Indexing Example 2

The single most important aspect of ensuring that your cluster will be successful and high-performing is in getting the data model right. The main aspect of data modeling is in designing a proper primary key. Primary keys in Cassandra are split into two parts: the partition key, and the clustering key.

```
CREATE KEYSPACE IF NOT EXISTS admatic WITH replication = {'class':'SimpleStrategy',
'replication_factor' : 1};
use admatic;

CREATE TABLE logins_by_user (
  user_id text,
  login_datetime timestamp,
  origin_ip text,
  PRIMARY KEY ((user_id), login_datetime)
) WITH CLUSTERING ORDER BY (login_datetime DESC);

INSERT INTO logins_by_user (user_id, login_datetime, origin_ip) VALUES ('admatic','2
017-06-01 12:36:01','192.168.0.101');
INSERT INTO logins_by_user (user_id, login_datetime, origin_ip) VALUES ('admatic','2
017-06-01 12:53:28','192.168.0.101');
INSERT INTO logins_by_user (user_id, login_datetime, origin_ip) VALUES ('admin','201
7-06-02 13:23:11','192.168.0.105');
INSERT INTO logins_by_user (user_id, login_datetime, origin_ip) VALUES ('admatic','2
017-06-03 09:04:55','192.168.0.101');

SELECT * FROM logins_by_user WHERE user_id='admatic';

 user_id | login_datetime                  | origin_ip
---------+---------------------------------+---------------
 admatic | 2017-06-03 09:04:55.000000+0000 | 192.168.0.101
 admatic | 2017-06-01 12:53:28.000000+0000 | 192.168.0.101
 admatic | 2017-06-01 12:36:01.000000+0000 | 192.168.0.101

(3 rows)

SELECT token(user_id), user_id, login_datetime FROM logins_by_user WHERE user_id='ad
matic';

 system.token(user_id) | user_id | login_datetime
-----------------------+---------+---------------------------------
   4540968551724967090 | admatic | 2017-06-03 09:04:55.000000+0000
   4540968551724967090 | admatic | 2017-06-01 12:53:28.000000+0000
   4540968551724967090 | admatic | 2017-06-01 12:36:01.000000+0000

(3 rows)
```

Looking at the first column of the result set, you can see that the `user_id` all match to the same token. This means that they will be stored in the same partition, and thus, together on any node responsible for the token range that encompasses `4540968551724967090`. Within this partition, the results are ordered by `login_datetime`, descending.

```
SELECT token(user_id), user_id, login_datetime FROM logins_by_user;

 system.token(user_id) | user_id | login_datetime
-----------------------+---------+---------------------------------
  -1036951229475898308 |   admin | 2017-06-02 13:23:11.000000+0000
   4540968551724967090 | admatic | 2017-06-03 09:04:55.000000+0000
   4540968551724967090 | admatic | 2017-06-01 12:53:28.000000+0000
   4540968551724967090 | admatic | 2017-06-01 12:36:01.000000+0000

(4 rows)
```

`WHERE` clauses in a query can only contain components of the primary key. Furthermore, they must respect the order of the keys.

```
SELECT * FROM logins_by_user WHERE origin_ip='192.168.0.101';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute
 this query as it might involve data filtering and thus may have unpredictable perfo
rmance. If you want to execute this query despite the performance unpredictability,
use ALLOW FILTERING"

SELECT * FROM logins_by_user WHERE origin_ip='192.168.0.101' ALLOW FILTERING;

 user_id | login_datetime                  | origin_ip
---------+---------------------------------+---------------
 admatic | 2017-06-03 09:04:55.000000+0000 | 192.168.0.101
 admatic | 2017-06-01 12:53:28.000000+0000 | 192.168.0.101
 admatic | 2017-06-01 12:36:01.000000+0000 | 192.168.0.101

(3 rows)
```

You can omit clustering keys, but you cannot skip them. For instance, I can omit `login_datetime` because I am specifying the keys that precede it. I cannot omit `user_id` and only query by `login_datetime`, because Cassandra needs to know which partition to look at, and cannot figure that out from a clustering key.

```
SELECT * FROM logins_by_user WHERE login_datetime='2017-06-01 12:36:01';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute
 this query as it might involve data filtering and thus may have unpredictable perfo
rmance. If you want to execute this query despite the performance unpredictability,
use ALLOW FILTERING"

SELECT * FROM logins_by_user WHERE login_datetime='2017-06-01 12:36:01' ALLOW FILTER
ING;

 user_id | login_datetime                  | origin_ip
```

```
---------+---------------------------------+---------------
 admatic | 2017-06-01 12:36:01.000000+0000 | 192.168.0.101

(1 rows)
```

When to use an index

Built-in indexes are best on a table having many rows that contain the indexed value. The more unique values that exist in a particular column, the more overhead you will have, on average, to query and maintain the index. For example, suppose you had a `races` table with a billion entries for cyclists in hundreds of races and wanted to look up rank by the cyclist. Many cyclists' ranks will share the same column value for race year. The `race_year` column is a good candidate for an index.

When not to use an index

Do not use an index in these situations:

- On high-cardinality columns for a query of a huge volume of records for a small number of results.
- In tables that use a counter column.
- On a frequently updated or deleted column.
- To look for a row in a large partition unless narrowly queried.

Using a secondary index

Create indexes on a column after defining a table. Secondary indexes are used to query a table using a column that is not normally query-able.

Secondary indexes are tricky to use and can impact performance greatly. The index table is stored on each node in a cluster, so a query involving a secondary index can rapidly become a performance nightmare if multiple nodes are accessed. A general rule of thumb is to index a column with low cardinality of few values. Before creating an index, be aware of when and when not to create an index.

```
use admatic;

SELECT * FROM logins_by_user WHERE origin_ip='192.168.0.101';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute
 this query as it might involve data filtering and thus may have unpredictable perfo
rmance. If you want to execute this query despite the performance unpredictability,
use ALLOW FILTERING"
```

An index is created for the `origin_ip`, and the query will succeed.

```
CREATE INDEX rip ON logins_by_user (origin_ip);

SELECT * FROM logins_by_user WHERE origin_ip='192.168.0.101';

 user_id | login_datetime                  | origin_ip
---------+---------------------------------+---------------
 admatic | 2017-06-03 09:04:55.000000+0000 | 192.168.0.101
 admatic | 2017-06-01 12:53:28.000000+0000 | 192.168.0.101
 admatic | 2017-06-01 12:36:01.000000+0000 | 192.168.0.101

(3 rows)
```

A clustering column can also be used to create an index.

```
SELECT * FROM logins_by_user WHERE login_datetime='2017-06-01 12:36:01';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute
 this query as it might involve data filtering and thus may have unpredictable perfo
rmance. If you want to execute this query despite the performance unpredictability,
use ALLOW FILTERING"

CREATE INDEX rdate ON logins_by_user (login_datetime);

SELECT * FROM logins_by_user WHERE login_datetime='2017-06-01 12:36:01';

 user_id | login_datetime                  | origin_ip
---------+---------------------------------+---------------
 admatic | 2017-06-01 12:36:01.000000+0000 | 192.168.0.101

(1 rows)
```

Using multiple indexes

Indexes can be created on multiple columns and used in queries. The general rule about cardinality applies to all columns indexed. In a real-world situation, certain columns might not be good choices, depending on their cardinality.

```
CREATE TABLE cyclist_alt_stats ( id UUID PRIMARY KEY, lastname text, birthday timest
amp, nationality text, weight text, height text );

CREATE INDEX birthday_idx ON cyclist_alt_stats ( birthday );
CREATE INDEX nationality_idx ON cyclist_alt_stats ( nationality );
```

```
INSERT INTO cyclist_alt_stats (id, lastname, birthday, nationality, weight, height)
```

```
VALUES (41d01b63-244e-435d-bd6d-5dc8a0addb8c, 'J', '1982-01-29', 'Russia', '80', '18
0');

SELECT * FROM cyclist_alt_stats WHERE birthday = '1982-01-29' AND nationality = 'Rus
sia';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute
 this query as it might involve data filtering and thus may have unpredictable perfo
rmance. If you want to execute this query despite the performance unpredictability,
use ALLOW FILTERING"
```

The indexes have been created on appropriate low cardinality columns, but the query still fails. Why? The answer lies with the partition key, which has not been defined. When you attempt a potentially expensive query, such as searching a range of rows, the database requires the ALLOW FILTERING directive. The error is not due to multiple indexes, but the lack of a partition key definition in the query.

```
SELECT * FROM cyclist_alt_stats WHERE birthday = '1982-01-29' AND nationality = 'Rus
sia' ALLOW FILTERING;

 id                                   | birthday                        | height | l
astname | nationality | weight
--------------------------------------+---------------------------------+--------+--
--------+-------------+--------
 41d01b63-244e-435d-bd6d-5dc8a0addb8c | 1982-01-29 00:00:00.000000+0000 |    180 |
      J |      Russia |     80

(1 rows)
```

Indexing a collection