Assignment 14
Column-Family Databases

Nika Bakhtadze

Column Store for Relational Database

Given is our relational database lesson:

1. For what use case(s) would you store the content as rows, for what use case(s) would you store the content as columns?

Storing content as rows is beneficial if client wants to read whole (or almost whole) information of row / performing write operation on all (almost all) information of row. For example reading every field of student: SELECT * FROM student WHERE s_username = …

Storing content as columns is advantageous if one wants to read small number of columns within a large number of rows. If I have a lot number of rows but I want to read only one column, then using column storage is the best approach.

2. Write down 2 queries for our lesson db that are efficiently executed within a row store.

SELECT * FROM student WHERE s_gender = 'f';

SELECT * FROM teacher;

3. Write down 2 queries for our lesson db that are efficiently executed within a column store.

SELECT count(s_username) FROM student;

SELECT AVG(t_balance) as avg_balance FROM teacher;

- Which column(s) of the table lesson in the lesson database would be suitable for compression in a column store?

subjectcode character (5)
EN
EN
EN
EN
EN
EN
EN
EN
FR
FR
EN
EN
EN
EN
EN

In general, suitable columns for compression are columns that have redundant values / have small number of different values. Such column(s) in our table lesson is subjectcode due to very limited number of different values.

- Take the first 6 rows of the table lesson. (Populate if not yet done.)

	t_id [PK] integer	lesson_time [PK] timestamp without time zone	s_username character (30)	subjectcode character (5)	insertion_time timestamp without time zone
1	1	2023-03-06 05:22:12	Mickey ...	EN	2025-02-18 17:59:30.777694
2	1	2023-03-08 05:22:12	Mickey ...	EN	[null]
3	1	2024-01-19 18:46:19.879467	Rose ...	EN	2025-02-18 17:59:30.777694
4	1	2024-02-07 05:22:12	Mickey ...	EN	2025-02-18 18:01:28.230769
5	1	2024-03-03 05:22:12	Rose ...	EN	[null]
6	1	2024-03-04 05:22:12	Rose ...	EN	[null]

- Create bitmap and run-length encoding for the column and fill the index for the first 6 rows of your table.

Bitmap index on column subjectcode with values {EN, FR, CS}

Row number (down) /subjectcode (right)	EN	FR	CS
1	1	0	0
2	1	0	0
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0

Run_Length Encoding:

0,5	5	5
0 zeroes, 5 ones	5 zeroes	5 zeroes

Create the table (column family) for Q3

Q3: Return a teacher list by given, specific postalcode
= Return the teachers that live at a given, specific postalcode

Teachers should be grouped by postalcodes, each node will store each postalcode and teachers living there.

- Write the Cassandra create command for table for Q3

```
CREATE TABLE lesson.teachers_by_postalcode (
t_email text,
t_name text,
t_phone text,
t_postalcode text,
address frozen,
t_subjects set ,
t_education text,
PRIMARY KEY ((t_postalcode), t_email)
WITH comment = 'Q3. Find teachers according to specific postalcode'
AND CLUSTERING ORDER BY (t_email ASC) ;
```

- What is the primary key? PRIMARY KEY ((t_postalcode), t_email)
- Is it a composite or an atomic key? composite, partition key + clustering key
- what is the partition key? t_postalcode
- Do you have a clustering key? t_email, sorted in ascending order

Populate / query the table

Are the following queries possible / allowed? How are they executed? Are they efficient or not?

- select * from teachers_by_postalcode where postalcode = "xxxx"

Possible/allowed. Very efficient, because it depends on main principle why to use column storage. This query use partition key – postalcode. Initially, value 'xxxx' is hashed to some number, node is found and every row for that node is returned.

- select * from teachers_by_postalcode where postalcode between "xxxx" and "zzzz"

Not allowed. As I gathered information, Cassandra does not support range queries on partition keys.

- select * from teachers_by_postalcode where t_email = "xxxxx"

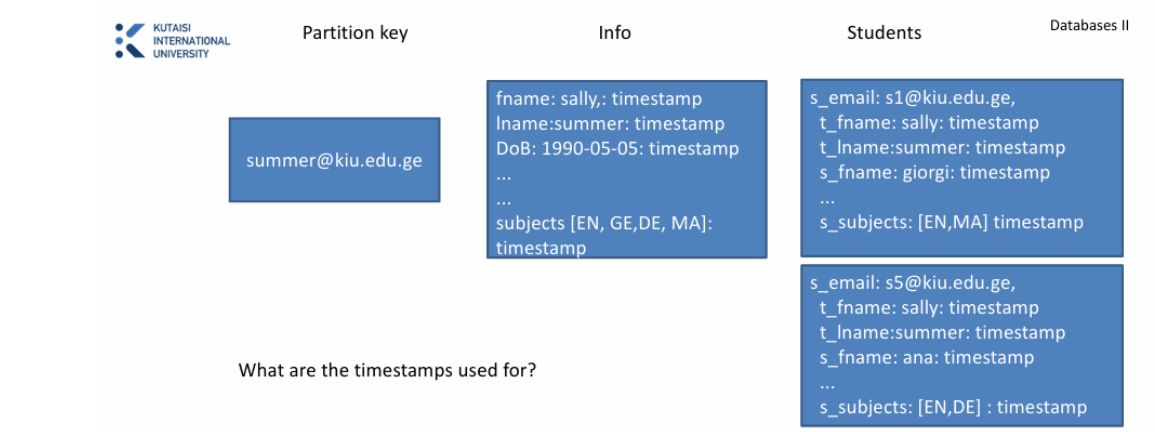
Not allowed. Query is not possible without partitioning key, only clustering key is not enough.

- select * from teachers_by_postalcode

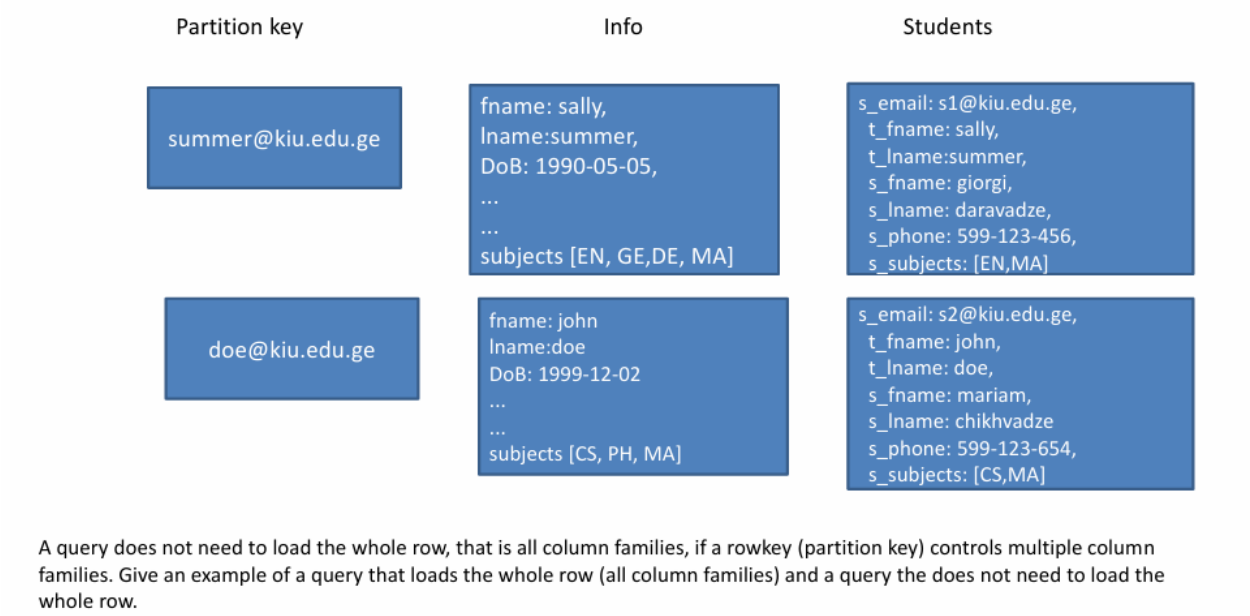
Possible/allowed. Executable, but very inefficient, because it has to go through all nodes and retrieve all columns/rows, like full table scan. As data size increases, it becomes more and more inefficient.

- select * from teachers_by_postalcode where postalcode = "xxxx" and t_email in {"xxxx", "yyyy", "wwww", "zzzz"}

Possible/allowed. Very efficient. It initially hashes postalcode and finds desired node. Then finds such t_emails as it was specified, which is simple process due to clustering key.



The main reason here timestamps are used for is to deal with conflicts and resolve them. If same data objects are updated, then we may face some stale data on some nodes. We store timestamps to track which data was lastly modified. By this we can get latest version of that data and update stale ones with newer one (concurrent writes).



Loading whole row (meaning all column families, Info + Students):

SELECT * FROM students_by_teacher WHERE t_email = ‘summer@kiu.edu.ge’

Or

SELECT * FROM students_by_teacher WHERE t_email = ‘doe@kiu.edu.ge’

Doesn’t loading whole row:

SELECT DoB FROM students_by_teacher where t_email = ‘summer@kiu.edu.ge’

DoB is inside Info column family, thus there is no need to load Students column family.

Use Case Examples

- Global Press Agencies storing their articles / press releases by date / topic
  - write a query for this database (keyspace)
  - write down a possible table (column-family) definition

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

CREATE TABLE press.articles_by_date (
   release_date date,
   topic text,
   article_id uuid,
   author text,
   PRIMARY KEY ((topic), release_date))
WITH CLUSTERING ORDER BY (release_date DESC);
```

I think taking topic as partitioning key is efficient for column-family. Articles will be partitioned according to topic, inside specified topic, articles will be stored with latest to oldest, so data retrieval is efficient. This is case is user prioritizes articles according to topics.

Other way is to take release_date as partitioning key and topic as clustering key. By this approach, data will be partitioned according to date, so retrieval according the specific date becomes more efficient.

- Social media: feeds by followed persons per user
  - write a query for this database (keyspace)
  - write down a possible table (column-family) definition

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

CREATE TABLE soc_media. feeds (
   user_id uuid,
   follower_id uuid,
   post_time timestamp,
   post text,
   PRIMARY KEY ((user_id), post_time))
WITH CLUSTERING ORDER BY (post_time DESC);
```

I think user_id is best candidate for partitioning key as it will enable to group feeds per user. Clustering key must be post_time and ordering must be descending. By this approach, user will see feeds by followed persons according to time they posted it, from latest to oldest.