PARTITION KEY VS COMPOSITE KEY VS CLUSTERING COLUMNS IN CASSANDRA

Here we explain the differences between partition key, composite key and clustering key in Cassandra.

In brief, each table requires a unique **primary key**. The first field listed is the **partition key**, since its hashed value is used to determine the node to store the data. If those fields are wrapped in parentheses then the partition key is **composite**. Otherwise the first field is the partition key. Any fields listed after the partition key are called **clustering columns**. These store data in ascending or descending order within the partition for the fast retrieval of similar values. All the fields together are the primary key.

We discussed partitioning data here.

Clustering columns

Clustering columns determines the order of data in partitions.

What is the reason for having clustering columns? The whole point of a **column-oriented database** like Cassandra is to put adjacent data records next to each other for fast retrieval.

Remember than in a regular rdbms database, like Oracle, each row stores all values, including empty ones. But in a column oriented database one row can have columns (a,b,c) and another (a,b) or just (a). So if we are only interested in the value a then why not store that in the same data center, rack, or drive for fast retrieval? Remember that SQL select statements create subsets. So the column-oriented approach makes the prime data structure a type of subset.

This approach makes logical sense since we are usually only interested in a part of the data at any one time. For example why retrieve employee tax IDs, salary, manager's name, when we just want their name and phone number?

Examples

Let's look at books. The ISBN is a serial number of a book used by publishers. They are supposed to be unique. But let's suppose they do not need to be for these examples.

Create a books keyspace, table, and put some data into it. (We discussed keyspaces here.)

Note that the primary key is **PRIMARY KEY (isbn, author, publisher)**. In this case isbn is the partition key and author and publisher are clustering keys. It would make sense that in a collection of books you would want to store them by author and then publisher.

```
CREATE KEYSPACE books
       WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy',
'datacenter1' : 1};

CREATE TABLE books (
   isbn text,
   title text,
   author text,
  publisher text,
   category text,
   PRIMARY KEY (isbn, author, publisher)
);

insert into books (isbn, title, author, publisher, category) values ('111',
'Fishing', 'Fred', 'Penguin Group', 'Sports');

insert into books (isbn, title, author, publisher, category) values ('111',
'Sailing', 'Sally', 'Penguin Group', 'Sports');

insert into books (isbn, title, author, publisher, category) values ('111',
'Archery', 'Fred', 'Penguin Group', 'Sports');

insert into books (isbn, title, author, publisher, category) values ('111',
'Boating', 'Joe', 'Penguin Group', 'Sports');
```

Notice that all of the values in the primary key must be unique, so it dropped one record because author Fred wrote and published more than one book with published Penguin Group.

Now select the partition key and the primary key. They are all the same since we want them all stored on the same virtual node.

```
select token(isbn), isbn from books;

 system.token(isbn)  | isbn
---------------------+------
```

```
 -175843201295106731 |  111
 -175843201295106731 |  111
 -175843201295106731 |  111
```

Now select all records and notices that the data is sorted by author and then publisher within the partition key 111.

```
select * from books;

 isbn | author | publisher     | category | title
------+--------+---------------+----------+---------
  111 |   Fred | Penguin Group |   Sports | Archery
  111 |    Joe | Penguin Group |   Sports | Boating
  111 |  Sally | Penguin Group |   Sports | Sailing
```

Now add another record but give it a different primary key value, which could result it in being stored in a different partition. What virtual node it is stored on depends on the token range assigned to the virtual node.

```
insert into books (isbn, title, author, publisher, category) values ('333',
'Trees', 'Charles Darwin', 'Hachette', 'Nature');


insert into books (isbn, title, author, publisher, category) values ('333',
'Trees', 'Charles Darwin', 'Amazon', 'Nature');
```

Observe again that the data is sorted on the cluster columns author and publisher. And the token is different for the 333 primary key value.

Recall that the partitioner has function configured in cassandra.yaml calculated the hash value and then distributes the data based upon **partitioner**. The default is **org.apache.cassandra.dht.Murmur3Partitioner**

```
select token(isbn), isbn, author, publisher from books;

 system.token(isbn)  | isbn | author          | publisher
---------------------+------+-----------------+---------------
 -175843201295106731 |  111 |            Fred | Penguin Group
 -175843201295106731 |  111 |             Joe | Penguin Group
 -175843201295106731 |  111 |           Sally | Penguin Group
 7036029452358700311 |  333 | Charles Darwin |        Amazon
 7036029452358700311 |  333 | Charles Darwin |      Hachette
```

Composite Keys

This is just a table with more than one column used in the calculation of the partition key. We denote that with parentheses like this: **PRIMARY KEY ((isbn, author), publisher)**. In this case isbn and author are the partition key and publisher is a clustering key.

```
drop table books;
```

```
CREATE TABLE books (
   isbn text,
   title text,
   author text,
  publisher text,
   category text,
   PRIMARY KEY ((isbn, author), publisher)
);

insert into books (isbn, title, author, publisher, category) values ('111',
'Fishing', 'Fred', 'Penguin Group', 'Sports');

insert into books (isbn, title, author, publisher, category) values ('111',
'Sailing', 'Sally', 'Penguin Group', 'Sports');

insert into books (isbn, title, author, publisher, category) values ('111',
'Archery', 'Fred', 'Penguin Group', 'Sports');

insert into books (isbn, title, author, publisher, category) values ('111',
'Boating', 'Joe', 'Penguin Group', 'Sports');
```

Now to show the partition key value we use the SQL **token** function and give it both the isbn and author values:

```
select token(isbn,author), isbn from books;

 system.token(isbn, author) | isbn
----------------------------+------
         725505645253967381 |  111
         960809148155353310 |  111
        5462216525918432145 |  111
```

Add the same data as above with the insert SQL statements. Notice that adding this data also drops one book because one author wrote more than one book with the same ISBN. By definition the primary key must be unique. That includes clustering columns, since they are part of the primary key. All we have changed with the compound key is the calculation of the partition key and thus where the data is stored.

```
select * from books;

 isbn | author | publisher     | category | title
------+--------+---------------+----------+---------
  111 |   Fred | Penguin Group |   Sports | Archery
  111 |    Joe | Penguin Group |   Sports | Boating
  111 |  Sally | Penguin Group |   Sports | Sailing
```