Querying data

While Apache Cassandra is known for its restrictive query model (design your tables to suit your queries), the previous content has shown that CQL can still be quite powerful. Consider the following table:

```
CREATE TABLE query_test (
 pk1 TEXT,
 pk2 TEXT,
 ck3 TEXT,
 ck4 TEXT,
 c5 TEXT,
 PRIMARY KEY ((pk1,pk2), ck3, ck4))
WITH CLUSTERING ORDER BY (ck3 DESC, ck4 ASC);

INSERT INTO query_test (pk1,pk2,ck3,ck4,c5) VALUES ('a','b','c1','d1','e1');
INSERT INTO query_test (pk1,pk2,ck3,ck4,c5) VALUES ('a','b','c2','d2','e2');
INSERT INTO query_test (pk1,pk2,ck3,ck4,c5) VALUES ('a','b','c2','d3','e3');
INSERT INTO query_test (pk1,pk2,ck3,ck4,c5) VALUES ('a','b','c2','d4','e4');
INSERT INTO query_test (pk1,pk2,ck3,ck4,c5) VALUES ('a','b','c3','d5','e5');
INSERT INTO query_test (pk1,pk2,ck3,ck4,c5) VALUES ('f','b','c3','d5','e5');
```

Let's start by querying everything for `pk1`:

```
SELECT * FROM query_test WHERE pk1='a';

InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering a
```

So what happened here? Cassandra is essentially informing us that it cannot ensure that this query will be served by a single node. This is because we have defined `pk1` and `pk2` as a composite partition key. Without both `pk1` and `pk2` specified, a single node containing the requested data cannot be ascertained. However, it does say the following:

```
If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING
```

So let's give that a try:

```
SELECT * FROM query_test
WHERE pk1='a' ALLOW FILTERING;

 pk1 | pk2 | ck3 | ck4 | c5
-----+-----+-----+-----+----
   a |   b |  c3 |  d5 | e5
   a |   b |  c2 |  d2 | e2
   a |   b |  c2 |  d3 | e3
   a |   b |  c2 |  d4 | e4
   a |   b |  c1 |  d1 | e1

(5 rows)
```

That worked. But the bigger question is why? The `ALLOW FILTERING` directive tells Cassandra that it should perform an exhaustive seek of all partitions, looking for data that might match. With a total of six rows in the table, served by a single node cluster, that will still run fast. But in a multi-node cluster, with millions of other rows, that query will likely time out.

So, let's try that query again, and this time we'll specify the complete partition key, as well as the first clustering key:

```
SELECT * FROM query_test
WHERE pk1='a' AND pk2='b' AND ck3='c2';

 pk1 | pk2 | ck3 | ck4 | c5
-----+-----+-----+-----+----
   a |   b |  c2 |  d2 | e2
   a |   b |  c2 |  d3 | e3
   a |   b |  c2 |  d4 | e4

(3 rows)
```

That works. So what if we just want to query for a specific `ck4`, but we don't know which `ck3` it's under? Let's try skipping `ck3`:

```
SELECT * FROM query_test
WHERE pk1='a' AND pk2='b' AND ck4='d2';

InvalidRequest: Error from server: code=2200 [Invalid query] message="PRIMARY KEY column "ck4" cannot be restricted as preceding col
```

Remember, components of the `PRIMARY KEY` definition can be omitted, as long as (some of) the preceding keys are specified. But they can only be omitted in order. You can't pick and choose which ones to leave out.

So how do we solve this issue? Let's use `ALLOW FILTERING`:

```
SELECT * FROM query_test
WHERE pk1='a' AND pk2='b' AND ck4='d2' ALLOW FILTERING;

 pk1 | pk2 | ck3 | ck4 | c5
-----+-----+-----+-----+----
   a |   b |  c2 |  d2 | e2

(1 rows)
```

That works. But given what we know about the `ALLOW FILTERING` directive, is this OK to do? The answer to this question lies in the fact that we have indeed specified the complete partition key. By doing that, Cassandra knows which node can serve the query. While this may not follow the advice of *build your tables to support your queries*, it may actually perform well (depending on the size of the result set).

Note

Avoid using `ALLOW FILTERING`. It might make certain ad-hoc queries work, but improper use of it may cause your nodes to work too hard, and whichever is chosen as the coordinator may crash. Definitely do not deploy any production code that regularly uses CQL queries containing the `ALLOW FILTERING` directive.

For a quick detour, what if I wanted to know what time it was on my Apache Cassandra cluster? I could query my table using the `now()` function:

```
SELECT now() FROM query_test;

 system.now()
--------------------------------------
f83015b0-8fba-11e8-91d4-a7c67cc60e89
f83015b1-8fba-11e8-91d4-a7c67cc60e89
f83015b2-8fba-11e8-91d4-a7c67cc60e89
f83015b3-8fba-11e8-91d4-a7c67cc60e89
f83015b4-8fba-11e8-91d4-a7c67cc60e89
f83015b5-8fba-11e8-91d4-a7c67cc60e89

(6 rows)
```

What happened here? The `now()` function was invoked for each row in the table. First of all, this is an unbound query and will hit multiple nodes for data that it's not even using. Secondly, we just need one result. And third, returning the current time as `TIMEUUID` isn't very easy to read.

Let's solve problems one and two by changing the table we're querying. Let's try that on the `system.local` table:

```
SELECT now() FROM system.local ;

 system.now()
--------------------------------------
 94a88ad0-8fbb-11e8-91d4-a7c67cc60e89

(1 rows)
```

The `system.local` table is unique to each node in a Cassandra cluster. Also, it only ever has one row in it. So, this query will be served by one node, and there will only be one row returned. But how can we make that more easier to read? We can use the `dateof()` function for this:

```
SELECT dateof(now()) FROM system.local;

 system.dateof(system.now())
---------------------------------
 2018-07-25 03:37:08.045000+0000

(1 rows)
```

Cassandra has other built-in functions that can help to solve other problems. We will cover those later.

Note

You can execute `SELECT CAST(now() as TIMESTAMP) FROM system.local;` to achieve the same result.

The IN operator

So we've seen that CQL has an `AND` keyword for specifying multiple filters in the `WHERE` clause. Does it also have an `OR` keyword, like SQL?

No, it does not. This is because Apache Cassandra is designed to serve sequential reads, not random reads. It works best when its queries give it a clear, precise path to the requested data. Allowing filters in the `WHERE` clause to be specified on an `OR` basis would force Cassandra to perform random reads, which really works against how it was built.

However, queries can be made to perform similarly to `OR`, via the `IN` operator:

```
SELECT * FROM query_test WHERE pk1='a' AND pk2 IN ('b','c');
```

While this query technically will work, its use is considered to be an anti-pattern in Cassandra. This is because it is a multi-key query, meaning the primary key filters are filtering on more than one key value. In this case, Cassandra cannot figure out which node the requested data is on. We are only giving it one part of the partition key. This means that it will have to designate one node as a coordinator node, and then scan each node to build the result set:

```
SELECT * FROM query_test WHERE pk1='a' AND  pk2='b' AND ck3 IN ('c1','c3');
```

This query is also a multi-key query. But, this query will perform better, because we are at least specifying a complete partition key. This way, a token-aware application will not require a coordinator node, and will be able to go directly to the one node that can serve this request.

Do note that, if you use `IN`, the same restrictions apply as for other operators. You cannot skip primary keys, and if you use `IN` on a key, you must do the following:

- Specify all of the keys prior to it
- Use `IN` only on the last key specified in the query

Note

Like its `ALLOW FILTERING` counterpart, `IN` queries can still be served by one node if the complete partition key is specified. However, it's a good idea to limit the number of values specified with the `IN` operator to less than 10.