,
   col4 int,
   col5 int [STATIC],   3
   PRIMARY KEY ( (col1,col2),col3,col4))   2
   [WITH CLUSTERING ORDER BY (col3 DESC, col4 ASC)];
```

- *col3*, *col4* ⇒ **clustering columns**
  - specify the order in a single partition
- *col5* : **static column**, stored once per partition
  - no clustering columns ⇒ all columns behave like static columns !

Cassandra 3.X physical model

- ***Map<PartitionKey, SortedMap<Clustering, Row>>***



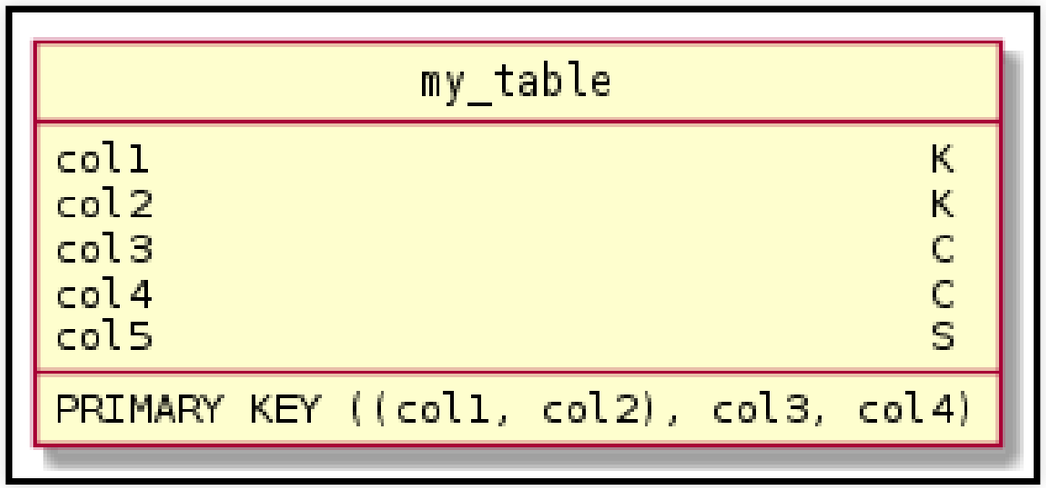
- **Row** = List<Columns>
- **Column**/Cell: **Name**, *Value* (optional), **Timestamp**, *TTL*(optional)

Cassandra physical model properties

- groups together related data in the same partition
- fast look-up by **partition key**
- efficient scans and slices by clustering columns.

Table: logical view notation

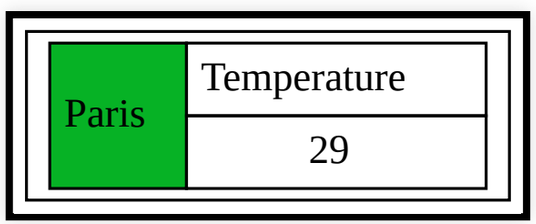
- high level view
- no data-types details



Simple primary key

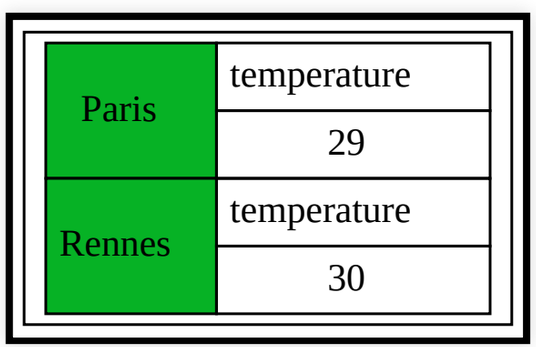
```
CREATE TABLE temperature (
    ville text,
    temperature int,
    PRIMARY KEY (ville)
);
INSERT INTO temperature (ville, temperature ) VALUES ( 'Paris', 30);
INSERT INTO temperature (ville, temperature ) VALUES ( 'Paris', 29);

SELECT * FROM temperature;
 ville  | temperature
--------+-------------
  Paris |          29
```

- temperature column behaves like a static column
- upsert



Simple primary key

```
CREATE TABLE temperature (
    ville text,
    temperature int,
    PRIMARY KEY (ville)
);

INSERT INTO temperature (ville, temperature ) VALUES ( 'Rennes', 30);

SELECT * FROM temperature;

 ville  | temperature
--------+-------------
  Paris |          29
 Rennes |          30
```

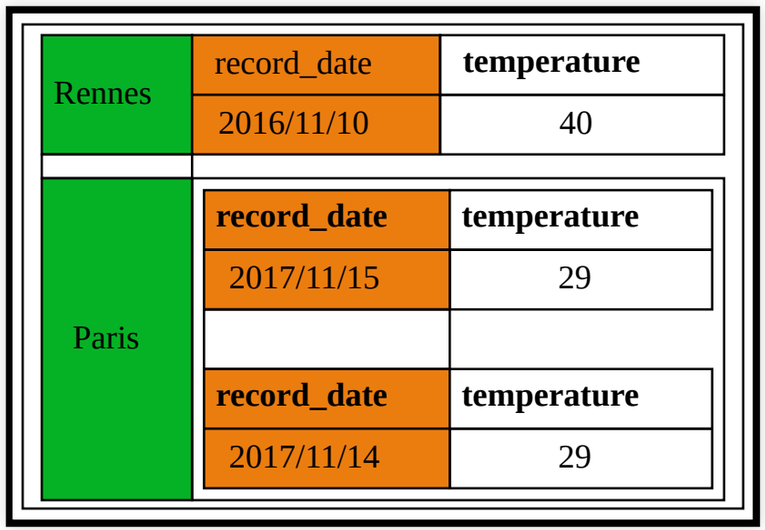


Simple primary key (Disk content)

```
> tools/bin/sstabledump mc-1-big-Data.db
{
  [
    { "partition" : {
      "key" : [ "Paris" ],
      "position" : 0
    },
    "rows" : [
      {
        "type" : "row",
        "position" : 30,
        "liveness_info" : { "tstamp" : "2017-11-14T15:34:40.012400Z" },
        "cells" : [
          { "name" : "temperature", "value" : 29 }
        ]
      }
    ]
  },
```

PK = Partition key + clustering column(s)

```
CREATE TABLE temperature_date (
    ville text,
    record_date text,
    temperature int,
    humidity int,
    PRIMARY KEY (ville, record_date)
) WITH CLUSTERING ORDER BY (record_date DESC) ;

INSERT INTO temperature_date (ville, record_date, temperature, ) VALUES ( 'Paris', '2017/11/14', 30);
INSERT INTO temperature_date (ville, record_date, temperature ) VALUES ( 'Paris', '2017/11/14', 29);
INSERT INTO temperature_date (ville, record_date, temperature ) VALUES ( 'Rennes', '2016/11/10', 40);
INSERT INTO temperature_date (ville, record_date, temperature ) VALUES ( 'Paris', '2017/11/15', 29);
```



Primary key + clustering column

```
cqlsh:temperature> SELECT * FROM temperature_date;

 ville  | record_date | temperature
--------+-------------+-------------
  Paris |  2017/11/15 |          29
  Paris |  2017/11/14 |          29
 Rennes |  2016/11/10 |          40



 cqlsh:temperature> SELECT * FROM temperature_date WHERE ville = 'Paris' LIMIT 2;

  ville | record_date | temperature
 -------+-------------+-------------
  Paris |  2017/11/15 |          29
  Paris |  2017/11/14 |          29

 (2 rows)
```

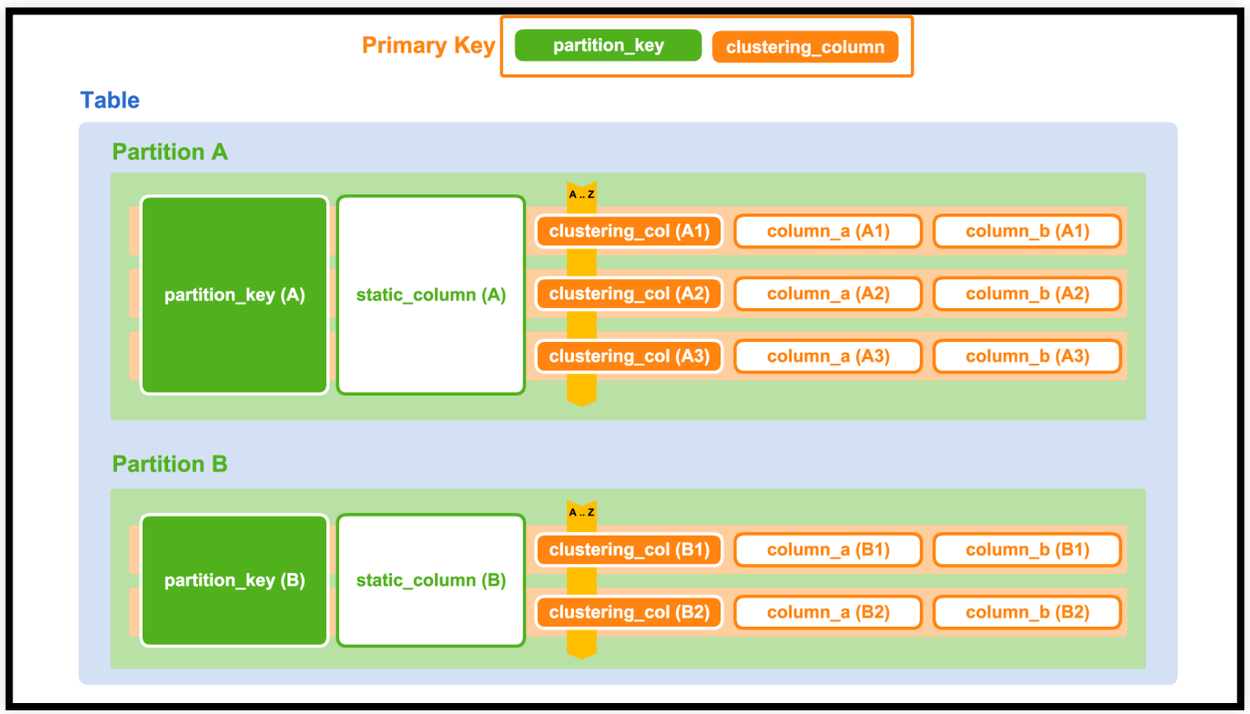
Static columns

```
CREATE TABLE teammember_by_team (
  teamname text,
  manager text static,
  location text static,
  membername text,
  nationality text,
  position text,
  PRIMARY KEY ((teamname), membername)
);
```

- stored once per partition
- model the **one side** of a *one-to-many relation*

Cassandra 3.X physical model

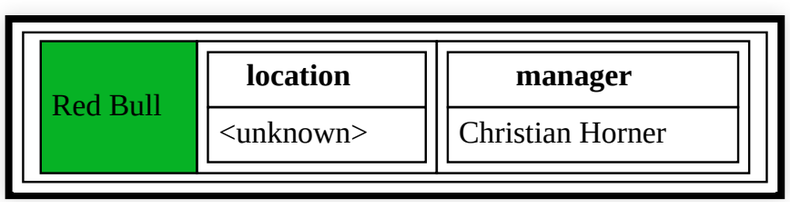
- ***Map<PartitionKey, SortedMap<Clustering, Row>>***



- **Row** = List<Columns>, **Column**/Cell: **Name**, *Value* (optional), **Timestamp**, *TTL*(optional)

Static columns

```
CREATE TABLE teammember_by_team (
  teamname text,
  manager text static,
  location text static,
  membername text,
  nationality text,
  position text,
  PRIMARY KEY ((teamname), membername)
);

INSERT INTO teammember_by_team (teamname, manager, location)
VALUES ('Red Bull', 'Christian Horner', '<unknown>');

teamname  | membername | location | manager          | nationality | position
----------+------------+----------+------------------+-------------+----------
 Red Bull |       null | <unkown> | Christian Horner |        null |     null
```



Static column storage

```
> tools/bin/sstabledump mc-1-big-Data.db
[
  {
    "partition" : {
      "key" : [ "Red Bull" ],
      "position" : 0
    },
    "rows" : [
      {
        "type" : "static_block",
        "position" : 64,
        "cells" : [
          { "name" : "location", "value" : "<unknown>", "tstamp" : "2018-11-26T16:57:37.374405Z" },
          { "name" : "manager", "value" : "Christian Horner", "tstamp" : "2018-11-26T16:57:37.374405Z" }
        ]
      }
    ]
  }
```

Clustering columns

```
CREATE TABLE teammember_by_team (
  teamname text,
  manager text static,
  location text static,
  membername text,
  nationality text,
  position text,
  PRIMARY KEY ((teamname), membername)
);
INSERT INTO teammember_by_team (teamname, membername, nationality, position)
                      VALUES ('Red Bull', 'Ricciardo', 'Australian', 'driver');


select * from teammember_by_team;
 teamname | membername | location  | manager           | nationality | position
----------+------------+-----------+-------------------+-------------+----------
 Red Bull |  Ricciardo | <unknown> | Christian Horner |  Australian |   driver
```

Clustering columns storage

```
> tools/bin/sstabledump mc-2-big-Data.db
[
  {
    "partition" : {
      "key" : [ "Red Bull" ],
      "position" : 0
    },
    "rows" : [
      {
        "type" : "row",
        "position" : 57,
        "clustering" : [ "Ricciardo" ],
        "liveness_info" : { "tstamp" : "2018-11-26T17:06:36.807957Z" },
        "cells" : [
          { "name" : "nationality", "value" : "Australian" },
          { "name" : "position", "value" : "driver" }
        ]
      }
```

Static and clustering after compaction

```
>tools/bin/sstabledump mc-3-big-Data.db
[
  {
    "partition" : {
      "key" : [ "Red Bull" ],
      "position" : 0
    },
    "rows" : [
      {
        "type" : "static_block",
        "position" : 96,
        "cells" : [
          { "name" : "location", "value" : "<unknown>", "tstamp" : "2018-11-26T16:57:37.374405Z" },
          { "name" : "manager", "value" : "Christian Horner", "tstamp" : "2018-11-26T16:57:37.374405Z" }
        ]
      },
      {
        "type" : "row",
```

Clustering columns

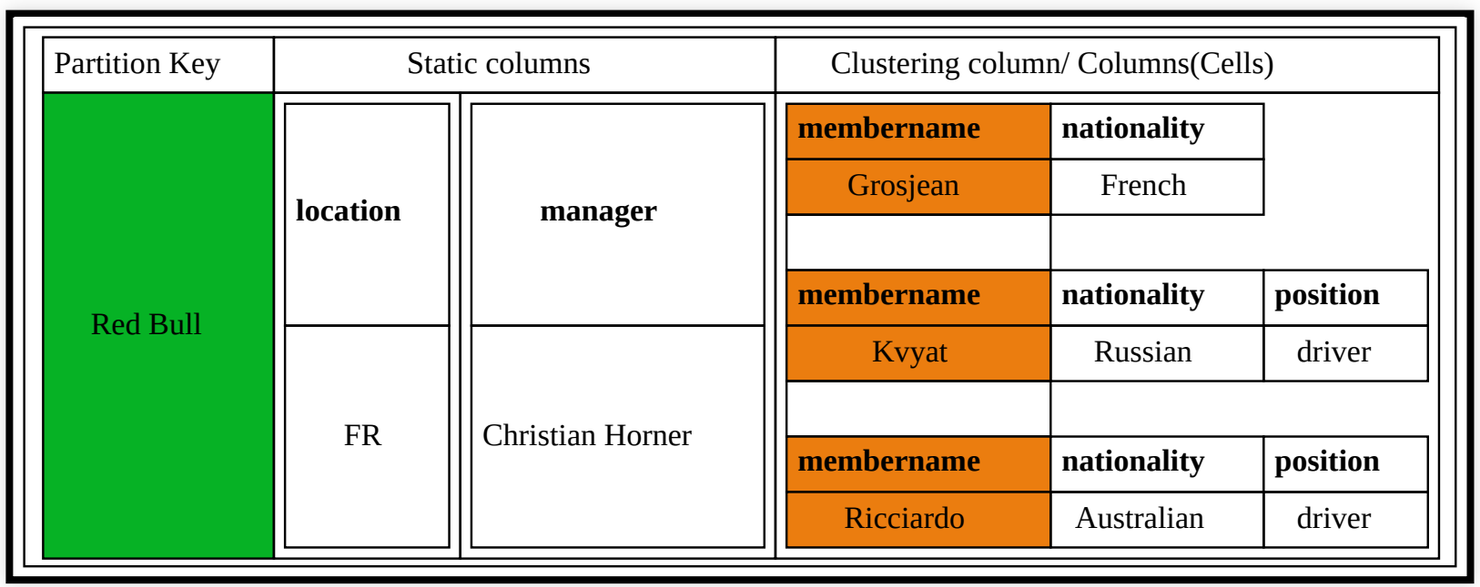
```
CREATE TABLE teammember_by_team (
  teamname text,
  manager text static,
  location text static,
  membername text, nationality text, position text,
  PRIMARY KEY ((teamname), membername)
);
INSERT INTO teammember_by_team (teamname, membername, nationality, position) VALUES ('Red Bull', 'Ricciard
INSERT INTO teammember_by_team (teamname, membername, nationality, position) VALUES ('Red Bull', 'Kvyat',
```

```
 teamname | membername | location | manager           | nationality | position
----------+------------+----------+-------------------+-------------+---------
 Red Bull |      Kvyat |<location>| Christian Horner |     Russian |   driver
 Red Bull |  Ricciardo |<location>| Christian Horner |  Australian |   driver
```

Partition Key	Static columns		Clustering column/ Columns(Cells)		
Red Bull	**location**	**manager**	**membername**	**nationality**	**position**
			Kvyat	Russian	driver
	<unknown>	Christian Horner	**membername**	**nationality**	**position**
			Ricciardo	Australian	driver

Static columns + clustering columns

```
INSERT INTO teammember_by_team (teamname, membername, nationality, location) VALUES ('Red Bull', 'Grosjean

 teamname | membername | location | manager          | nationality | position
----------+------------+----------+------------------+-------------+----------
 Red Bull |   Grosjean |       FR | Christian Horner |      French |     null
 Red Bull |      Kvyat |       FR | Christian Horner |     Russian |   driver
 Red Bull |  Ricciardo |       FR | Christian Horner |  Australian |   driver
```



Static columns + clustering columns: physical storage


```
{
  "partition" : {
    "key" : [ "Red Bull" ],
    "position" : 0
  },
  "rows" : [
    {
      "type" : "static_block",
      "position" : 87,
      "cells" : [
        { "name" : "location", "value" : "FR", "tstamp" : "2017-11-15T01:53:29.914293Z" },
        { "name" : "manager", "value" : "Christian Horner", "tstamp" : "2017-11-14T20:04:07.780008Z" }
      ]
    },
    {
      "type" : "row",
      "position" : 87,
      "clustering" : [ "Grosjean" ],
```

Column/Cell

- Data Model:
  - Name - mandatory
  - Value - optional
- Cassandra bookkeeping
  - Timestamp
  - TTL
  - Tombstone flag

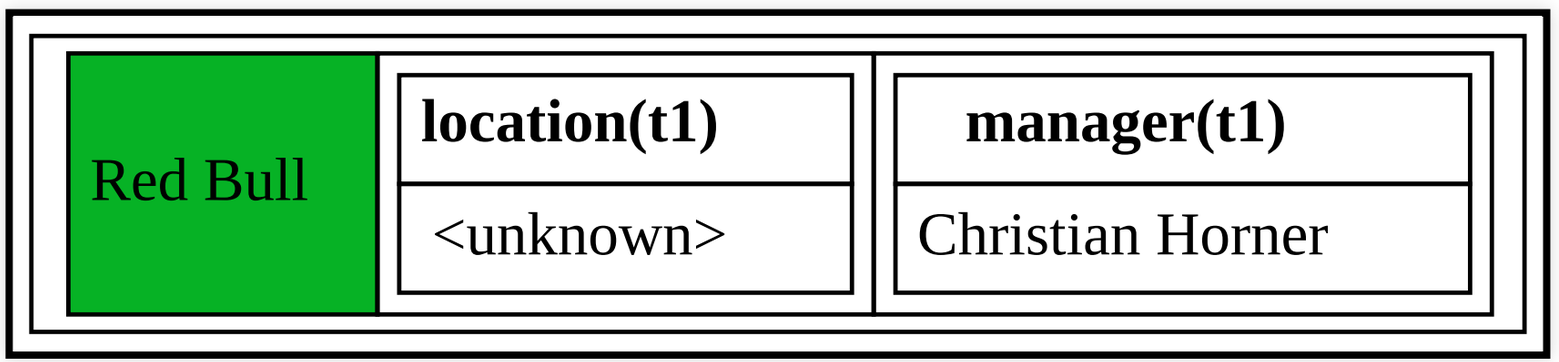
Timestamp

- date of last update, auto generated or user provided

```
Node1: INSERT INTO teammember_by_team (teamname, manager, location)
VALUES ('Red Bull', 'Christian Horner', '<unknown>') USING TIMESTAMP;
```

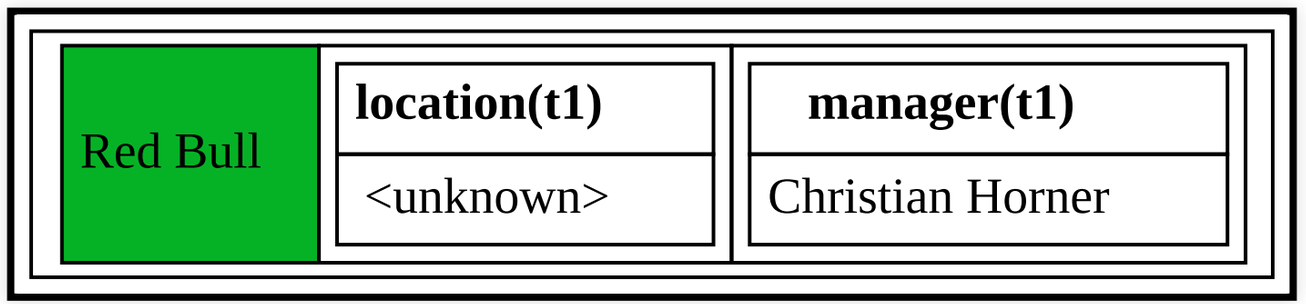
Timestamp

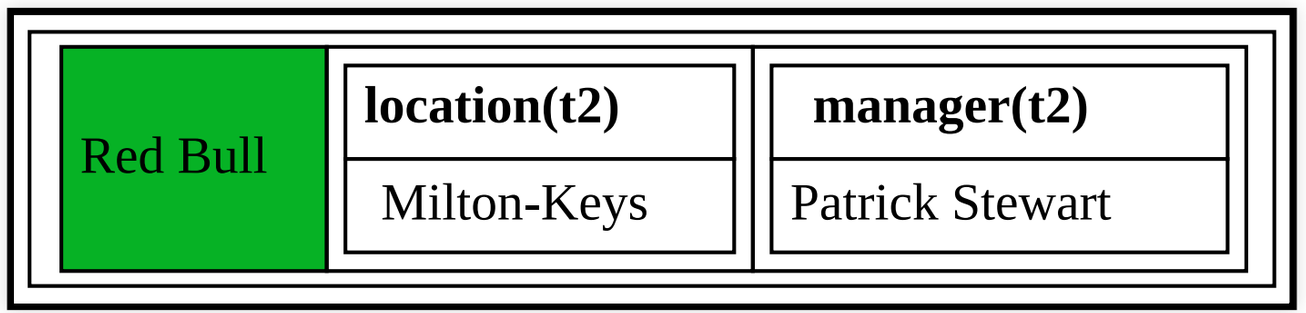
```
Node1: INSERT INTO teammember_by_team (teamname, manager, location)
VALUES ('Red Bull', 'Christian Horner', '<unknown>');
```



Last Write Win (LWW)

```
Node1: INSERT INTO teammember_by_team (teamname, manager, location)
VALUES ('Red Bull', 'Christian Horner', '<unknown>');


Node2: INSERT INTO teammember_by_team (teamname, manager, location)
VALUES ('Red Bull', 'Patrick Stewart', 'Milton-Keys');
```

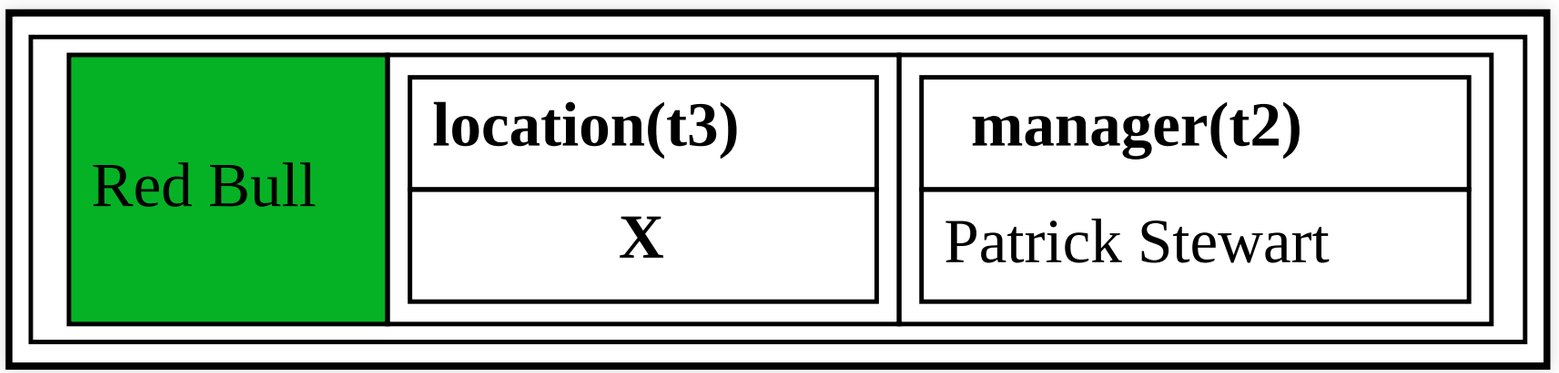




Tombstone

- Delete
  - mark cell as to be deleted = tombstone

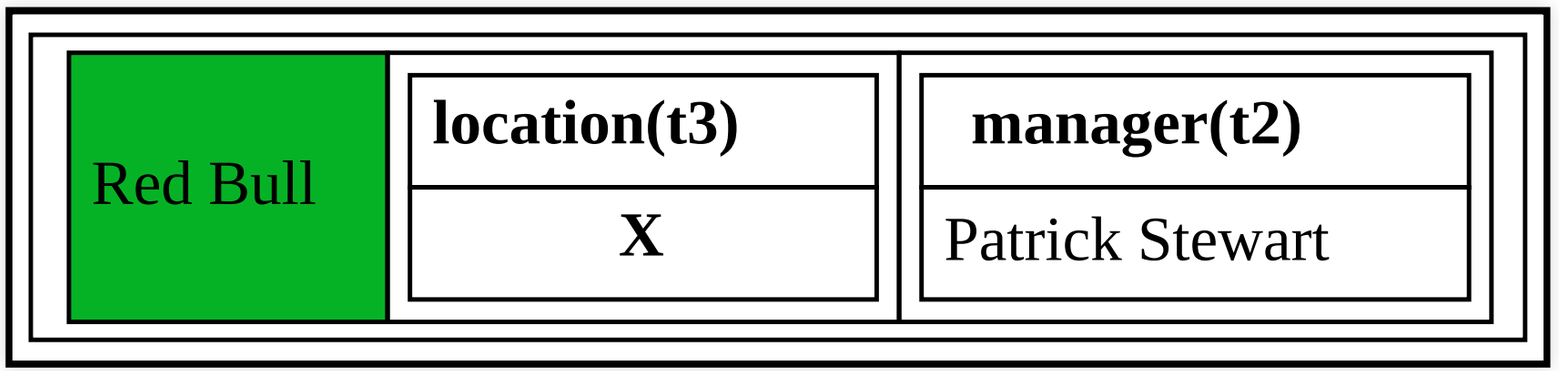
```
Node3: DELETE location FROM teammember_by_team WHERE teamname='Red Bull'
```



Tombstone storage (after compaction)


```
[
  {
    "partition" : {
      "key" : [ "Red Bull" ],
      "position" : 0
    },
    "rows" : [
      {
        "type" : "static_block",
        "position" : 91,
        "cells" : [
          { "name" : "location", "deletion_info" : { "local_delete_time" : "2018-11-26T17:21:43Z" },
            "tstamp" : "2018-11-26T17:21:43.283089Z"
          },
          { "name" : "manager", "value" : "Christian Horner", "tstamp" : "2018-11-26T16:57:37.374405Z" }
        ]
      },
      {
```

Last Write Win (LWW)



- consistency mechanisms ensure that the last value is propagated during repairs
- tombstones are cleaned on compaction

Zombie columns

- if a node is down during a DELETE
  - and gets back after the *hinted-hand-off window*
  - and after a compaction was done on the table after a *gc_grace_period* (10 days)
  - it will bring back the deleted data

Column restrictions

- Columns in a partition: 2B (2^31); single column value size: 2 GB (1 MB is recommended)
- Clustering column value, length of: 65535 (2^16-1)
- Key length: 65535 (2^16-1)
- Query parameters in a query: 65535 (2^16-1)
- Statements in a batch: 65535 (2^16-1)
- collection size: 2B (2^31); values size: 65535 (2^16-1)
- Blob size: 2 GB (less than 1 MB is recommended)

Datatypes

- basic types: int, text, varchar, date
- **NO AUTOINCREMENT** values ⇒ IDs??
- IDs ⇒ global identifiers
  - **uuid** (Universally Unique Identifier)
    - uuid() ⇒ adbad1fd-9947-4645-bfbe-b13eeacced47
  - **timeuuid** (Timed Universally Unique Identifier)
    - now() ⇒ fab5d1d0-c76a-11e7-b622-151d52dfc7bc
    - now() ⇒ 0431cc50-c76b-11e7-b622-151d52dfc7bc
- collections ⇒ set/map/list with JSON like syntax
- UDTs

Inserting data

```
INSERT INTO [keyspace_name.] table_name (column_list)
VALUES (column_values)
[IF NOT EXISTS]
[USING TTL seconds | TIMESTAMP epoch_in_microseconds]
```

- IF NOT EXISTS ⇒
  - inserts if no rows match the PRIMARY KEY
  - *lightweight transactions*
- USING TTL ⇒ automatic expiring data, will create a **tombstone** once expired
- TIMESTAMP
  - can be in the future ⇒ the insert will *"appear"* at TIMESTAMP

Inserts and updates

- Insert/update/delete operations on rows sharing the same partition key are performed **atomically** and in **isolation**.
- collections (list, set, map)

Inserting data

```
CREATE TABLE ratings_by_user (
    user_id text,
    movie_id text,
    name text,
    rating int,
    ts int,
    PRIMARY KEY (user_id, movie_id)
)

INSERT INTO ratings_by_user (user_id , movie_id , name , rating , ts )
VALUES ( 'uuid1','uuid2','Starwars',4,3); -- OK

INSERT INTO ratings_by_user (user_id, movie_id) VALUES ('2323..','2442..'); -- OK !!!


INSERT INTO ratings_by_user (user_id) VALUES ('2323..'); -- KO !!!
```

Inserting data - sets

```
INSERT INTO cyclist_career_teams (id,lastname,teams)
    VALUES (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS',
  { 'Rabobank-Liv Woman Cycling Team','Rabobank-Liv Giant','Rabobank Women Team','Nederland bloeit' } ); 1

-- ADD ELEMENT
UPDATE cyclist_career_teams
    SET teams = teams + {'Team DSB - Ballast Nedam'} WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2; 2

-- REMOVE ELEMENT
UPDATE cyclist_career_teams
    SET teams = teams - {'WOMBATS - Womens Mountain Bike & Tea Society'} WHERE id = 5b6962dd-3f90-4c93-8f6

-- DELETE
UPDATE cyclist.cyclist_career_teams SET teams = {} WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2; 4

DELETE teams FROM cycling.cyclist_career_teams WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2; 5
```

Inserting data - lists

```
INSERT INTO upcoming_calendar (year, month, events)
VALUES (2015, 06, ['Criterium du Dauphine','Tour de Suisse']);

-- ADD ELEMENT
UPDATE upcoming_calendar
SET events = ['The Parx Casino Philly Cycling Classic'] + events WHERE year = 2015 AND month = 06;


-- ADD ELEMENT + SHIFT
 UPDATE upcoming_calendar SET events[2] = 'Vuelta Ciclista a Venezuela' WHERE year = 2015 AND month = 06;
```

- **read-before-write** semantic for some operations ⇒ can be costly, prefer sets

Bulk insert - COPY

```
COPY table1(col1, col2,...) FROM 'users.csv';

COPY table1(col1, col2,...) FROM 'users.csv'
WITH HEADER=true;
```

Bulk insert - batches

```
BEGIN [UNLOGGED | LOGGED] BATCH
[USING TIMESTAMP [epoch_microseconds]]
   dml_statement [USING TIMESTAMP [epoch_microseconds]];
   [dml_statement; ...]
APPLY BATCH;
```

Bulk insert - batches

- ensure **atomicity**(all or nothing) and **isolation** for same partition
- ensure **atomicity** for multi-partition
- needs coordination !
  - single partition
  - multiple partition inserts (via replicated batchlog)
- ! maximum size of a single operation (**max_mutation_size_in_kb**)
- !! do not use for many partitions

Bulk insert - client code

- parellize inserts in application code
  - usually via a distributed framework
    - that is aware of the data palacement (***token aware***)
    - use ***prepared statements*** and ***batches***

Counters

- special column for storing a number that is changed in increments
- atomic update

```
CREATE TABLE popular_count (
  id UUID PRIMARY KEY,
  popularity counter);

UPDATE cycling.popular_count
 SET popularity = popularity + 1
 WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
```

UDT / Blob

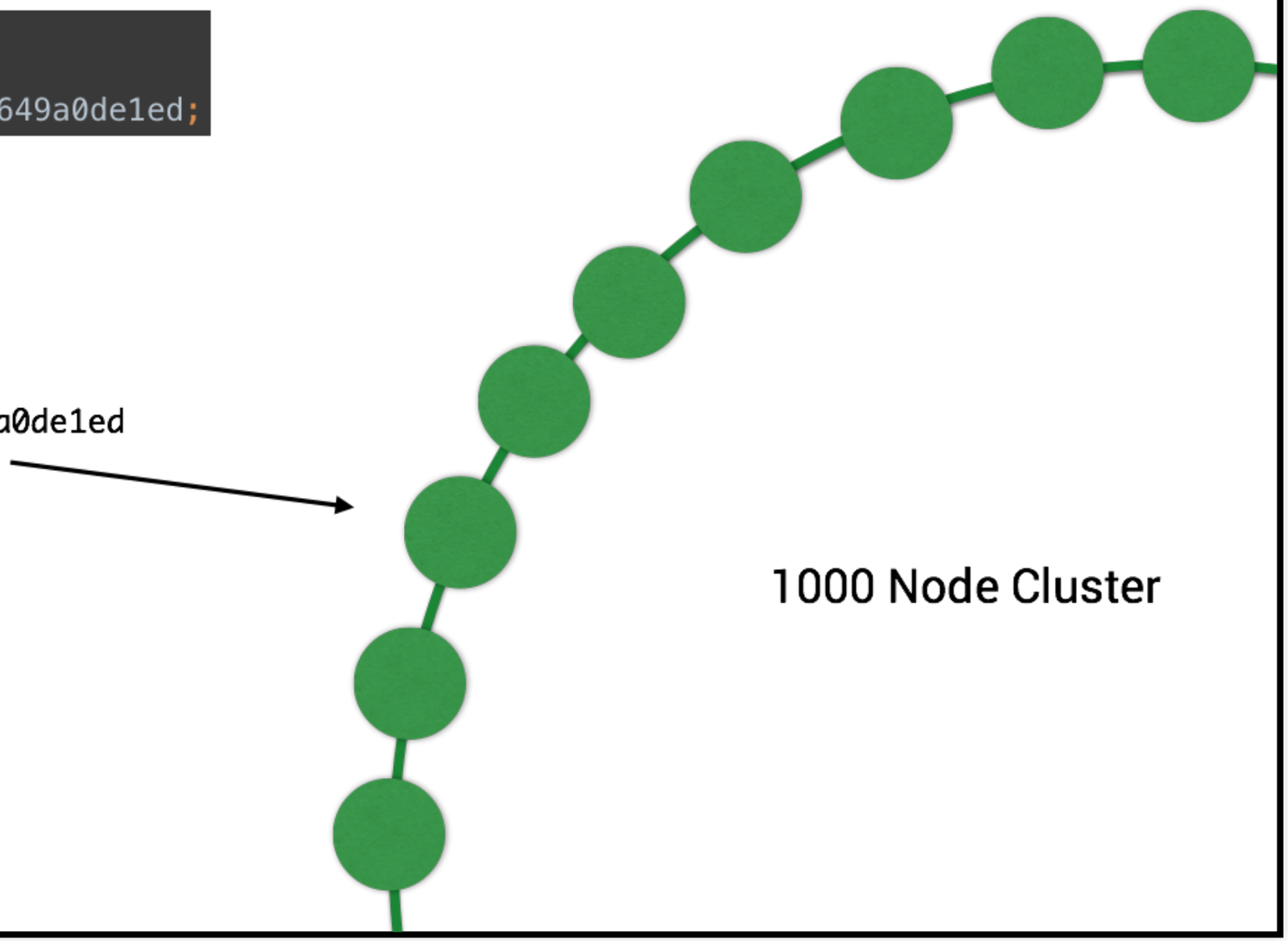
```
 CREATE TYPE cycling.basic_info (
  birthday timestamp,
  nationality text,
  weight text,
  height text
);

CREATE TABLE cycling.cyclist_stats (
            id uuid PRIMARY KEY, lastname text, basics FROZEN<basic_info>);
```

- **frozen** ⇒ cannot update parts of a UDT (blob semantics)
- used to model *one 2 many* relations

Query restrictions

```
SELECT name, description, added_date
FROM videos
WHERE videoid = 06049cbb-dfed-421f-b889-5f649a0de1ed;
```

videoid = 06049cbb-dfed-421f-b889-5f649a0de1ed



1000 Node Cluster

Physical model properties

```
CREATE TABLE users (
   user_id int,
   user_name text,
   user_age int,
   PRIMARY KEY (user_id));

SELECT * FROM users WHERE user_age = 22;
```

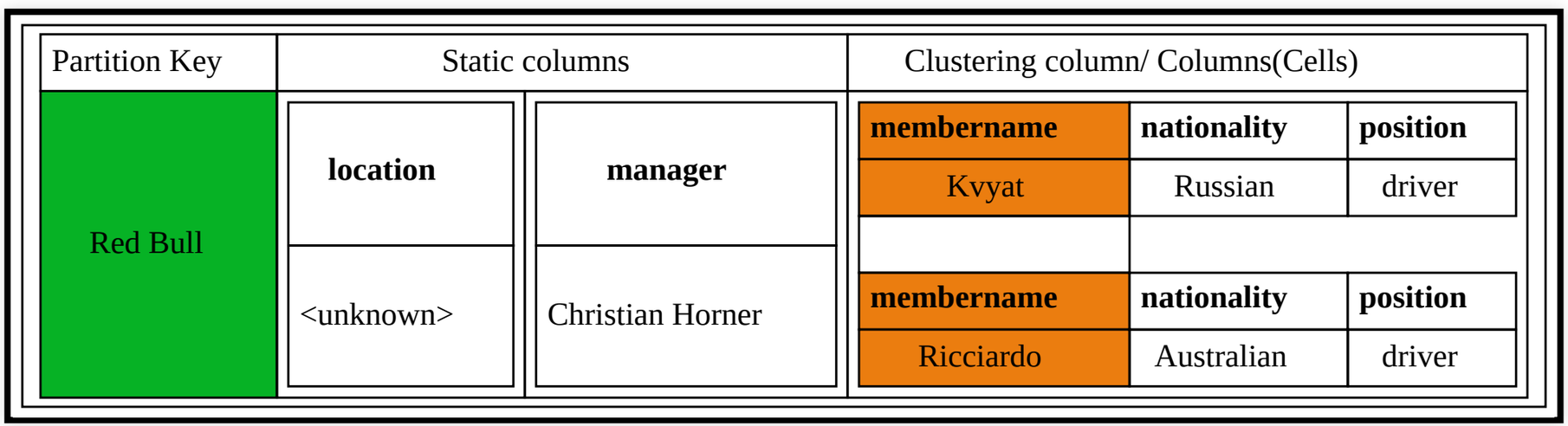
[Invalid query] message="Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"

Query restrictions

- all queries (INSERT/UPDATE/DELETE/SELECT) must provide **#partition**
  - exact match (=) on #partition,
- ***clustering columns*** ⇒ **range queries** (<, ≤, >, ≥) and **exact**
- **WHERE** clause only on columns in **PRIMARY KEY**
- if a clustering column is used ⇒ all clustering key columns that precede it must be used

Query restrictions

```
CREATE TABLE teammember_by_team (
  teamname text,
  manager text static,
  location text static,
  membername text, nationality text, position text,
  PRIMARY KEY ((teamname), membername)
);
SELECT * FROM teammember_by_team WHERE teamname='Red Bull'; --OK
SELECT * FROM teammember_by_team WHERE teamname='Red Bull' AND membername='Kyvat'; --OK

SELECT * FROM teammember_by_team WHERE membername='Kyvat'; --KO
SELECT * FROM teammember_by_team WHERE position='driver'; --KO
SELECT * FROM teammember_by_team WHERE manager='Christian Horner'; --KO
```



Allow filtering

- signals that our **query will not be efficient** (partition key is not fixed)
- it's almost **never a good ideea**
  - exception, when we know that only one partition will be involved
- ***SELECT * FROM blogs***
  - Cassandra will return you all the data that the table blogs contains
    - distributed scan
    - no ALLOW FILTERING WARNING!
- ***SELECT * FROM teammember_by_team WHERE position='driver'***
  - *scan all rows and filter the drivers*
  - potentially very inefficient

Allow filtering

⇒ change your data model ⇒ add an index, ***add another table***

Secondary indexes

- for **convenience** not for performance !
- index a column from the PRIMARY KEY
  - with low cardinality of few values !

```
CREATE TABLE cycling.rank_by_year_and_name (
  race_year int,
  race_name text,
  cyclist_name text,
  rank int,
  PRIMARY KEY ((race_year, race_name), rank)
);

SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015 AND race_name='TJI'; -- OK (both race_nam

SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015; -- KO
```

```
CREATE INDEX ryear ON cycling.rank_by_year_and_name (race_year);
SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015;
```

Materialized views (@deprecated)

- creates a query only table from a base table
- when changes are made to the base table the materialized view is automatically updated
- performance / caveats (USE TRACING !)

```
CREATE TABLE cc_transactions (
    userid text,
    year int,
    month int,
    day int,
    id int,
    amount int,
    card text,
    status text,
    PRIMARY KEY ((userid, year), month, day, id)
);
```

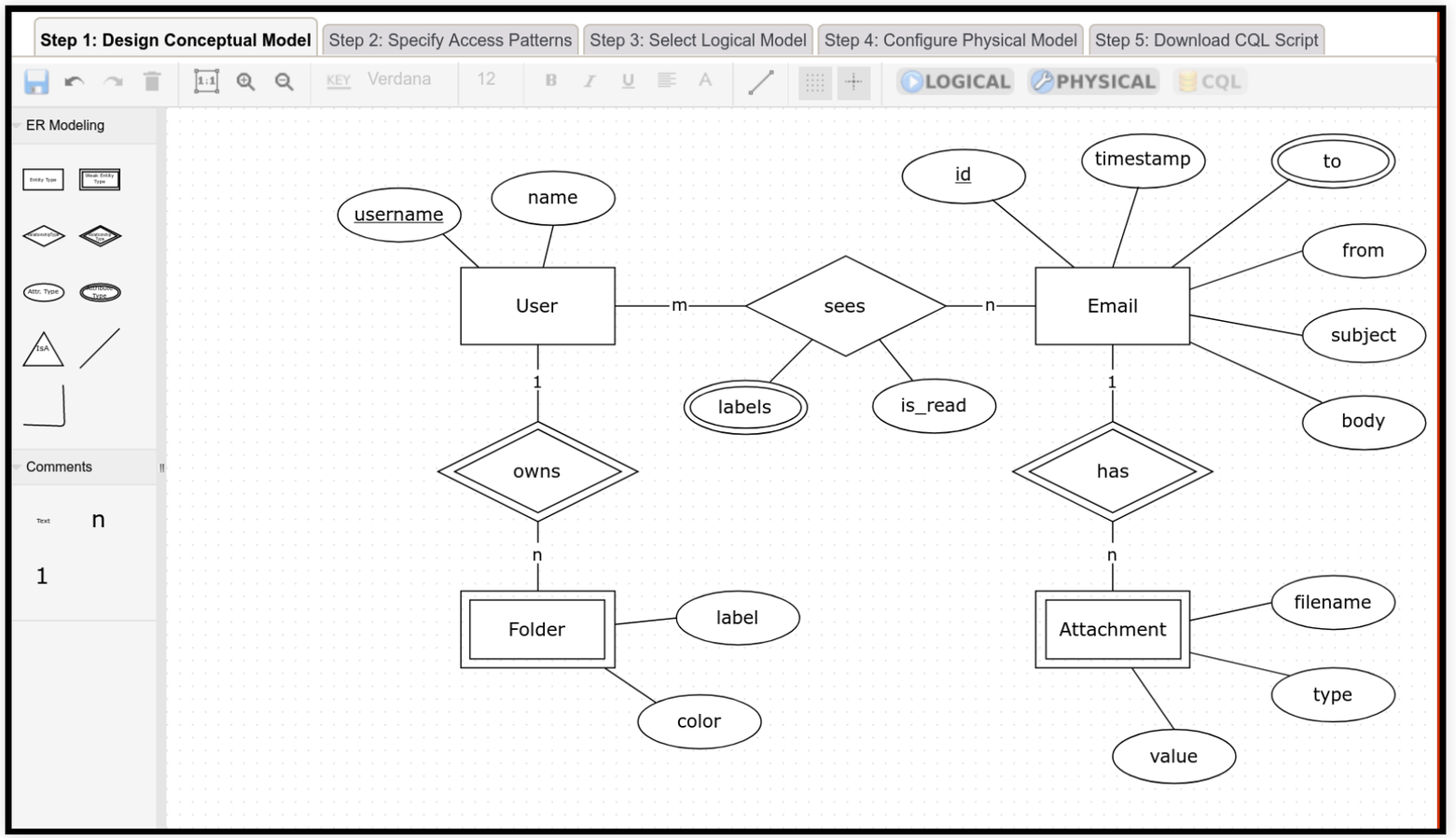
```
CREATE MATERIALIZED VIEW transactions_by_day AS
    SELECT year, month, day, userid, id, amount, card, status
    FROM cc_transactions
```

```
    WHERE userid IS NOT NULL AND year IS NOT NULL AND month IS NOT NULL AND day IS NOT NULL AND id IS NOT
    PRIMARY KEY ((year, month, day), userid, id);
```

Plan

- Cassandra Query Language (CQL) by examples
- **Data modeling with Cassandra**
- TP2: data modeling with Apache Cassandra

Data modeling : KDM approach



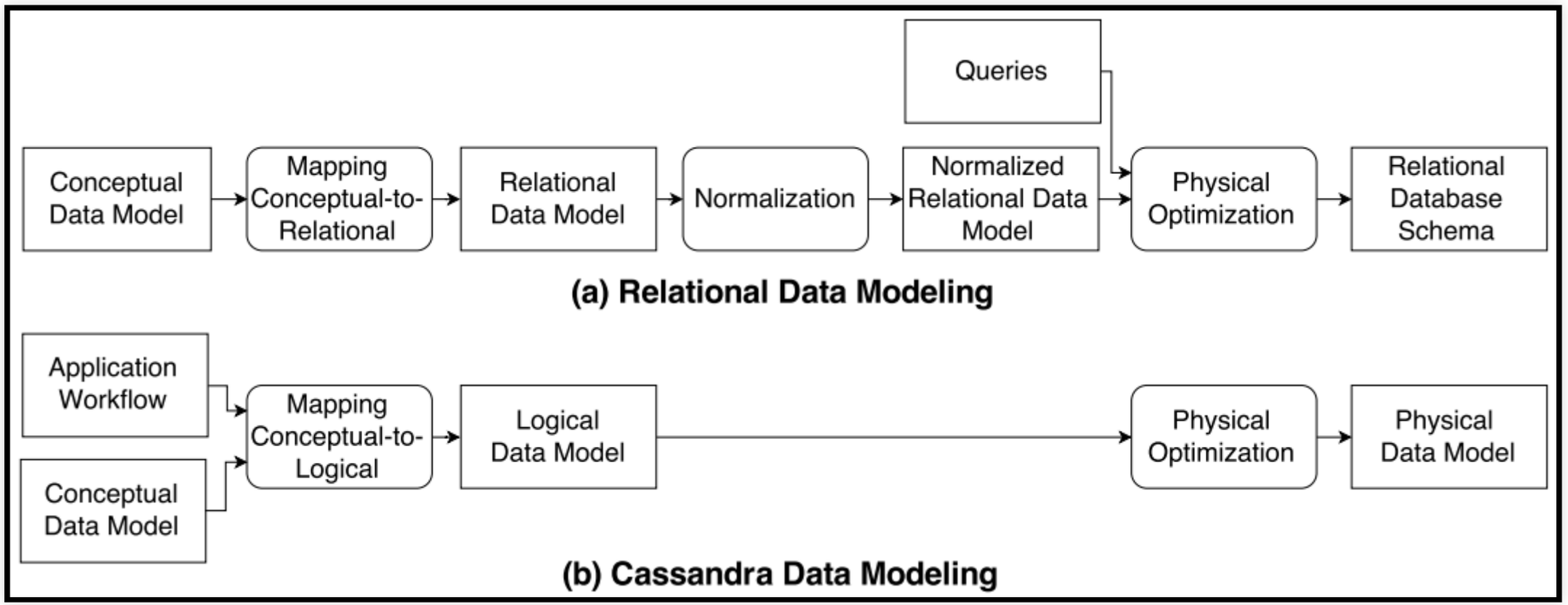
The Kashlev Data Modeler

A Big Data Modeling Methodology for Apache Cassandra

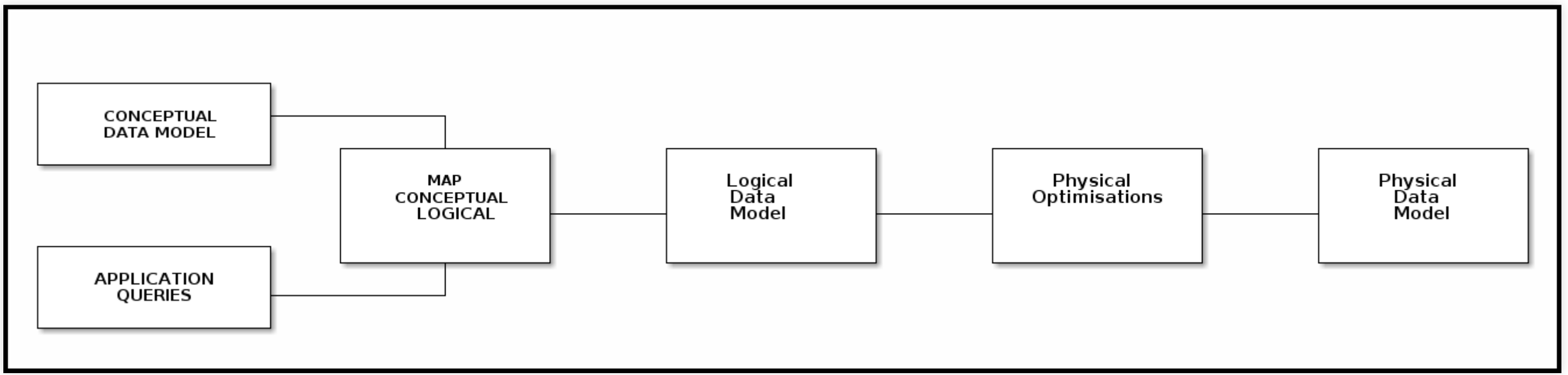
SQL vs Cassandra

- SQL:
  - relational model ⇒ general model able to answer all the queries
  - start with a conceptual ER model, design tables
  - optimize for data access patterns using Indexes
- Cassandra:
  - 1 data access path (table/index/materialized view) for each query
  - use the query workflow at the center of the data modeling

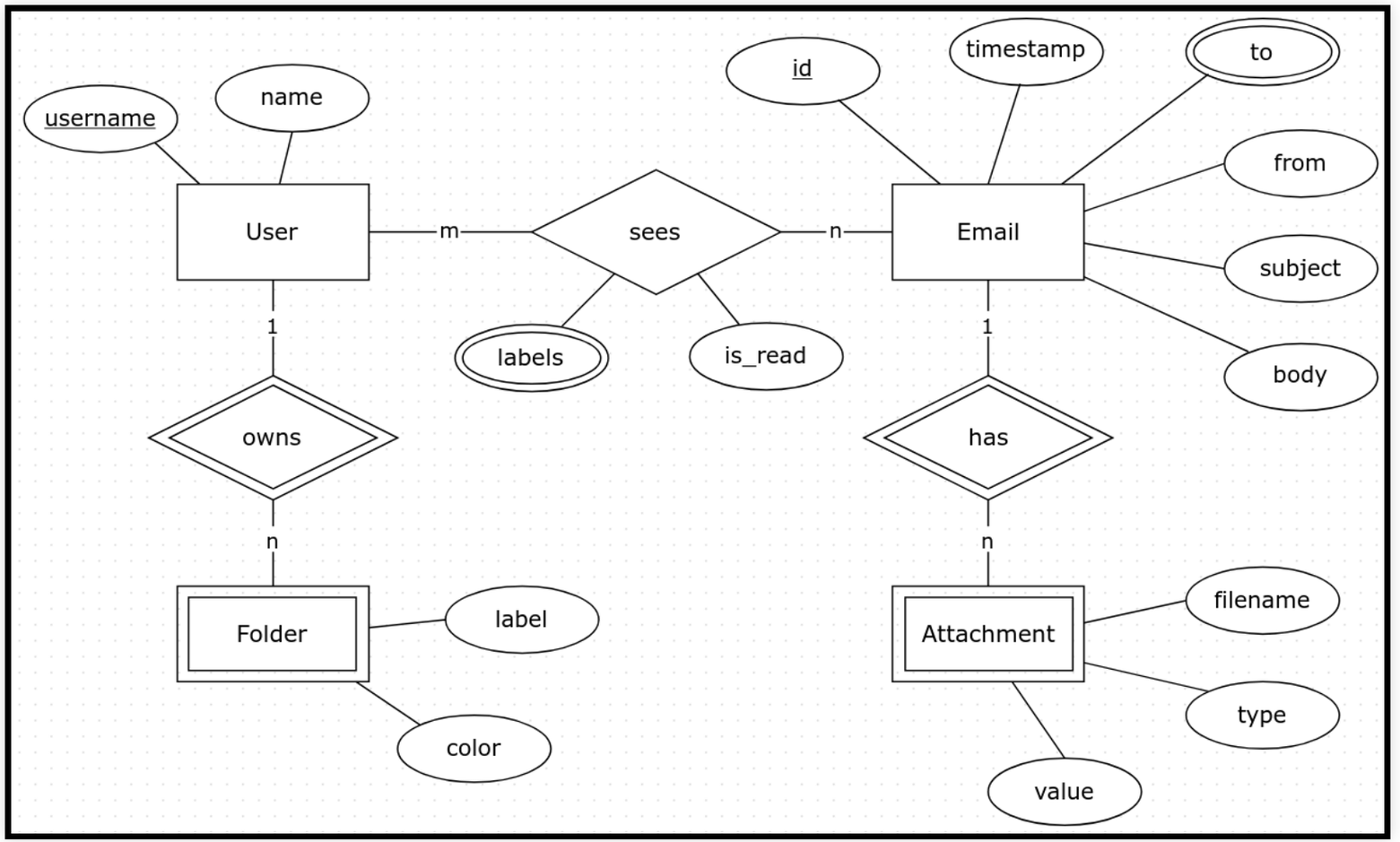
SQL vs Cassandra



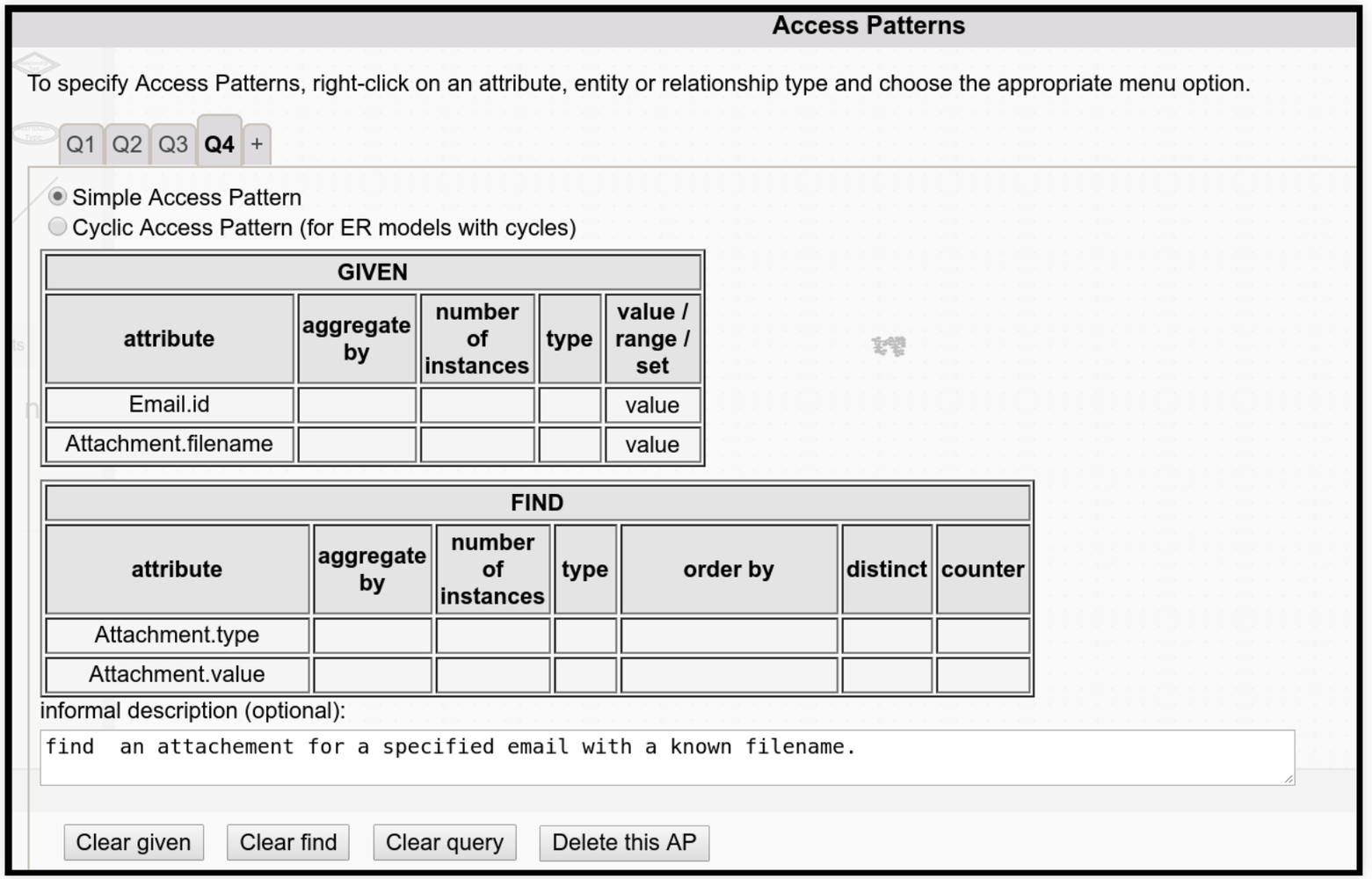
Modeling approach



Conceptual Data Model



Application queries ⇒ Access patterns



Access Patterns

To specify Access Patterns, right-click on an attribute, entity or relationship type and choose the appropriate menu option.

Q1 Q2 Q3 **Q4** +

(•) Simple Access Pattern
( ) Cyclic Access Pattern (for ER models with cycles)

GIVEN				
attribute	**aggregate by**	**number of instances**	**type**	**value / range / set**
Email.id				value
Attachment.filename				value

FIND						
attribute	**aggregate by**	**number of instances**	**type**	**order by**	**distinct**	**counter**
Attachment.type						
Attachment.value						

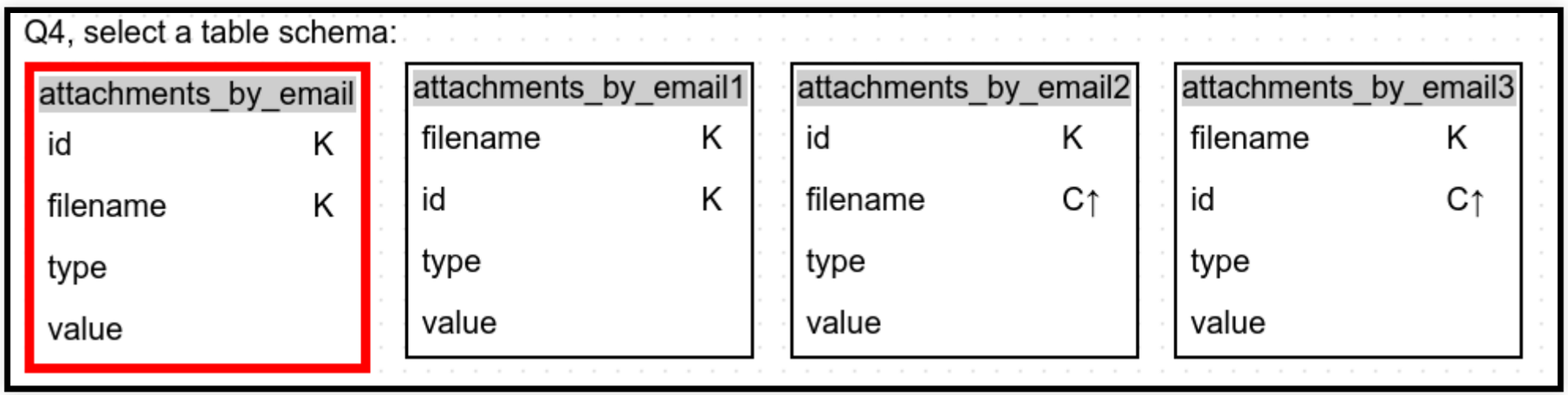
informal description (optional):

find an attachement for a specified email with a known filename.

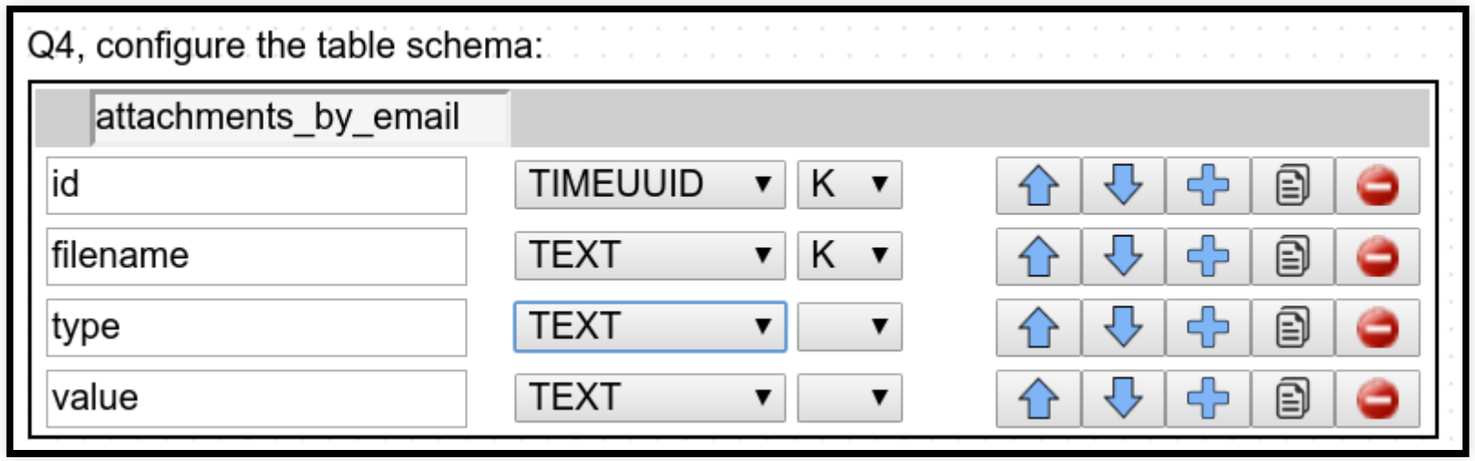
Clear given | Clear find | Clear query | Delete this AP

Logical Data Model

Q4, select a table schema:



Physical Data Model



CQL

```
// Q4:
CREATE TABLE attachments_by_email (id TIMEUUID, filename TEXT, type TEXT, value TEXT, PRIMARY KEY ((id,fil

/* SELECT type, value FROM attachments_by_email WHERE id=? and filename=?; */
```

Plan

- Cassandra Query Language (CQL) by examples
- Data modeling with Cassandra
- **TP2: data modeling with Apache Cassandra**

TP Cassandra Modeling

- Model simple time series in Cassandra:
  - focus on physical model + query opportunities
  - use sstabledump to understand the physical storage model
- Applying a KDM approach to model a IoT network

Ressources:

The Kashlev Data Modeler

A Big Data Modeling Methodology for Apache Cassandra

DatastaxAcademy and blogs

Modelisation Cassandra de Jérôme Mainaud