Creating an index

Cassandra comes with the ability to apply distributed, secondary indexes on arbitrary columns in a table. For an application of this, let's look at a different solution for our `order_status` table. For starters, we'll add the date/time of the order as a clustering column. Next, we'll store the total as a `BIGINT` representing the number of cents (instead of `DECIMAL` for dollars), to ensure that we maintain our precision accuracy. But the biggest difference is that, in talking with our business resources, we will discover that bucketing by week will give us a partition of manageable size:

```
CREATE TABLE order_status_by_week (
   week_bucket bigint,
   order_datetime timestamp,
   order_id uuid,
   shipping_weight_kg decimal,
   status text,
   total bigint,
   PRIMARY KEY (week_bucket, order_datetime, order_id)
) WITH CLUSTERING ORDER BY (order_datetime DESC, order_id ASC)
```

Next, we will add similar rows into this table:

```
 INSERT INTO order_status_by_week (status,order_id,total,week_bucket,order_datetime)
VALUES ('PENDING',UUID(),11422,20180704,'2018-07-25 15:22:28');
INSERT INTO order_status_by_week (status,order_id,total,week_bucket,order_datetime)
VALUES ('PENDING',UUID(),3312,20180704,'2018-07-27 09:44:18');
INSERT INTO order_status_by_week (status,order_id,total,week_bucket,order_datetime)
VALUES ('PENDING',UUID(),8663,20180704,'2018-07-27 11:33:01');
INSERT INTO order_status_by_week (status,order_id,total,shipping_weight_kg,week_bucket,order_datetime)
VALUES ('PICKED',UUID(),30311,2,20180704,'2018-07-24 16:02:47');
INSERT INTO order_status_by_week (status,order_id,total,shipping_weight_kg,week_bucket,order_datetime)
VALUES ('SHIPPED',UUID(),21899,1.05,20180704,'2018-07-24 13:28:54');
INSERT INTO order_status_by_week (status,order_id,total,shipping_weight_kg,week_bucket,order_datetime)
VALUES ('SHIPPED',UUID(),17708,1.2,20180704,'2018-07-25 08:02:29');
```

Now I can query for orders placed during the fourth week of July, 2018:

```
 SELECT * FROM order_status_by_week WHERE week_bucket=20180704;

week_bucket | order_datetime                  | order_id | shipping_weight_kg | status  | total
-------------+---------------------------------+--------------------------------------+--------------------+---------+-------
   20180704 | 2018-07-27 16:33:01.000000+0000 | 02d3af90-f315-41d9-ab59-4c69884925b9 |               null | PENDING | 8663
   20180704 | 2018-07-27 14:44:18.000000+0000 | cb210378-752f-4a6b-bd2c-6d41afd4e614 |               null | PENDING | 3312
   20180704 | 2018-07-25 20:22:28.000000+0000 | 59cf4afa-742c-4448-bd99-45c61660aa64 |               null | PENDING | 11422
   20180704 | 2018-07-25 13:02:29.000000+0000 | c5d111b9-d048-4829-a998-1ca51c107a8e |                1.2 | SHIPPED | 17708
   20180704 | 2018-07-24 21:02:47.000000+0000 | b111d1d3-9e54-481e-858e-b56e38a14b57 |                  2 | PICKED  | 30311
   20180704 | 2018-07-24 18:28:54.000000+0000 | c8b3101b-7804-444f-9c4f-65c17ff201f2 |               1.05 | SHIPPED | 21899

(6 rows)
```

This works, but without status as a part of the primary key definition, how can we query for `PENDING` orders? Here is where we will add a secondary index to handle this scenario:

```
CREATE INDEX [index_name] ON [keyspace_name.]<table_name>(<column_name>);
```

Note

You can create an index without a name. Its name will then default to `[table_name]_[column_name]_idx`.

In the following code block, we will create an index, and then show how it is used:

```
CREATE INDEX order_status_idx ON order_status_by_week(status);

SELECT week_bucket,order_datetime,order_id,status,total FROM order_status_by_week
 WHERE week_bucket=20180704 AND status='PENDING';

 week_bucket | order_datetime      | order_id                             | status  | total
-------------+---------------------+--------------------------------------+-----------------
    20180704 | 2018-07-27 16:33:01 | 02d3af90-f315-41d9-ab59-4c69884925b9 | PENDING |  8663
    20180704 | 2018-07-27 14:44:18 | cb210378-752f-4a6b-bd2c-6d41afd4e614 | PENDING |  3312
    20180704 | 2018-07-25 20:22:28 | 59cf4afa-742c-4448-bd99-45c61660aa64 | PENDING | 11422

(3 rows)
```

In this way, we can query on a column that has a more dynamic value. The status of the order can effectively be updated, without having to delete the entire prior row.

Caution with implementing secondary indexes

While secondary indexes seem like a simple solution to add a dynamic querying capability to a Cassandra model, caution needs to be given when addressing their use. Effective, high-performing, distributed database-indexing is a computing problem that has yet to be solved. Proper, well-defined queries based on primary key definitions are high-performing within Apache Cassandra, because they take the underlying storage model into consideration. Secondary indexing actually works against this principle.

Secondary indexes in Apache Cassandra store data in a hidden table (behind the scenes) that only contains lookups for data contained on the current node. Essentially, a secondary index query (which is not also filtered by a partition key) will need to confer with every node in the cluster. This can be problematic with large clusters and potentially lead to query timeouts.

Note

Our preceding example using a secondary index gets around this problem, because our query is also filtering by its partition key. This forces the query to limit itself to a single node.

Cardinality is another problem to consider when building a secondary index in Apache Cassandra. Let's say we created a secondary index on `order_id`, so that we can pull up an individual order if we had to. In that scenario, the high cardinality of `order_id` would essentially mean that we would query every node in the cluster, just to end up reading one partition from one node.

Author (and DataStax Cassandra MVP) Richard Low accurately explains this in his article *The Sweet Spot for Cassandra Secondary Indexing*, when he describes creating an index on a high-cardinality column such as an email address:

> This means only one node (plus replicas) store data for a given email address but all nodes are queried for each lookup. This is wasteful—every node has potentially done a disk seek but we've only got back one partition.

On the flip-side of that coin, consider a secondary index on a low-cardinality column, such as a Boolean. Now consider that the table in question has 20,000,000 rows. With an even distribution, both index entries will each point to 10,000,000 rows. That is far too many to be querying at once.

We have established that querying with a secondary index in conjunction with a partition key can perform well. But is there a time when querying only by a secondary index could be efficient? Once again, Low's article concludes the following:

> *...the best use case for Cassandra's secondary indexes is when p is approximately n; i.e. the number of partitions is about equal to the number of nodes. Any fewer partitions and your n index lookups are wasted; many more partitions and each node is doing many seeks. In practice, this means indexing is most useful for returning tens, maybe hundreds of results.*

In conclusion, secondary indexing can help with large solutions at scale under certain conditions. But when used alone, the trade-off is usually one of giving up performance in exchange for convenience. The number of nodes in the cluster, total partition keys, and cardinality of the column in question must all be taken into consideration.

Dropping an index

Dropping a secondary index on a table is a simple task:

```
DROP INDEX [index_name]
```

If you do not know the name of the index (or created it without a name), you can describe the table to find it. Indexes on a table will appear at the bottom of the definition. Then you can `DROP` it:

```
 CREATE INDEX ON query_test(c5);
DESC TABLE query_test ;

CREATE TABLE fenago_ch3.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)
...
   AND speculative_retry = '99PERCENTILE';
CREATE INDEX query_test_c5_idx ON fenago_ch3.query_test (c5);

DROP INDEX query_test_c5_idx ;
```