Writing data

Given the ways in which Apache Cassandra has been shown to handle things such as `INSERT`, `UPDATE`, and `DELETE`, it is important to discuss what they have in common. They all result in writes to the database. Let's take a look at how each one behaves in certain scenarios. Assume that we need to keep track of statuses for orders from an e-commerce website. Consider the following table:

```
CREATE TABLE order_status (
 status TEXT,
 order_id UUID,
 shipping_weight_kg DECIMAL,
 total DECIMAL,
 PRIMARY KEY (status,order_id))
WITH CLUSTERING ORDER BY (order_id DESC);
```

Note

The `order_status` table is for example purposes only, and is intended to be used to show how writes work in Apache Cassandra. I do not recommend building an order-status-tracking system this way.

Inserting data

Let's write some data to that table. To do this, we'll use the `INSERT` statement. With an `INSERT` statement, all `PRIMARY KEY` components must be specified; we will specify `status` and `order_id`. Additionally, every column that you wish to provide a value for must be specified in a parenthesis list, followed by the `VALUES` in their own parenthesis list:

```
INSERT INTO order_status (status,order_id,total) VALUES ('PENDING',UUID(),114.22);
INSERT INTO order_status (status,order_id,total) VALUES ('PENDING',UUID(),33.12);
INSERT INTO order_status (status,order_id,total) VALUES ('PENDING',UUID(),86.63);
INSERT INTO order_status (status,order_id,total,shipping_weight_kg)
  VALUES ('PICKED',UUID(),303.11,2);
INSERT INTO order_status (status,order_id,total,shipping_weight_kg)
  VALUES ('SHIPPED',UUID(),218.99,1.05);
INSERT INTO order_status (status,order_id,total,shipping_weight_kg)
  VALUES ('SHIPPED',UUID(),177.08,1.2);
```

Note

If you're going to need a unique identifier for things such as IDs, the `UUID()` and `TIMEUUID()` functions can be invoked in-line as a part of `INSERT`.

As you can see, not all columns need to be specified. In our business case, assume that we do not know the shipping weight until the order has been `PICKED`. If I query for all orders currently in a `PENDING` status, it shows `shipping_weight_kg` as `null`:

```
SELECT * FROM order_status WHERE status='PENDING';

 status  | order_id                             | shipping_weight_kg | total
---------+--------------------------------------+--------------------+--------
 PENDING | fcb15fc2-feaa-4ba9-a3c6-899d1107cce9 |               null | 114.22
 PENDING | ede8af04-cc66-4b3a-a672-ab1abed64c21 |               null | 86.63
 PENDING | 1da6aef1-bd1e-4222-af01-19d2ab0d8151 |               null | 33.12

(3 rows)
```

Remember, Apache Cassandra does not use `null` in the same way that other databases may. In the case of Cassandra, `null` simply means that the currently-requested column does not contain a value.

Note

Do not literally `INSERT` a null value into a table. Cassandra treats this as a `DELETE`, and writes a tombstone. It's also important to make sure that your application code is also not writing nulls for column values that are not set.

Updating data

So now let's update one of our `PENDING` orders to a status of `PICKED`, and give it a value for shipping weight. We can start by updating our `shipping_weight_kg` for order `fcb15fc2-feaa-4ba9-a3c6-899d1107cce9`, and we'll assume that it is `1.4` kilograms. This can be done in two different ways. Updates and inserts are treated the same in Cassandra, so we could actually update our row with the `INSERT` statement:

```
INSERT INTO order_status (status,order_id,shipping_weight_kg)
  VALUES ('PENDING',fcb15fc2-feaa-4ba9-a3c6-899d1107cce9,1.4);
```

Or, we can also use the `UPDATE` statement that we know from SQL:

```
UPDATE order_status SET shipping_weight_kg=1.4
WHERE status='PENDING'
AND order_id=fcb15fc2-feaa-4ba9-a3c6-899d1107cce9;
```

Either way, we can then query our row and see this result:

```
SELECT * FROM order_status
WHERE status='PENDING'
AND order_id=fcb15fc2-feaa-4ba9-a3c6-899d1107cce9;

 status  | order_id                             | shipping_weight_kg | total
---------+--------------------------------------+--------------------+--------
 PENDING | fcb15fc2-feaa-4ba9-a3c6-899d1107cce9 |                1.4 | 114.22

(1 rows)
```

Ok, so now how do we set the `PENDING` status to `PICKED`? Let's start by trying to `UPDATE` it, as we would in SQL:

```
UPDATE order_status SET status='PICKED'
WHERE order_id='fcb15fc2-feaa-4ba9-a3c6-899d1107cce9';

InvalidRequest: Error from server: code=2200 [Invalid query] message="PRIMARY KEY part status found in SET part"
```

With the `UPDATE` statement in CQL, all the `PRIMARY KEY` components are required to be specified in the `WHERE` clause. So how about we try specifying both `PRIMARY KEY` components in `WHERE`, and both columns with values in `SET`:

```
UPDATE order_status SET shipping_weight_kg=1.4,total=114.22
WHERE status='PICKED'
AND order_id=fcb15fc2-feaa-4ba9-a3c6-899d1107cce9;
```

So that doesn't error out, but did it work? To figure that out, let's run a (bad) query using `ALLOW FILTERING`:

```
SELECT * FROM order_status WHERE order_id=fcb15fc2-feaa-4ba9-a3c6-899d1107cce9 ALLOW FILTERING;

 status  | order_id                             | shipping_weight_kg | total
---------+--------------------------------------+--------------------+--------
 PENDING | fcb15fc2-feaa-4ba9-a3c6-899d1107cce9 |                1.4 | 114.22
  PICKED | fcb15fc2-feaa-4ba9-a3c6-899d1107cce9 |                1.4 | 114.22

(2 rows)
```

So for this order ID, there are now two rows present in our table; not at all what we really wanted to do. Sure, we now have our order in a `PICKED` state, but our `PENDING` row is still out there. Why did this happen?

First of all, with both `INSERT` and `UPDATE` you must specify all of the `PRIMARY KEY` components or the operation will fail. Secondly, primary keys are unique in Cassandra. When used together, they essentially point to the column values we want. But that also means they cannot be updated. The only way to update a `PRIMARY KEY` component is to delete and then rewrite it.

Note

To Cassandra, `INSERT` and `UPDATE` are synonymous. They behave the same, and can mostly be used interchangeably. They both write column values to a specific set of unique keys in the table. You can insert new rows with `UPDATE` and you can update existing rows with `INSERT`.

Deleting data

While deleting data and its associated implications have been discussed, there are times when rows or individual column values may need to be deleted. In our use case, we discussed the difficulties of trying to work with the primary key on something that needs to be dynamic, such as `status`. In our case, we have an extra row for our order that we need to delete:

```
DELETE FROM order_status
WHERE status='PENDING'
AND order_id=fcb15fc2-feaa-4ba9-a3c6-899d1107cce9;
```

As mentioned previously, `DELETE` can also enforce the removal of individual column values:

```
DELETE shipping_weight_kg FROM order_status
WHERE status='PICKED'
AND order_id=99886f63-f271-459d-b0b1-218c09cd05a2;
```

Note

Again, take care when using `DELETE`. Deleting creates tombstones, which can be problematic to both data consistency and query performance.

Similar to the previous write operations, `DELETE` requires a complete primary key. But unlike the other write operations, you do not need to provide all of the clustering keys. In this way, multiple rows in a partition can be deleted with a single command.

Lightweight transactions

One difference between the CQL `INSERT` and `UPDATE` statements is in how they handle lightweight transactions. Lightweight transactions are essentially a way for Apache Cassandra to enforce a sequence of read-and-then-write operations to apply conditional writes.

Lightweight transactions are invokable at query time. Apache Cassandra implements the paxos (consensus algorithm) to enforce concurrent lightweight transactions on the same sets of data.

Note

A lightweight transaction in flight will block other lightweight transactions, but will not stop normal reads and writes from querying or mutating the same data.

In any case, an `INSERT` statement can only check whether a row does not already exist for the specified the `PRIMARY KEY` components. If we consider our attempts to insert a new row to set our order status to `PENDING`, this could have been used:

```
INSERT INTO order_status (status,order_id,shipping_weight_kg)
VALUES ('PENDING',fcb15fc2-feaa-4ba9-a3c6-899d1107cce9,1.4)
IF NOT EXISTS;
```

Essentially what is happening here is that Cassandra is performing a read to verify the existence of a row with the specified keys. If that row exists, the operation does not proceed, and a response consisting of applied with a value of `false` (along with the column values which failed to write) is returned. If it succeeds, an applied value of `true` is returned.

On the other hand, `UPDATE` allows for more granular control in terms of lightweight transactions. It allows for the use of both `IF EXISTS` and `IF NOT EXISTS`. Additionally, it can determine whether a write should occur based on arbitrary column values. In our previous example, we could make our update to `shipping_weight_kg` and order total based on a threshold for `shipping_weight_kg`:

```
UPDATE order_status SET shipping_weight_kg=1.4,total=114.22
WHERE status='PICKED' AND order_id=fcb15fc2-feaa-4ba9-a3c6-899d1107cce9
IF shipping_weight_kg > 1.0;
```

Deletes can also make use of lightweight transactions, much in the same way that updates do:

```
DELETE FROM order_status
WHERE status='PENDING'
AND order_id=fcb15fc2-feaa-4ba9-a3c6-899d1107cce9
IF EXISTS;
```

Note

Lightweight transactions do incur a performance penalty, so use them sparingly. However using them with `DELETE` is probably the best use case, as the performance hit is preferable to generating many needless tombstones.