Lab 4. Indexes

There is no doubt that Cassandra can store a gigantic volume of data effortlessly. However, if we cannot efficiently look for what we want in such a data abyss, it is meaningless. Cassandra provides very good support to search and retrieve the desired data by the primary index and secondary index.

In this lab, we will look at how Cassandra uses the primary index and the secondary index to spotlight the data. After developing an understanding of them, we can then design a high-performance data model.

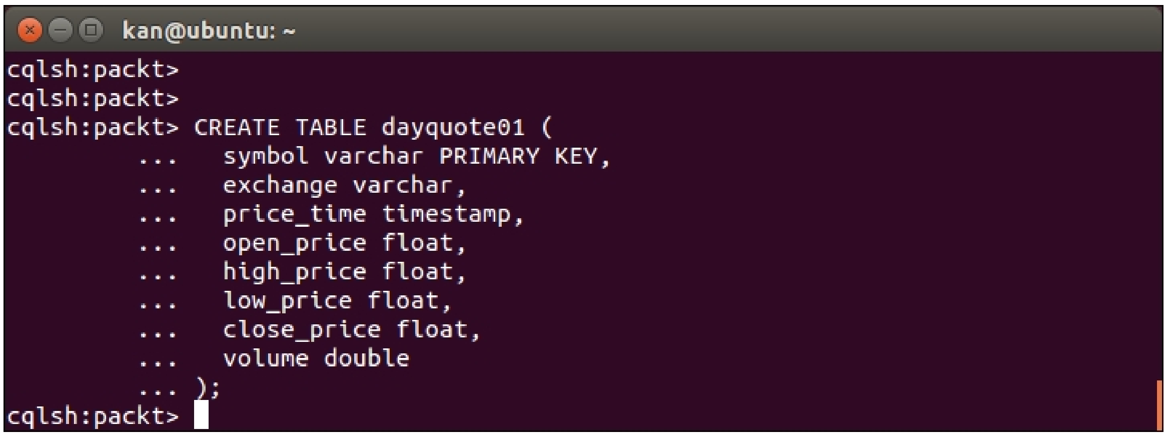
Note: Directory "lab_04" contains all examples for this lab. Open vscode to see all the files.

Primary index

Cassandra is a column-based database. Each row can have different number of columns. A cell is the placeholder of the value and the timestamp data is identified by a row and column. Each cell can store values that are less than 2 GB. The rows are grouped by partitions. The maximum number of cells per partition is limited to the condition that the number of rows times the number of columns is less than 2 billion. Each row is identified by a row key that determines which machine stores the row. In other words, the row key determines the node location of the row. A list of row keys of a table is known as a primary key. A primary index is just created on the primary key.

A primary key can be defined on a single column or multiple columns. In either case, the first component of a table's primary key is the partition key. Each node stores a data partition of the table and maintains its own primary key for the data that it manages. Therefore, each node knows what ranges of row key it can manage and the rows can then be located by scanning the row indexes only on the relevant replicas. The range of the primary keys that a node manages is determined by the partition key and a cluster-wide configuration parameter called partitioner. Cassandra provides three choices to partitioner that will be covered later in this lab.

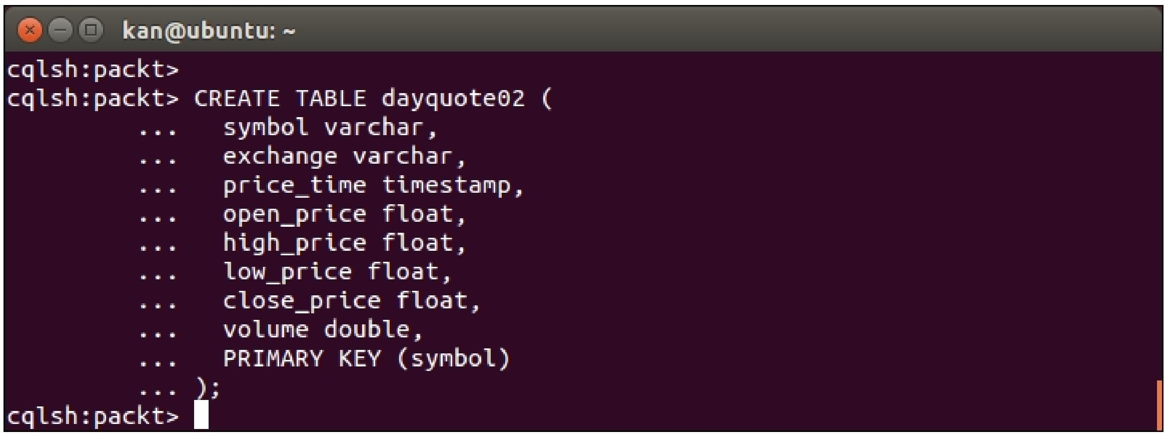
A primary key can be defined by the CQL keywords `PRIMARY KEY`, with the column(s) to be indexed. Imagine that we want to store the daily stock quotes into a Cassandra table called `dayquote01`. The `CREATE TABLE` statement creates a table with a simple primary key that involves only one column, as shown in the following screenshot:



```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote01 (
         ...   symbol varchar PRIMARY KEY,
         ...   exchange varchar,
         ...   price_time timestamp,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double
         ... );
cqlsh:packt>
```

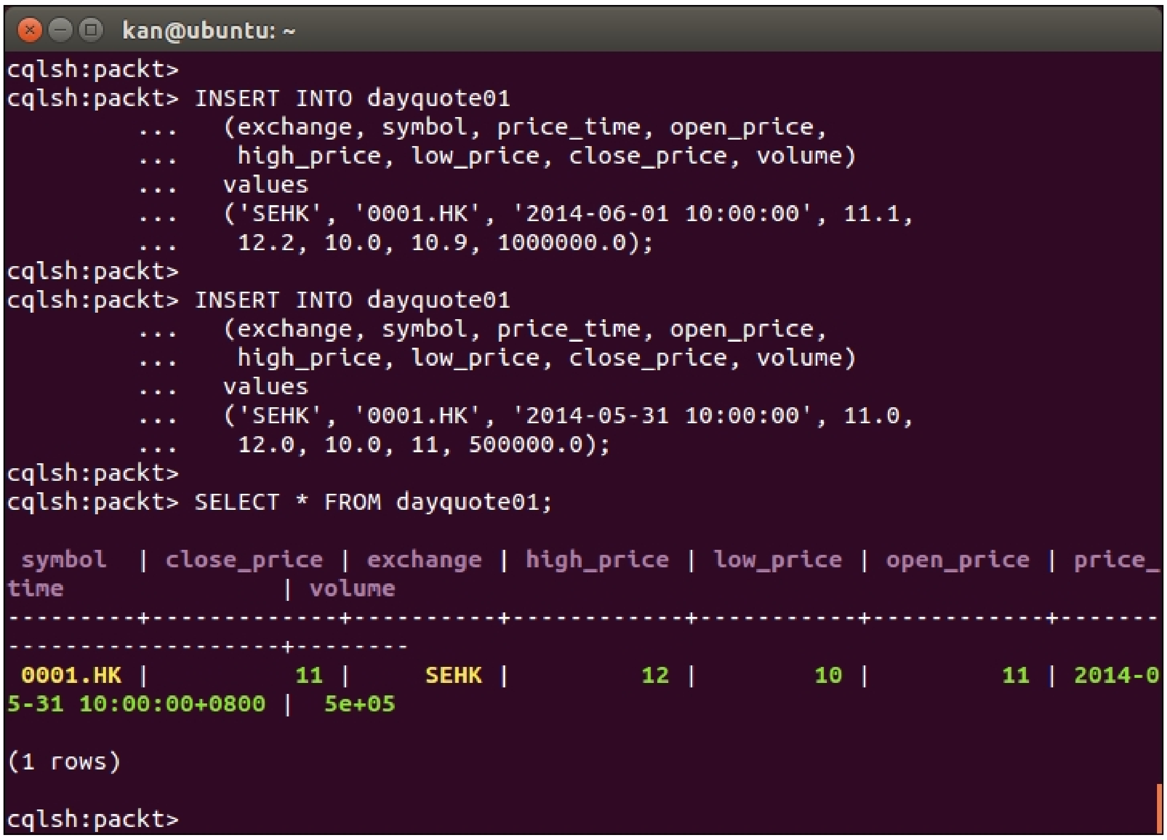
The `symbol` field is assigned the primary key of the `dayquote01` table. This means that all the rows of the same symbol are stored on the same node. Hence, this makes the retrieval of these rows very efficient.

Alternatively, the primary key can be defined by an explicit `PRIMARY KEY` clause, as shown in the following screenshot:



```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote02 (
         ...   symbol varchar,
         ...   exchange varchar,
         ...   price_time timestamp,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double,
         ...   PRIMARY KEY (symbol)
         ... );
cqlsh:packt>
```

Unlike relational databases, Cassandra does not enforce a unique constraint on the primary key, as there is no *primary key violation* in Cassandra. An `INSERT` statement using an existing row key is allowed. Therefore, in CQL, `INSERT` and `UPDATE` act in the same way, which is known as **UPSERT**. For example, we can insert two records into the table `dayquote01` with the same symbol and no primary key violation is alerted, as shown in the following screenshot:



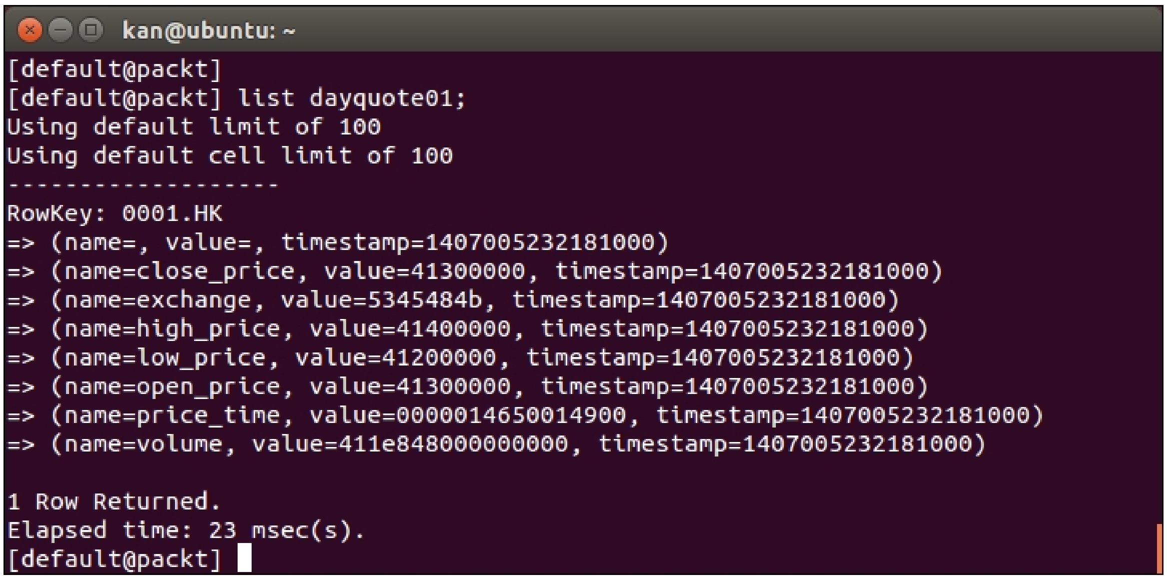
```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote01
         ...   (exchange, symbol, price_time, open_price,
         ...    high_price, low_price, close_price, volume)
         ...   values
         ...   ('SEHK', '0001.HK', '2014-06-01 10:00:00', 11.1,
         ...    12.2, 10.0, 10.9, 1000000.0);
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote01
         ...   (exchange, symbol, price_time, open_price,
         ...    high_price, low_price, close_price, volume)
         ...   values
         ...   ('SEHK', '0001.HK', '2014-05-31 10:00:00', 11.0,
         ...    12.0, 10.0, 11, 500000.0);
cqlsh:packt>
cqlsh:packt> SELECT * FROM dayquote01;

 symbol  | close_price | exchange | high_price | low_price | open_price | price_
time                     | volume
---------+-------------+----------+------------+-----------+------------+-------
-------------------+--------
 0001.HK |          11 |     SEHK |         12 |        10 |         11 | 2014-0
5-31 10:00:00+0800 |  5e+05

(1 rows)

cqlsh:packt>
```

The returned query result contains only one row, not two rows as expected. This is because the primary key is the symbol and the row in the latter `INSERT` statement overrode the record that was created by the former `INSERT` statement. There is no warning for a duplicate primary key. Cassandra simply and quietly updated the row. This silent UPSERT behavior might sometimes cause undesirable effects in the application logic.

Tip

Hence, it is very important for an application developer to handle duplicate primary key situations in the application logic. Do not rely on Cassandra to check the uniqueness for you.

In fact, the reason why Cassandra behaves like this becomes more clear when we know how the internal storage engine stores the row, as shown by Cassandra CLI in the following screenshot:



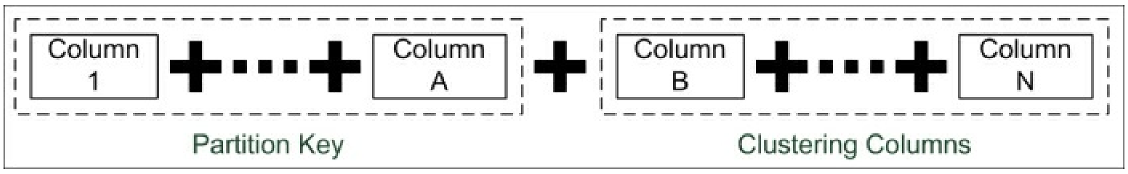
```
kan@ubuntu: ~
[default@packt]
[default@packt] list dayquote01;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: 0001.HK
=> (name=, value=, timestamp=1407005232181000)
=> (name=close_price, value=41300000, timestamp=1407005232181000)
=> (name=exchange, value=5345484b, timestamp=1407005232181000)
=> (name=high_price, value=41400000, timestamp=1407005232181000)
=> (name=low_price, value=41200000, timestamp=1407005232181000)
=> (name=open_price, value=41300000, timestamp=1407005232181000)
=> (name=price_time, value=0000014650014900, timestamp=1407005232181000)
=> (name=volume, value=411e848000000000, timestamp=1407005232181000)

1 Row Returned.
Elapsed time: 23 msec(s).
[default@packt]
```

The row key is `0001.HK`. It is used to locate which node is used to store the row. Whenever we insert or update the row of the same row key, Cassandra blindly locates the row and modifies the columns accordingly, even though an `INSERT` statement has been used.

Although a single column primary key is not uncommon, a primary key composed of more than one column is much more practical.

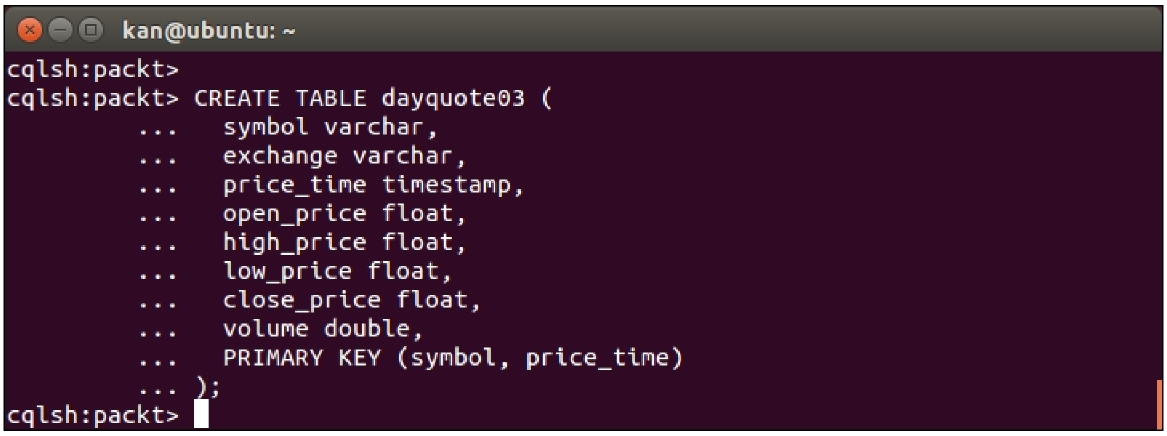
Compound primary key and composite partition key

A compound primary key is composed of more than one column. The order of the columns is important. The structure of a compound primary key is depicted in the following figure:



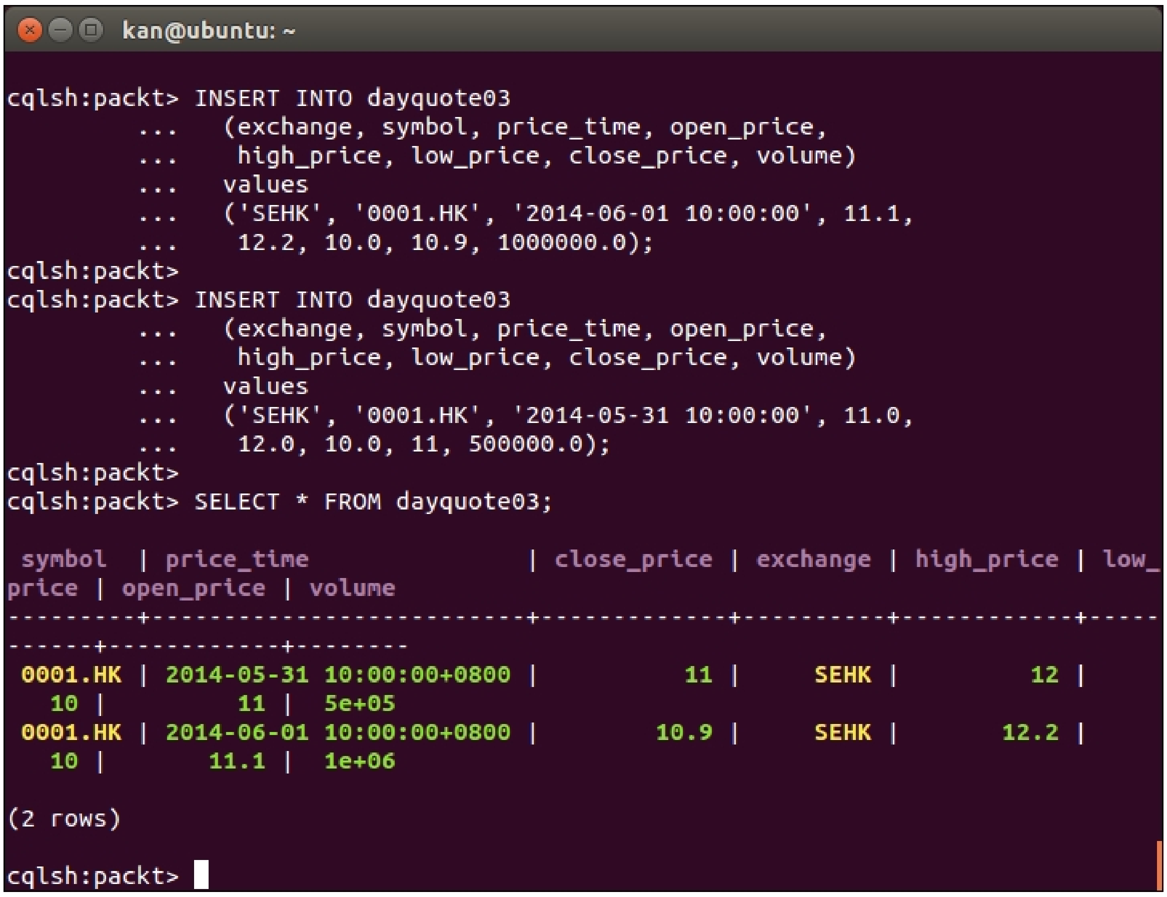
Columns 1 to A are used as the partition key for Cassandra to determine the node location for the partition. The remaining columns, columns B to N, are referred to as the clustering columns for the ordering of data. The clustering columns are used to locate a unique record in the data node. They are ordered, by default, and have the ability to use the `ORDER BY [DESC]` clause in the `SELECT` statements. Moreover, we can get the `MIN` or `MAX` values for clustering keys with the `LIMIT 1` clause. We also need to use the clustering columns for the predicates in a `WHERE` clause. We cannot leave out one when trying to build a query.

To define a compound primary key, an explicit `PRIMARY KEY` clause must be used in the `CREATE TABLE` or `ALTER TABLE` statements. We can define a compound primary key for the table `dayquote03` , as shown in the following screenshot:

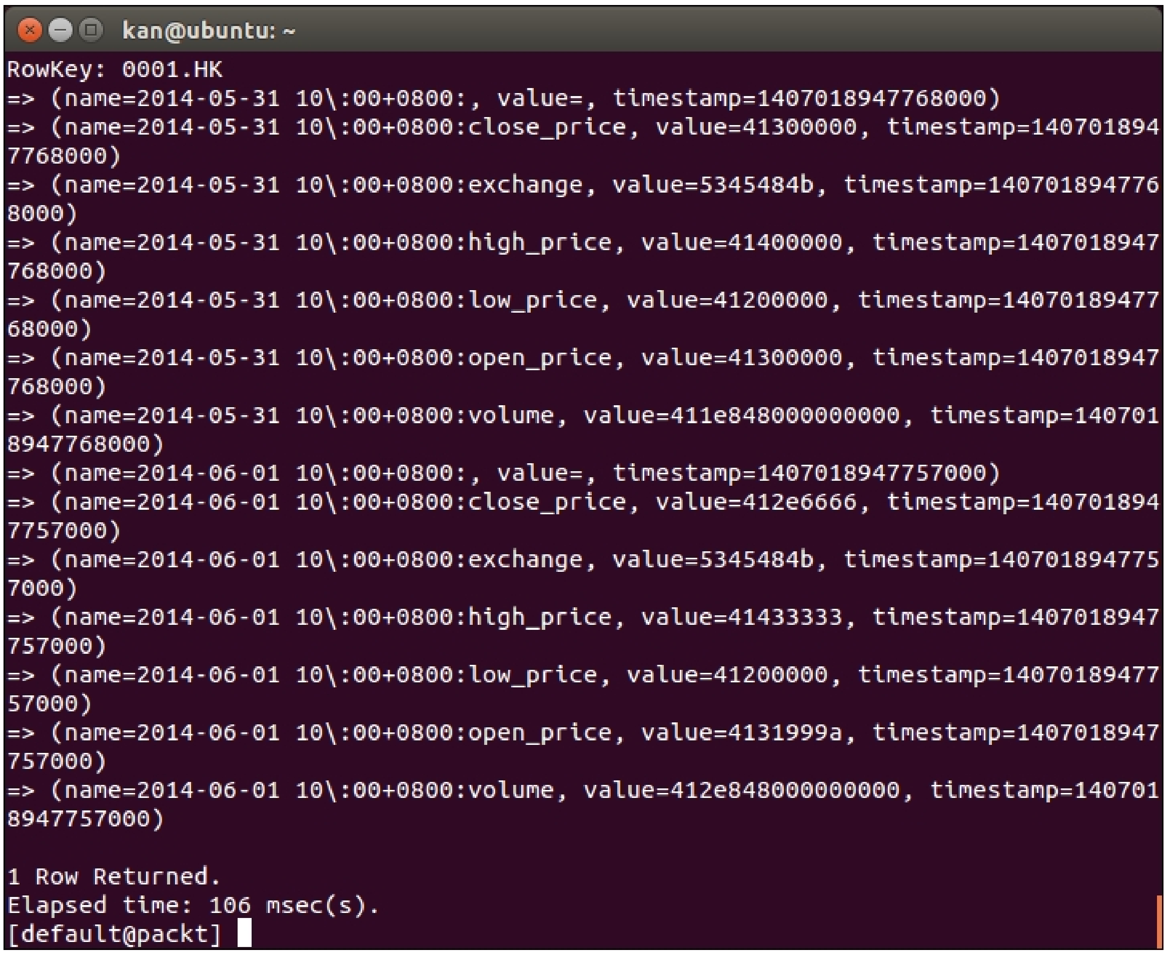

```
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote03 (
         ...   symbol varchar,
         ...   exchange varchar,
         ...   price_time timestamp,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double,
         ...   PRIMARY KEY (symbol, price_time)
         ... );
cqlsh:packt>
```

Because the first part of the primary key (that is `symbol`) is the same as that of the simple primary key, the partition key is the same as that in `dayquote01` . Therefore, the node location is the same regardless of whether the primary key is compound or not, as in this case.

So, what is difference between the simple primary key (`symbol`) and this compound one (`symbol, price_time`)? The additional field `price_time` instructs Cassandra to guarantee the clustering or ordering of the rows within the partition by the values of `price_time` . Thus, the compound primary key sorts the rows of the same symbol by `price_time` . We insert two records into the `dayquote03` table and select all the records to see the effect, as shown in the following screenshot:


```

cqlsh:packt> INSERT INTO dayquote03
         ...   (exchange, symbol, price_time, open_price,
         ...    high_price, low_price, close_price, volume)
         ...   values
         ...   ('SEHK', '0001.HK', '2014-06-01 10:00:00', 11.1,
         ...    12.2, 10.0, 10.9, 1000000.0);
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote03
         ...   (exchange, symbol, price_time, open_price,
         ...    high_price, low_price, close_price, volume)
         ...   values
         ...   ('SEHK', '0001.HK', '2014-05-31 10:00:00', 11.0,
         ...    12.0, 10.0, 11, 500000.0);
cqlsh:packt>
cqlsh:packt> SELECT * FROM dayquote03;

 symbol  | price_time               | close_price | exchange | high_price | low_
price | open_price | volume
---------+--------------------------+-------------+----------+------------+-----
------+------------+--------
 0001.HK | 2014-05-31 10:00:00+0800 |          11 |     SEHK |         12 |
   10 |         11 |  5e+05
 0001.HK | 2014-06-01 10:00:00+0800 |        10.9 |     SEHK |       12.2 |
   10 |       11.1 |  1e+06

(2 rows)

cqlsh:packt>
```

Two records are returned as expected (compared to only one record in `dayquote01`). Moreover, the ordering of the results is sorted by the values of `price_time` . The following screenshot shows the internal view of the rows in the `dayquote03` table:

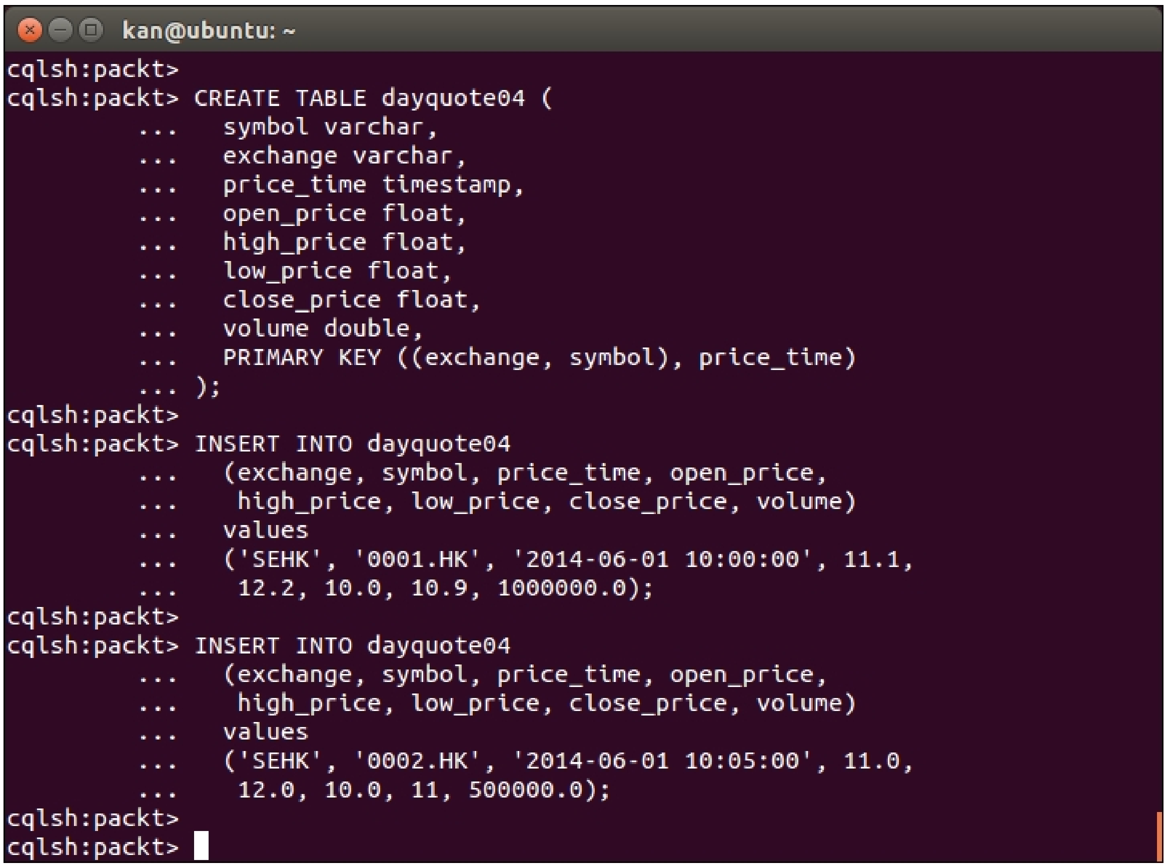


```
kan@ubuntu: ~
RowKey: 0001.HK
=> (name=2014-05-31 10\:00+0800:, value=, timestamp=1407018947768000)
=> (name=2014-05-31 10\:00+0800:close_price, value=41300000, timestamp=140701894
7768000)
=> (name=2014-05-31 10\:00+0800:exchange, value=5345484b, timestamp=140701894776
8000)
=> (name=2014-05-31 10\:00+0800:high_price, value=41400000, timestamp=1407018947
768000)
=> (name=2014-05-31 10\:00+0800:low_price, value=41200000, timestamp=14070189477
68000)
=> (name=2014-05-31 10\:00+0800:open_price, value=41300000, timestamp=1407018947
768000)
=> (name=2014-05-31 10\:00+0800:volume, value=411e848000000000, timestamp=140701
8947768000)
=> (name=2014-06-01 10\:00+0800:, value=, timestamp=1407018947757000)
=> (name=2014-06-01 10\:00+0800:close_price, value=412e6666, timestamp=140701894
7757000)
=> (name=2014-06-01 10\:00+0800:exchange, value=5345484b, timestamp=140701894775
7000)
=> (name=2014-06-01 10\:00+0800:high_price, value=41433333, timestamp=1407018947
757000)
=> (name=2014-06-01 10\:00+0800:low_price, value=41200000, timestamp=14070189477
57000)
=> (name=2014-06-01 10\:00+0800:open_price, value=4131999a, timestamp=1407018947
757000)
=> (name=2014-06-01 10\:00+0800:volume, value=412e848000000000, timestamp=140701
8947757000)

1 Row Returned.
Elapsed time: 106 msec(s).
[default@packt]
```

The row key is still the partition key, that is, `0001.HK`. However, Cassandra stores the two rows returned by the CQL `SELECT` statement, as one single internal row in its storage. The values of the clustering columns are used as a prefix to the columns that are not specified in the `PRIMARY KEY` clause. As Cassandra stores the internal columns in the sorting order of the column name, the rows returned by the CQL `SELECT` statement are sorted inherently. In a nutshell, on a physical node, when the rows for a partition key are stored in the order that is based on the clustering columns, the retrieval of rows is very efficient.

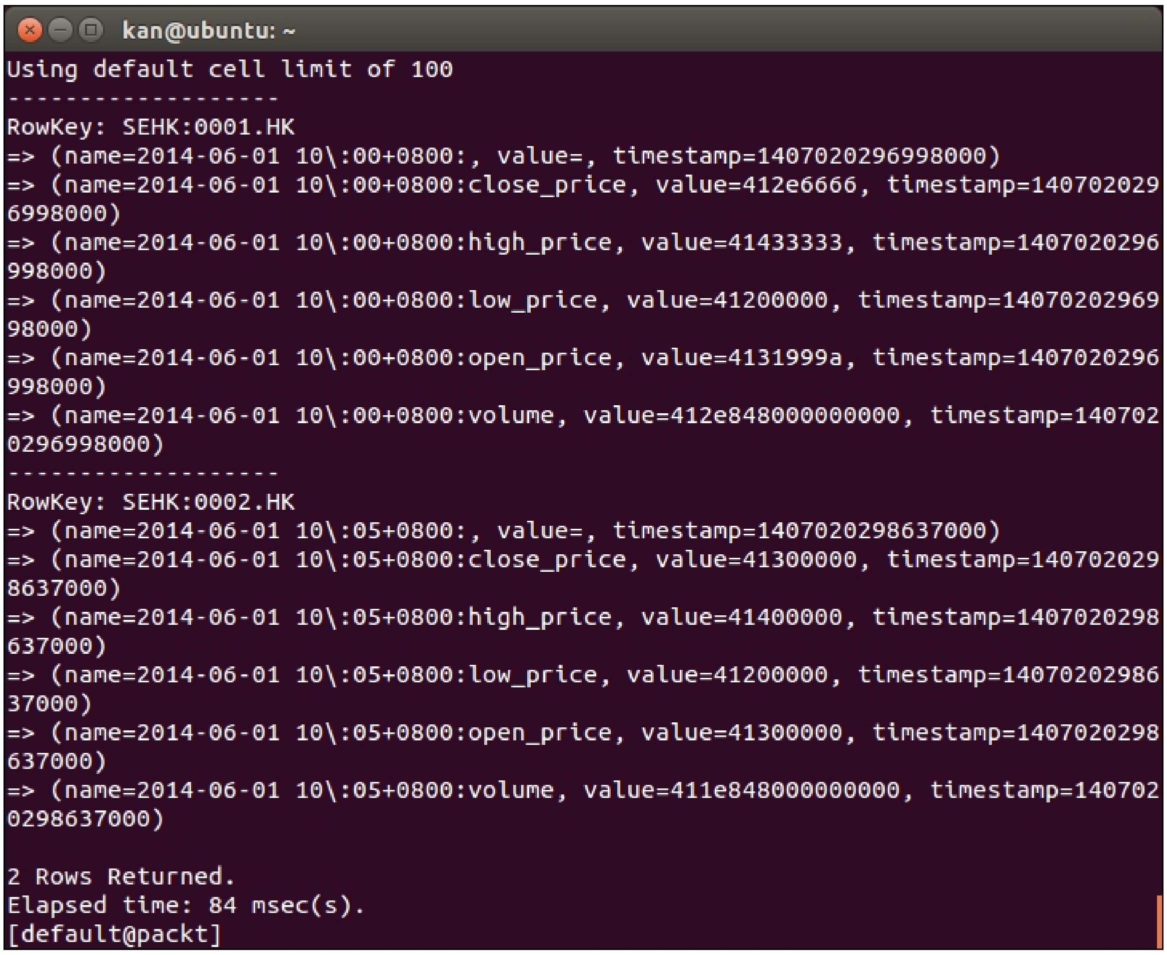
Now you know that the first part of a compound primary key is the partition key. If we need to keep on storing 3,000 daily quotes (around 10 years) for `0001.HK`, although the CQL `SELECT` statement returns 3,000 virtual rows, Cassandra requires to store these 3,000 virtual rows as one entire row on a node by the partition key. The size of the entire row gets bigger and bigger on a node as a result of storing more and more daily quotes. The row will quickly become gigantic over a period of time and will then pose a serious performance problem, as a result of an unbalanced cluster. The solution is a feature offered by Cassandra called composite partition key.

The composite partition key spreads the data over multiple nodes. It is defined by an extra set of parentheses in the `PRIMARY KEY` clause. Let us create another table `dayquote04` with a composite partition key in order to illustrate the effect. The columns `exchange` and `symbol` are now members of a composite partition key, whereas the column `price_time` is a clustering column. We insert the same two records of different symbols into `dayquote04`, as shown in the following screenshot:



```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote04 (
         ...   symbol varchar,
         ...   exchange varchar,
         ...   price_time timestamp,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double,
         ...   PRIMARY KEY ((exchange, symbol), price_time)
         ... );
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote04
         ...   (exchange, symbol, price_time, open_price,
         ...    high_price, low_price, close_price, volume)
         ...   values
         ...   ('SEHK', '0001.HK', '2014-06-01 10:00:00', 11.1,
         ...    12.2, 10.0, 10.9, 1000000.0);
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote04
         ...   (exchange, symbol, price_time, open_price,
         ...    high_price, low_price, close_price, volume)
         ...   values
         ...   ('SEHK', '0002.HK', '2014-06-01 10:05:00', 11.0,
         ...    12.0, 10.0, 11, 500000.0);
cqlsh:packt>
cqlsh:packt>
```

With reference to the following screenshot, two internal rows are returned with their row keys as `SEHK:0001.HK` and `SEHK:0002.HK` , respectively. Internally, Cassandra concatenates the columns in the composite partition key together as an internal row key. In short, the original row without a composite partition key is now split into two rows. As the row keys are now different from each other, the corresponding rows can be stored on different nodes. The value of the clustering column `price_time` is still used as a prefix in the internal column name to preserve the ordering of data:

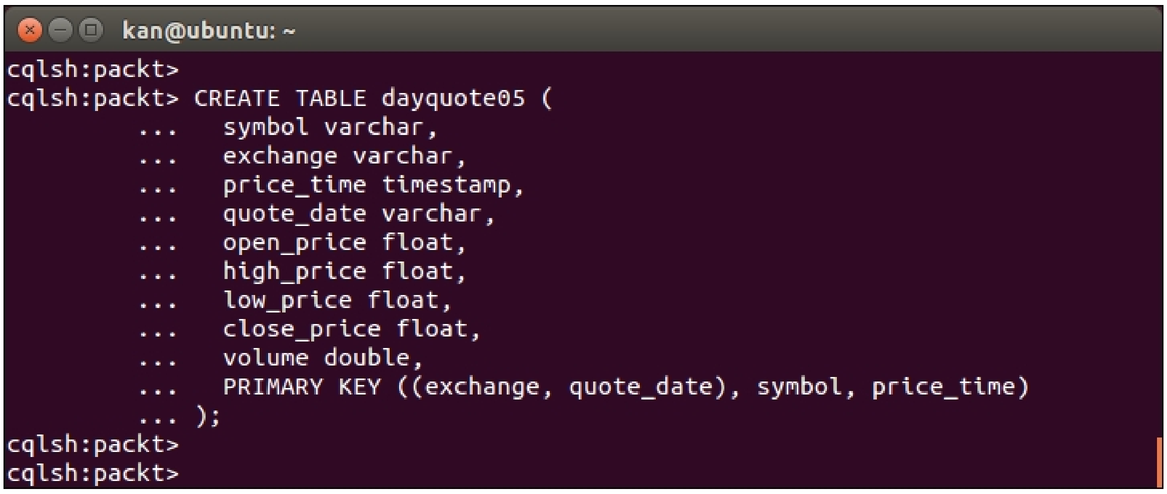


```
kan@ubuntu: ~
Using default cell limit of 100
-------------------
RowKey: SEHK:0001.HK
=> (name=2014-06-01 10\:00+0800:, value=, timestamp=1407020296998000)
=> (name=2014-06-01 10\:00+0800:close_price, value=412e6666, timestamp=140702029
6998000)
=> (name=2014-06-01 10\:00+0800:high_price, value=41433333, timestamp=1407020296
998000)
=> (name=2014-06-01 10\:00+0800:low_price, value=41200000, timestamp=14070202969
98000)
=> (name=2014-06-01 10\:00+0800:open_price, value=4131999a, timestamp=1407020296
998000)
=> (name=2014-06-01 10\:00+0800:volume, value=412e848000000000, timestamp=140702
0296998000)
-------------------
RowKey: SEHK:0002.HK
=> (name=2014-06-01 10\:05+0800:, value=, timestamp=1407020298637000)
=> (name=2014-06-01 10\:05+0800:close_price, value=41300000, timestamp=140702029
8637000)
=> (name=2014-06-01 10\:05+0800:high_price, value=41400000, timestamp=1407020298
637000)
=> (name=2014-06-01 10\:05+0800:low_price, value=41200000, timestamp=14070202986
37000)
=> (name=2014-06-01 10\:05+0800:open_price, value=41300000, timestamp=1407020298
637000)
=> (name=2014-06-01 10\:05+0800:volume, value=411e848000000000, timestamp=140702
0298637000)

2 Rows Returned.
Elapsed time: 84 msec(s).
[default@packt]
```

Time-series data

Cassandra is very suitable for handling time-series type of data, such as web server logfiles, usage data, sensor data, SIP packets, and so on. The tables `dayquote01` to `dayquote04` in the previous sections are used to store the daily stock quotes is an example of the time-series data.

We have just seen in the last section that a composite partition key is a better way of not overwhelming the row. It limits the size of the rows on the basis of a symbol. However, this does partially solve the problem. The size of the row of a symbol still grows over a period of time. Do you have any other suggestion? We can define an artificial column, `quote_date` , in the table and set the composite partition key to `exchange` and `quote_date` instead, as shown in the following screenshot:



```
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote05 (
         ...   symbol varchar,
         ...   exchange varchar,
         ...   price_time timestamp,
         ...   quote_date varchar,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double,
         ...   PRIMARY KEY ((exchange, quote_date), symbol, price_time)
         ... );
cqlsh:packt>
cqlsh:packt>
```

Now the composite partition key limits the size of the rows on a daily basis, and makes the rows more manageable. This way of doing is analogous to inserting the data into different buckets labeled by a particular date. Hence, it is given a name called the **date bucket pattern**. Partitioning by the date also makes table maintenance easier by allowing you to drop the partition of `quote_date` . One drawback of the date bucket pattern is that you always need to know the partition key in order to get the rows. So, in `dayquote05` , you cannot get the latest `quote_date` value using the `ORDER BY DESC` and `LIMIT 1` clauses.

The date bucket pattern gives an application developer a design option to attain a more balanced cluster, but how balanced a cluster is depends on a number of factors in which the most important one is the selection of the partitioner.

Partitioner

A partitioner is basically a hash function used to calculate the `TOKEN()` (the hash value) of a row key and so, it determines how data is distributed across the nodes in a cluster. Choosing a partitioner determines which node is used to place the first copy of data. Each row of data is uniquely identified by a partition key and is distributed across the cluster by the value of the `TOKEN()` . Cassandra provides the following three partitioners:

- `Murmur3Partitioner` (default since version 1.2)
- `RandomPartitioner` (default before version 1.2)
- `ByteOrderedPartitioner`

Murmur3Partitioner

`Murmur3Partitioner` provides faster hashing and improved performance than the partitioner `RandomPartitioner` . It is the default partitioning strategy and the right choice for new clusters in almost all cases. It uses the *MurmurHash* function that creates a 64-bit hash value of the partition key. The possible range of hash values is from -2^63^ to +2^63^ -1. When using `Murmur3Partitioner` , you can page through all the rows using the `TOKEN()` function in a CQL `SELECT` statement.

RandomPartitioner

`RandomPartitioner` was the default partitioner prior to Cassandra Version 1.2. It distributes data evenly across the nodes using an *MD5* hash value of the row key. The possible range of hash values is from 0 to 2^127^ -1. The MD5 hash function is slow in performance, that is why Cassandra has moved to Murmur3 hashes. When using `RandomPartitioner` , you can page through all rows using the `TOKEN()` function in a CQL `SELECT` statement.

ByteOrderedPartitioner

`ByteOrderedPartitioner` , as its name suggests, is used for ordered partitioning. This partitioner orders rows lexically by key bytes. Tokens are calculated by looking at the actual values of the partition key data and using a hexadecimal representation of the leading character(s) in a key. For example, if you wanted to partition rows alphabetically, you can assign a B `TOKEN()` using its hexadecimal representation of `0x42` .

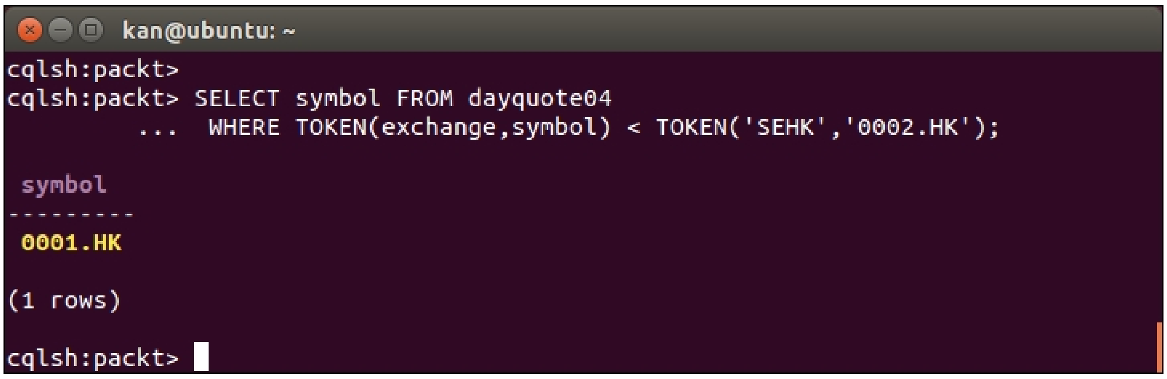
Using `ByteOrderedPartitioner` allows ordered scans by a primary key as though you were moving a cursor through a traditional index in a relational table. This type of range scan query is not possible using `RandomPartitioner` because the keys are stored in the order of their MD5 hash, and not in the sequential order of the keys.

Apparently, performing range-scan on rows sounds like a desirable feature of `ByteOrderedPartitioner` . There are ways to achieve the same functionality using secondary indexes. Conversely, using `ByteOrderedPartitioner` is not recommended for the following reasons:

- **Difficult load balancing**: More administrative overhead is required to load balance the cluster. `ByteOrderedPartitioner` requires administrators to manually calculate partition ranges based on their estimates of the partition key distribution.
- **Sequential writes can cause hot spots**: If the application tends to write or update a sequential block of rows at a time, the writes will not be distributed across the cluster. They all go to one node. This is frequently a problem for applications dealing with timestamped data.
- **Uneven load balancing for multiple tables**: If the application has multiple tables, chances are that these tables have different row keys and different distributions of data. An ordered partitioner that is balanced for one table can cause hot spots and uneven distribution for another table in the same cluster.

Paging and token function

When using the `RandomPartitioner` or `Murmur3Partitioner` , the rows are ordered by the hash of their value. Hence, the order of the rows is not meaningful. Using CQL, the rows can still be paged through even when using `RandomPartitioner` or `Murmur3Partitioner` using the `TOKEN()` function, as shown in the following screenshot:



```
cqlsh:packt>
cqlsh:packt> SELECT symbol FROM dayquote04
         ...  WHERE TOKEN(exchange,symbol) < TOKEN('SEHK','0002.HK');

 symbol
---------
 0001.HK

(1 rows)

cqlsh:packt>
```

`ByteOrderedPartitioner` arranges tokens in the same way as key values, while `RandomPartitioner` and `Murmur3Partitioner` distribute tokens in a completely unordered manner. The `TOKEN()` function makes it possible to page through the unordered partitioner results. It actually queries results directly using tokens.

Secondary indexes

As Cassandra only allows each table to have one primary key, it supports secondary index on columns other than those in the primary key. The benefit is a fast, efficient lookup of data matching the indexed columns in the `WHERE` clause. Each table can have more than one secondary index. Cassandra uses secondary indexes to find the rows that are not using the row key. Behind the scenes, the secondary index is implemented as a separate, hidden table that is maintained automatically by the internal process of Cassandra. As with relational databases, keeping secondary indexes up to date is not free, so unnecessary indexes should be avoided.

Note

the between a primary index and a secondary index is that the primary index is a distributed index used to locate the node that stores the row key, whereas the secondary index is a local index just to index the data on the local node.

Therefore, the secondary index will not be able to know immediately the locations of all matched rows without having examined all the nodes in the cluster. This makes the performance of the secondary index unpredictable.

Note

The secondary index is the most efficient when using equality predicates. This is indeed a limitation that must have at least one equality predicate clause to hopefully limit the set of rows that need to be read into memory.

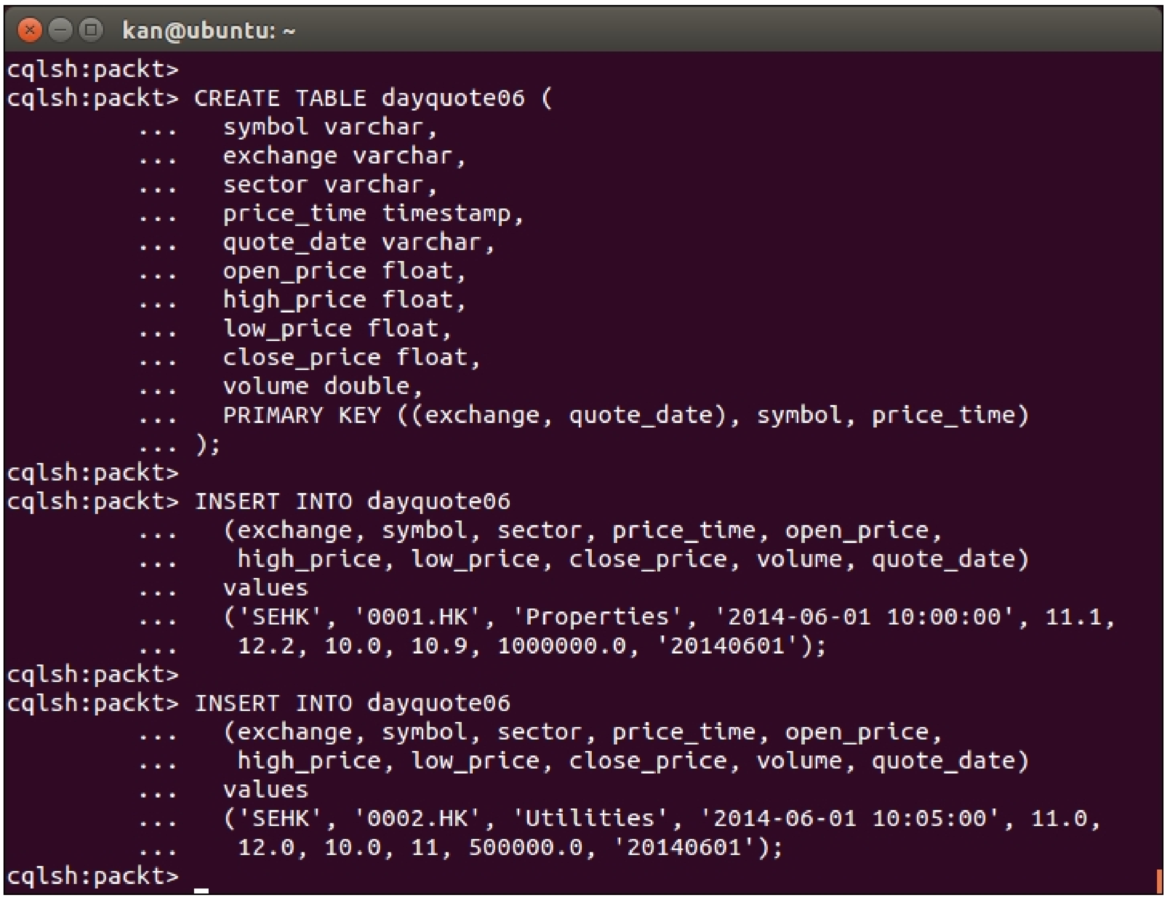
In addition, the secondary index cannot be created on the primary key itself.

Note

Caveat!

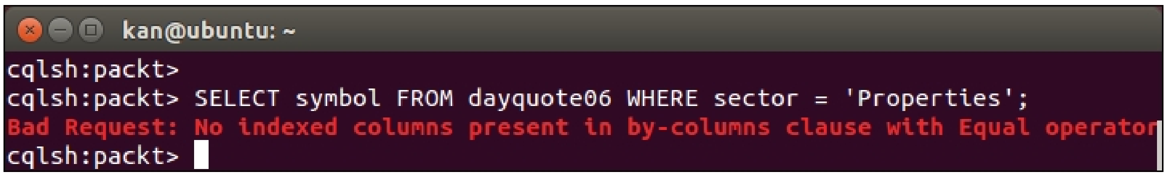
Secondary indexes in Cassandra are *NOT* equivalent to those in the traditional RDBMS. They are not akin to a B-tree index in RDBMS. They are mostly like a hash. So, the range queries do not work on secondary indexes in Cassandra, only equality queries work on secondary indexes.

We can use the CQL `CREATE INDEX` statement to create an index on a column after we define a table. For example, we might want to add a column `sector` to indicate the sector that the stock belongs to, as shown in the following screenshot:



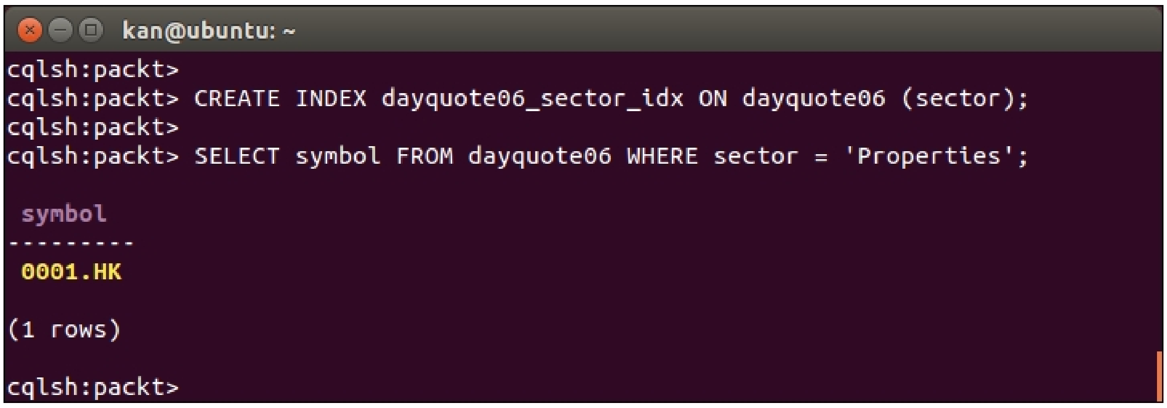
```
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote06 (
         ...   symbol varchar,
         ...   exchange varchar,
         ...   sector varchar,
         ...   price_time timestamp,
         ...   quote_date varchar,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double,
         ...   PRIMARY KEY ((exchange, quote_date), symbol, price_time)
         ... );
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote06
         ...   (exchange, symbol, sector, price_time, open_price,
         ...    high_price, low_price, close_price, volume, quote_date)
         ...   values
         ...   ('SEHK', '0001.HK', 'Properties', '2014-06-01 10:00:00', 11.1,
         ...    12.2, 10.0, 10.9, 1000000.0, '20140601');
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote06
         ...   (exchange, symbol, sector, price_time, open_price,
         ...    high_price, low_price, close_price, volume, quote_date)
         ...   values
         ...   ('SEHK', '0002.HK', 'Utilities', '2014-06-01 10:05:00', 11.0,
         ...    12.0, 10.0, 11, 500000.0, '20140601');
cqlsh:packt> _
```

If we want to search `dayquote06` for symbols that belong to `Properties` , we might run the command, as shown in the following screenshot:



```
cqlsh:packt>
cqlsh:packt> SELECT symbol FROM dayquote06 WHERE sector = 'Properties';
Bad Request: No indexed columns present in by-columns clause with Equal operator
cqlsh:packt>
```

As `sector` is not in the primary key, we cannot query Cassandra directly by `sector` . Instead, we can create a secondary index on the column `sector` to make this possible, as shown in the following screenshot:



```
cqlsh:packt>
cqlsh:packt> CREATE INDEX dayquote06_sector_idx ON dayquote06 (sector);
cqlsh:packt>
cqlsh:packt> SELECT symbol FROM dayquote06 WHERE sector = 'Properties';

 symbol
---------
 0001.HK

(1 rows)

cqlsh:packt>
```

The index name `dayquote06_sector_idx` is optional, but must be unique within the keyspace. Cassandra assigns a name such as `dayquote06_idx` if you do not provide a name. We can now query Cassandra for daily stock quotes by `sector` .

You can see that the columns in the primary key are not present in the `WHERE` predicate clause in the previous screenshot and Cassandra uses the secondary index to look for the rows matching the selection condition.

Multiple secondary indexes

Cassandra supports multiple secondary indexes on a table. The `WHERE` clause is executed if at least one column is involved in a secondary index. Thus, we can use multiple conditions in the `WHERE` clause to filter the results. When multiple occurrences of data match a condition in the `WHERE` predicate clause, Cassandra selects the least frequent occurrence of a condition to process first so as to have a better query efficiency.

When a potentially expensive query is attempted, such as a range query, Cassandra requires the `ALLOW FILTERING` clause, which can apply additional filters to the result set for values of other non-indexed columns. It works very slowly because it scans all rows in all nodes. The `ALLOW FILTERING` clause is used to explicitly direct Cassandra to execute that potentially expensive query on any `WHERE` clause without creating secondary indexes, despite unpredictability of the performance.

Secondary index do's and don'ts

The secondary index is best on a table that has many rows that contain fewer unique values, that is low cardinality in the relational database terminologies, which is counterintuitive to the relational people. The more unique values that exist in a particular column, the more overhead you will have to query and maintain the index. Hence, it is not suitable for querying a huge volume of records for a small number of results.

Tip

Do index the columns with values that have low cardinality. Cassandra stores secondary indexes only for local rows in the data node as a hash-multimap or as bitmap indexes, you can refer to it at https://issues.apache.org/jira/browse/CASSANDRA-1472.

Secondary indexes should be avoided in the following situations:

- On high-cardinality columns for a small number of results out of a huge volume of rows

  An index on a high-cardinality column will incur many seeks for very few results. For columns containing unique values, using an index for convenience is fine from a performance perspective, as long as the query volume to the indexed column family is moderate and not under constant load.

- In tables that use a counter column
- On a frequently updated or deleted column

  Cassandra stores tombstones (a marker in a row that indicates that a column was deleted. During compaction, marked columns are deleted in the index (a hidden table) until the tombstone limit reaches 100 K cells. After exceeding this limit, the query that uses the indexed value will fail.

- To look for a row in a large partition

  A query on an indexed column in a large cluster typically requires collating responses from multiple data partitions. The query response slows down as more machines get added to the cluster.

Tip

Important points to take note of

- Don't index on high-cardinality columns
- Don't use index in tables having a counter column
- Don't index on a frequently updated or deleted column
- Don't abuse the index to look for a row in a large partition

Summary

We have learned about the primary and secondary indexes in this lab. Related topics such as compound primary key, composite partition key, and partitioner are also introduced. With the help of the explanation of the internal storage and inner working mechanisms of Cassandra, you should now be able to state the difference between the primary index and the secondary index, as well as use them properly in your data model.

In the next lab, we will start building the first version of the technical analysis application using Cassandra and Python. A quick installation and setup guide on how to connect Python to Cassandra and collect market data will also be provided.