Indexing in Cassandra

Credit where credit is due

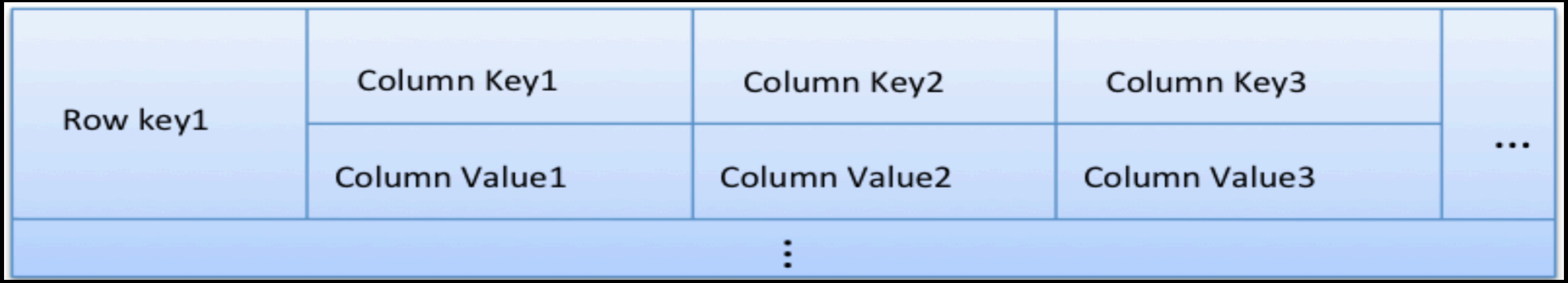
- **Learning Apache Cassandra by** Mat Brown
- **Apache Cassandra Hands-On Training Level One by Ruth Stryker**
- Website: http://planetcassandra.org
- Cassandra documentation: http://www.datastax.com/

Topics

- Introduction to Indexing in Cassandra
- Primary Indexes
- Secondary Indexes
- Composite Columns
- When to use Indexing?

Indexing in Cassandra

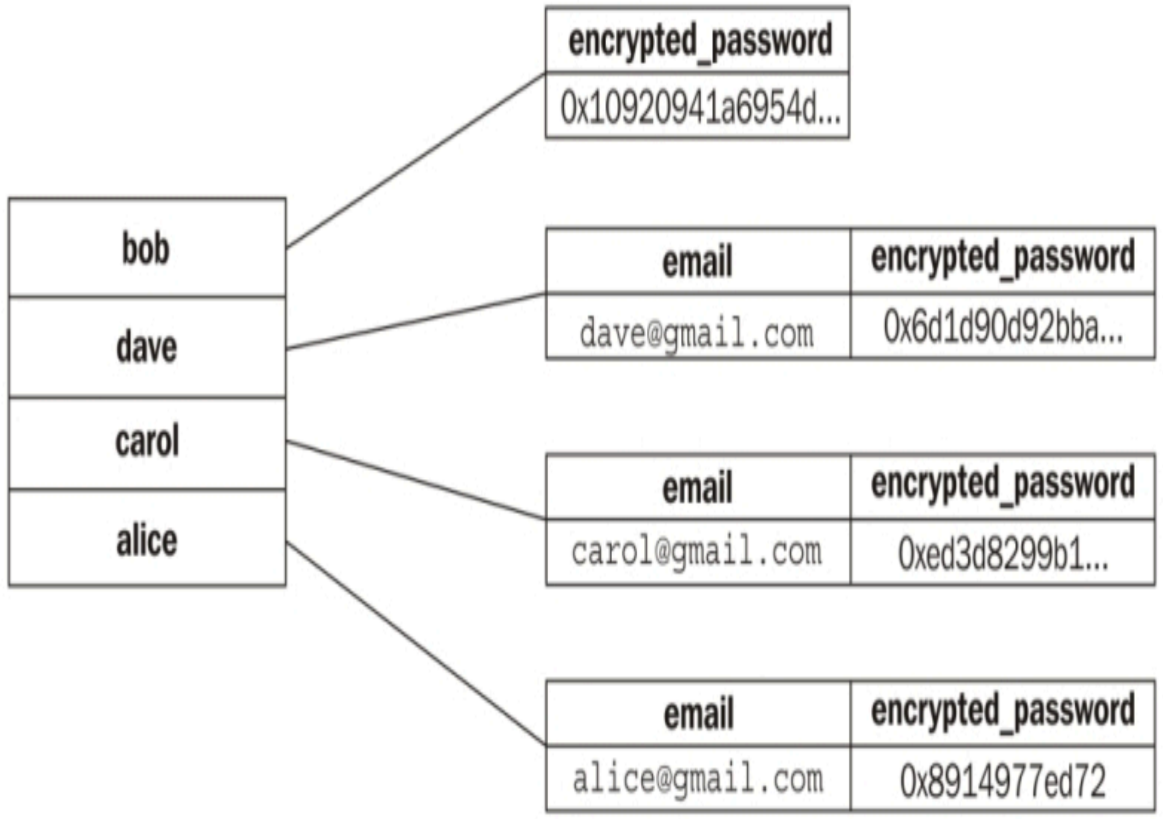
- Physically, data is stored on blocks in data structure form (sstable in Cassandra).
- Data on a Cassandra node is stored locally for each row. Rows are distributed across multiple nodes, but all columns for a particular row key will be stored locally on a node.



Row key1	Column Key1	Column Key2	Column Key3	...
	Column Value1	Column Value2	Column Value3	
		⋮		

Definition

- Indexing is a process to create and manage a data structure called Index for fast data retrieval.
- Indexes are stored in sorted order into B-tree (balanced tree) structure, where indexes are leaf nodes under branch nodes.
- Cassandra by default provides the primary index as row key for faster retrieval by row key.
- Accessing data records without a primary key or index would require a linear search across multiple nodes.


bob
dave
carol
alice
encrypted_password
0x10920941a6954d...
email
encrypted_password
dave@gmail.com
0x6d1d90d92bba...
email
encrypted_password
carol@gmail.com
0xed3d8299b1...
email
encrypted_password
alice@gmail.com
0x8914977ed72

Primary Index

Relational Databases:

- a primary key is the unique key used to identify each row in a table.

Cassandra:

- the primary index for a column family is the index of its row keys. Each node maintains this index for the data it manages.

Primary Index

First off we'll create and use a test keyspace:

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

Let's look at a simple table composed of three fields using a simple primary key.

```
cqlsh:test> CREATE TABLE example (
      ... field1 int PRIMARY KEY,
      ... field2 int,
      ... field3 int);
```

Let’s insert some rows:

```
cqlsh:test> INSERT INTO example (field1, field2, field3) VALUES ( 1,2,3);
cqlsh:test> INSERT INTO example (field1, field2, field3) VALUES ( 4,5,6);
cqlsh:test> INSERT INTO example (field1, field2, field3) VALUES ( 7,8,9);
```

Let’s see these rows in cql shell:

```
cqlsh:test> SELECT * FROM example;

 field1 | field2 | field3
--------+--------+--------
      1 |      2 |      3
      4 |      5 |      6
      7 |      8 |      9
```

Let’s look at these rows in cassandra-cli:

```
[default@test] list  example;
-------------------
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: 1
=> (name=, value=, timestamp=1427339274240000)
=> (name=field2, value=00000002, timestamp=1427339274240000)
=> (name=field3, value=00000003, timestamp=1427339274240000)
-------------------
RowKey: 4
=> (name=, value=, timestamp=1427339280727000)
=> (name=field2, value=00000005, timestamp=1427339280727000)
=> (name=field3, value=00000006, timestamp=1427339280727000)
-------------------
RowKey: 7
=> (name=, value=, timestamp=1427339288575000)
=> (name=field2, value=00000008, timestamp=1427339288575000)
=> (name=field3, value=00000009, timestamp=1427339288575000)
```

- The value of the CQL primary key is used internally as the row key (which in the new CQL paradigm is being called a “partition key”).
- The names of the non-primary key CQL fields are used internally as columns names. The values of the non-primary key CQL fields are then internally stored as the corresponding column values.

Composite Columns

- If we want to sort data in a table, then we need to use a composite-keyed table.
- A composite-key made up of multiple fields from the table.

Example:

```
CREATE TABLE "user_status_updates" (
  "username" text,
  "id" timeuuid,
  "body" text,
  PRIMARY KEY ("username", "id")
)
```

  - Note that, we make a separate PRIMARY KEY declaration at the end of the list of columns, which itself specifies the two columns that make up the compound primary key.

Composite Columns

- In the example, username is a **partition key**. A table's partition key groups rows together into logically related bundles.
- The id column a **clustering column**. The job of a clustering column is to determine the ordering of rows within a partition.
- All Cassandra tables must have a partition key; clustering columns are optional.

Is sorting by clustering column efficient?

Sorting any collection at read time is expensive for a non-trivial number of elements. Happily, Cassandra stores rows in clustering order, so when you retrieve them, it simply returns them in the order they're stored in. There's no expensive sorting operation at read time.

Composite Columns: Example

```
cqlsh:test> create table testTable (col1 int, col2 int, col3 int, primary key (col1, col2));

cqlsh:test> insert into testTable values (1, 2, 3);
cqlsh:test> insert into testTable (col1, col2, col3) values (1, 2, 3);
cqlsh:test> insert into testTable (col1, col2, col3) values (1, 3, 4);
cqlsh:test> insert into testTable (col1, col2, col3) values (1, 4, 5);
cqlsh:test> insert into testTable (col1, col2, col3) values (1, 7, 7);
cqlsh:test> insert into testTable (col1, col2, col3) values (1, 5, 7);

cqlsh:test> select * from testTable;

 col1 | col2 | col3
------+------+------
    1 |    2 |    3
    1 |    3 |    4
    1 |    4 |    5
    1 |    5 |    7
    1 |    7 |    7

(5 rows)
```

Composite Columns

- A table can define one or more partition key columns, and zero or more clustering columns.
- Example, we might want to allow users to reply to other users' status updates. In this case, each status update would have a stream of replies; replies would be partitioned by the full primary key of the original status update, and each reply would get its own timestamped UUID:

```
CREATE TABLE "status_update_replies" (
  "status_update_username" text,
  "status_update_id" timeuuid,
  "id" timeuuid,
  "author_username" text,
  "body" text,
  PRIMARY KEY (
    ("status_update_username", "status_update_id"),
    "id"
  )
);
```

Secondary Indexes

- In a distributed cluster, data for a column family is distributed across multiple nodes. However data for a given row key value will always be on the same node. Using the primary index we can always retrieve a row. But what about retrieving it using non-row key values?
- Cassandra provides support to add indexes over column values, called **Secondary indexes.**

Secondary Indexes

Example:
Let's create **twitter** keyspace and column family **users**.

```
create keyspace twitter with replication
= {'class' : 'SimpleStrategy' , 'replication_factor' : 3};

use twitter;

create table users (user_id uuid primary key, first_name text,
twitter_handle text);

create index fname_idx on users(first_name);
```

Secondary Indexes

Let’s insert a few rows in the users column family.

```
insert into users(user_id,first_name,twitter_handle)
values(now(),'apress','#apress_team');
insert into users(user_id,first_name,twitter_handle)
values(now(),'jonathan','#jhassell');
insert into users(user_id,first_name,twitter_handle)
values(now(),'vivek','#mevivs');
insert into users(user_id,first_name,twitter_handle)
values(now(),'vivek','#vivekab');
```

Let’s try to find records using the indexed column first_name

```
select * from users where first_name='vivek';
```

Secondary Indexes

- Query over indexed column first_name with value 'vivek' returns two rows. Here both rows can be available on the same node or different nodes.

```
 user_id                              | first_name | twitter_handle
--------------------------------------+------------+----------------
 2ae21a70-d8c7-11e4-b769-cb43778b61a5 |      vivek |       #vivekab
 2ae1f360-d8c7-11e4-b769-cb43778b61a5 |      vivek |        #mevivs

(2 rows)
```

- if we try to fetch rows using column twitter_handle, which is non-indexed:

```
select * from users where twitter_handle='#imvivek';
```

it results in the following error:

```
Bad Request: No indexed columns present in by-columns clause with Equal operator
```

Secondary Indexes

Let’s try another type of example:

Add another column age:

```
alter table users  add age int  ;
```

Update a few rows for age:

```
update users set age = 21 where user_id = 498d9f00-d47f-11e3-be07-79e7dcea6dd7;
update users set age = 51 where user_id = 498b06f0-d47f-11e3-be07-79e7dcea6dd7;
```

Create an index on the age column:

```
create index age_idx on users(age);
```

Secondary Indexes

Now, try to fetch user_id for a user with age 21:
select user_id from users where age =21;

```
 user_id
--------------------------------------
 498d9f00-d47f-11e3-be07-79e7dcea6dd7

(1 rows)
```

Next, let's try to fetch users whose age is greater than 21:

select user_id from users where age >=21;

Will this work?

Secondary Indexes

- You can create multiple secondary indexes, for example on the firstname and lastname columns of the users table, and use multiple conditions in the WHERE clause to filter the results:

```
CREATE INDEX users_fname  ON users (firstname);
CREATE INDEX users_lname ON users (lastname);
```

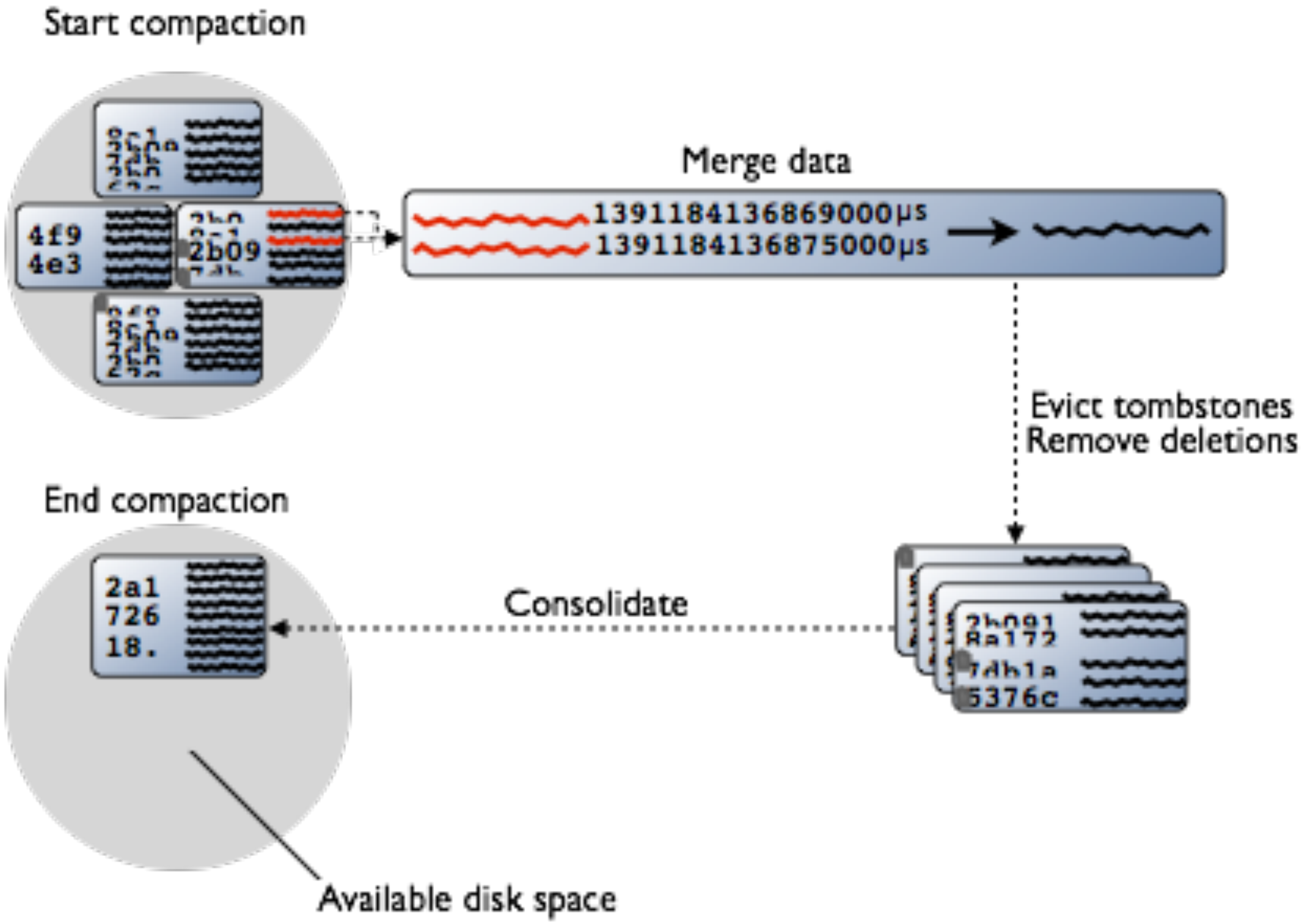
```
SELECT * FROM users WHERE users_fname = 'bob' AND
users_lname = 'smith';
```

When querying multiple columns which column is queried first?

Secondary Indexes

Best Used When:

- No scan (<, <=, =>, >) required. Secondary indexes allow users to retrieve records using indexed columns with “=” only.
- Columns with low cardinality values are generally recommended for indexing as probability of retrieving data from various nodes in a cluster is very high. Whereas a column with high cardinality may result in performance overhead under high data load and large Cassandra clusters.


Start compaction
4f9
4e3
2b09
Merge data
1391184136869000μs
1391184136875000μs
Evict tombstones
Remove deletions
End compaction
2a1
726
18.
Consolidate
5376c
Available disk space

Indexes

When Not to Use:

- On a frequently updated or deleted column. Cassandra stores tombstones in the index until the tombstone limit reaches 100K cells. After exceeding the tombstone limit, the query that uses the indexed value will fail.

Tombstone:

A marker in a row that indicates a column was deleted. During compaction, marked column are deleted.

Indexes

Some Implementation details:

- Indexes are built in the background automatically, without blocking reads or writes.
- They are just another column family. They are not directly accessible to users.
- The row and index updates are one, atomic operation.
- Each node only indexes data that it holds locally.