

Lab 3. CQL Data Types



In this lab, we will have an overview of Cassandra Query Language and take a detailed look into the wealthy set of data types supported by Cassandra. We will walk through the data types to study what their internal storage structure looks like.

Note: Directory "lab_03" contains all examples for this lab. Open vscode to see all the files.

Introduction to CQL

Cassandra introduced Cassandra Query Language (CQL) in release 0.8 as a SQL-like alternative to the traditional Thrift RPC API. As of the time of this writing, the latest CQL version is 3.1.7. I do not want to take you through all of its old versions and therefore, I will focus on version 3.1.7 only. It should be noted that CQL Version 3 is not backward compatible with CQL Version 2 and differs from it in many ways.

CQL statements

CQL Version 3 provides a model very similar to SQL. Conceptually, it uses a table to store data in rows of columns. It is composed of three main types of statements:

- **Data definition statements:** These are used to set and change how data is stored in Cassandra
- **Data manipulation statements:** These are used to create, delete, and modify data
- **Query statements:** These are used to look up data

CQL is case insensitive, unless the word is enclosed in double quotation marks. It defines a list of keywords that have a fixed meaning for the language. It distinguishes between reserved and non-reserved keywords. **Reserved** keywords cannot be used as identifiers. They are truly reserved for the language. **Non-reserved** keywords only have a specific meaning in certain contexts but can be used as identifiers. The list of CQL keywords is shown in DataStax's documentation at http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/keywords_r.html.

CQL command-line client – cqlsh

Cassandra bundles an interactive terminal supporting CQL, known as `cqlsh`. It is a Python-based command-line client used to run CQL commands. To start `cqlsh`, navigate to Cassandra's `bin` directory and type the following:

- On Linux, type `./cqlsh`
- On Windows, type `cqlsh.bat` or `python cqlsh`

As shown in the following figure, `cqlsh` shows the cluster name, Cassandra, CQL, and Thrift protocol versions on startup:

```
kan@ubuntu: ~  
kan@ubuntu:~$ cqlsh  
Connected to Test Cluster at localhost:9160.  
[cqlsh 4.1.1 | Cassandra 2.0.9 | CQL spec 3.1.1 | Thrift protocol 19.39.0]  
Use HELP for help.  
cqlsh> 
```

::: cqlsh connected to the Cassandra instance running on the local node :::

We can use `cqlsh` to connect to other nodes by appending the host (either hostname or IP address) and port as command-line parameters.

If we want to create a keyspace called `fenago` using `SimpleStrategy` (which will be explained in, *Enhancing a Version*) as its replication strategy and setting the replication factor as one for a single-node Cassandra cluster, we can type the CQL statement, shown in the following screenshot, in `cqlsh`.

This utility will be used extensively in this course to demonstrate how to use CQL to define the Cassandra data model:

```
kan@ubuntu: ~  
kan@ubuntu:~$ cqlsh  
Connected to Test Cluster at localhost:9160.  
[cqlsh 4.1.1 | Cassandra 2.0.9 | CQL spec 3.1.1 | Thrift protocol 19.39.0]  
Use HELP for help.  
cqlsh> CREATE KEYSPACE packt  
... WITH REPLICATION=  
... {'class': 'SimpleStrategy', 'replication_factor': 1};  
cqlsh> 
```

::: Create keyspace fenago in cqlsh :::

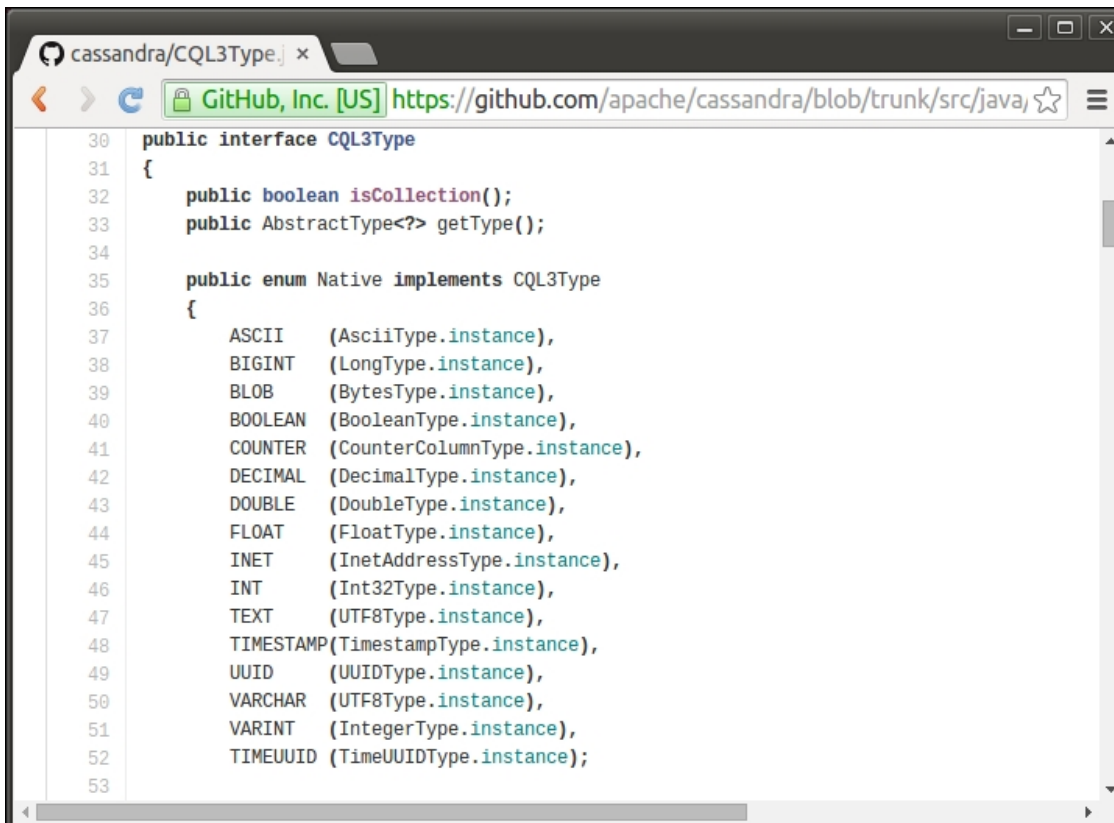
Native data types

CQL Version 3 supports many basic data types for columns. It also supports collection types and all data types available to Cassandra. The following table lists the supported basic data types and their corresponding meanings:

Type	Description
ascii	ASCII character string
bigint	64-bit signed long
blob	Arbitrary bytes (no validation)
Boolean	<code>True</code> or <code>False</code>
counter	Counter column (64-bit signed value)
decimal	Variable-precision decimal
double	64-bit IEEE 754 floating point
float	32-bit IEEE 754 floating point
inet	An IP address that can be either 4 bytes long (IPv4) or 16 bytes long (IPv6) and should be inputted as a string
int	32-bit signed integer
text	UTF8 encoded string
timestamp	A timestamp in which string constants are allowed to input timestamps as dates
timeuuid	Type 1 UUID that is generally used as a "conflict-free" timestamp
uuid	Type 1 or type 4 UUID
varchar	UTF8-encoded string
varint	Arbitrary-precision integer

Cassandra implementation

If we look into the Cassandra's Java source code, the CQL Version 3 native data types are declared in an `enum` called `Native` in the `org.apache.cassandra.cql3.CQL3Type` interface, as shown in the following screenshot:



```
30 public interface CQL3Type
31 {
32     public boolean isCollection();
33     public AbstractType<?> getType();
34
35     public enum Native implements CQL3Type
36     {
37         ASCII      (AsciiType.instance),
38         BIGINT      (LongType.instance),
39         BLOB        (BytesType.instance),
40         BOOLEAN     (BooleanType.instance),
41         COUNTER     (CounterColumnType.instance),
42         DECIMAL     (DecimalType.instance),
43         DOUBLE      (DoubleType.instance),
44         FLOAT       (FloatType.instance),
45         INET        (InetAddressType.instance),
46         INT         (Int32Type.instance),
47         TEXT        (UTF8Type.instance),
48         TIMESTAMP  (TimestampType.instance),
49         UUID        (UUIDType.instance),
50         VARCHAR     (UTF8Type.instance),
51         VARINT      (IntegerType.instance),
52         TIMEUUID    (TimeUUIDType.instance);
53 }
```

... Cassandra source code declaring CQL Version 3 native data types ...

It is interesting to know that `TEXT` and `VARCHAR` are indeed both `UTF8Type`. The Java classes of `AsciiType`, `LongType`, `BytesType`, `DecimalType`, and so on are declared in the `org.apache.cassandra.db.marshall` package.

Note

Cassandra source code is available on GitHub at <https://github.com/apache/cassandra>.

Knowing the Java implementation of the native data types allows us to have a deeper understanding of how Cassandra handles them. For example, Cassandra uses the `org.apache.cassandra.serializers.InetAddressSerializer` class and `java.net.InetAddress` class to handle the serialization/deserialization of the `INET` data type.

A not-so-long example

These native data types are used in CQL statements to specify the type of data to be stored in a column of a table. Now let us create an experimental table with columns of each native data type (except counter type since it requires a separate table), and then insert some data into it. We need to specify the keyspace, `fenago` in this example, before creating the table called `table01`, as shown in the following screenshot:

```
kan@ubuntu: ~  
kan@ubuntu:~$ cqlsh  
Connected to Test Cluster at localhost:9160.  
[cqlsh 4.1.1 | Cassandra 2.0.9 | CQL spec 3.1.1 | Thrift protocol 19.39.0]  
Use HELP for help.  
cqlsh> USE packt;  
cqlsh:packt>  
cqlsh:packt> CREATE TABLE table01 (  
    ... rowkey ascii,  
    ... asciifield ascii,  
    ... bigintfield bigint,  
    ... blobfield blob,  
    ... booleanfield boolean,  
    ... decimalfield decimal,  
    ... doublefield double,  
    ... floatfield float,  
    ... inetfield inet,  
    ... intfield int,  
    ... textfield text,  
    ... timestampfield timestamp,  
    ... timeuuidfield timeuuid,  
    ... uuidfield uuid,  
    ... varcharfield varchar,  
    ... varintfield varint,  
    ... PRIMARY KEY (rowkey)  
    ... );  
cqlsh:packt>
```

::: Create table01 to illustrate each native data type :::

We create the table using the default values, but, there are other options to configure the new table for optimizations, including compaction, compression, failure handling, and so on. The `PRIMARY KEY` clause, which is on only one column, could also be specified along with an attribute, that is, `rowkey ascii PRIMARY KEY`. Then insert a sample record into `table01`. We make it with an `INSERT` statement, as shown in the following screenshot:

```
kan@ubuntu: ~  
cqlsh:packt> INSERT INTO table01  
    ... (rowkey, asciifield, bigintfield, blobfield, booleanfield,  
    ... decimalfield, doublefield, floatfield, inetfield, intfield,  
    ... textfield, timestampfield, timeuuidfield, uuidfield,  
    ... varcharfield, varintfield)  
    ... VALUES  
    ... ('1', 'ABC', 1000000000, textAsBlob('ABC'), True,  
    ... 1.0, 1.123456789, 1.123456, '192.168.0.1', 1,  
    ... 'ABC', '2014-05-01 01:02:03', now(), uuid(),  
    ... 'ABC', 1);  
cqlsh:packt>  
cqlsh:packt>
```

::: Insert a sample record into table01 :::

We now have data inside `table01`. We use `cqlsh` to query the table. For the sake of comparison, we also use another Cassandra command-line tool called Cassandra CLI to have a low-level view of the row. Let us open Cassandra CLI on a terminal.

Note

Cassandra CLI utility

Cassandra CLI is used to set storage configuration attributes on a per-keyspace or per-table basis. To start it up, you navigate to Cassandra bin directory and type the following:

- On Linux, `./cassandra-cli`
- On Windows, `cassandra.bat` :::

Note that it was announced to be deprecated in Cassandra 3.0 and `cqlsh` should be used instead.

The results of the `SELECT` statement in `cqlsh` and the `list` command in Cassandra CLI are shown in the following screenshot. We will then walk through each column one by one:

```
kan@ubuntu: ~  
cqlsh:packt> SELECT * FROM table01;  
  
 rowkey | asciifield | bigintfield | blobfield | booleanfield | decimalfield | d  
oublefield | floatfield | inetfield | intfield | textfield | timestampfield  
         | timeuuidfield |                | uuidfield  
         | varcharfield | varintfield  
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----  
      1 |          ABC | 1000000000 | 0x414243 |           True |            1.0 |  
    1.1235 |        1.1235 | 192.168.0.1 |          1 |          ABC | 2014-05-01 01:02:  
03+0800 | 84763d40-1a1e-11e4-8449-2d63f07021c6 | 60903075-d9e1-404f-86dc-9670f42  
ea10b |          ABC |              1  
  
(1 rows)  
  
cqlsh:packt>  
  
kan@ubuntu: ~  
[default@packt] list table01;  
Using default limit of 100  
Using default cell limit of 100  
-----  
RowKey: 1  
=> (name=, value=, timestamp=1406967910163000)  
=> (name=asciifield, value=414243, timestamp=1406967910163000)  
=> (name=bigintfield, value=000000003b9aca00, timestamp=1406967910163000)  
=> (name=blobfield, value=414243, timestamp=1406967910163000)  
=> (name=booleanfield, value=01, timestamp=1406967910163000)  
=> (name=decimalfield, value=000000010a, timestamp=1406967910163000)  
=> (name=doublefield, value=3ff1f9add3739636, timestamp=1406967910163000)  
=> (name=floatfield, value=3f8fcd68, timestamp=1406967910163000)  
=> (name=inetfield, value=c0a80001, timestamp=1406967910163000)  
=> (name=intfield, value=00000001, timestamp=1406967910163000)  
=> (name=textfield, value=414243, timestamp=1406967910163000)  
=> (name=timestampfield, value=00000145b395fef8, timestamp=1406967910163000)  
=> (name=timeuuidfield, value=84763d401a1e11e484492d63f07021c6, timestamp=140696  
7910163000)  
=> (name=uuidfield, value=60903075d9e1404f86dc9670f42ea10b, timestamp=1406967910  
163000)  
=> (name=varcharfield, value=414243, timestamp=1406967910163000)  
=> (name=varintfield, value=01, timestamp=1406967910163000)  
  
1 Row Returned.  
Elapsed time:32 msec(s).  
[default@packt]
```

::: Comparison of the sample row in cqlsh and Cassandra CLI :::

ASCII

Internally, a data value 'ABC' is stored as the byte values in hexadecimal representation of each individual character, 'A' , 'B' , and 'C' as 0x41 , 0x42 , and 0x43 respectively.

Bigint

This is simple; the hexadecimal representation of the number 1000000000 is 0x000000003b9aca00 of 64-bit length stored internally.

BLOB

A BLOB data type is used to store a large binary object. In our previous example, we inserted a text 'ABC' as a BLOB into the blobfield. The internal representation is 414243, which is just a stream of bytes in hexadecimal representation.

Obviously, a BLOB field can accept all kinds of data, and because of this flexibility it cannot have validation on its data value. For example, a data value `2` may be interpreted as either an integer `2` or a text `'2'`. Without knowing the interpretation we want, a BLOB field can impose a check on the data value.

Another interesting point of a BLOB field is that, as shown in the `SELECT` statement in the previous screenshot in `cqlsh`, the data value of `blobfield` returned is `0x414243` for 'ABC' text. We know from the previous section that `0x41`, `0x42`, `0x43` are the byte values of 'A', 'B', and 'C', respectively. However, for a BLOB field, `cqlsh` prefixes its data value with '`0x`' to make it a so-called BLOB constant. A BLOB constant is a sequence of bytes in their hexadecimal values prefixed by `0[xX](hex)+` where `hex` is a hexadecimal character, such as `[0-9a-fA-F]`.

CQL also provides a number of BLOB conversion functions to convert native data types into a BLOB and vice versa. For every <native-type> (except BLOB for an obvious reason) supported by CQL, the <native-type>AsBlob function takes an argument of type <native-type> and returns it as a BLOB. Contrarily, the blobAs<Native-type> function reverses the conversion from a BLOB back to a <native-type>. As demonstrated in the INSERT statement, we have used textAsBlob() to convert a text data type into a BLOB.

Note

BLOB constant

BLOB constants were introduced in CQL version 3.0.2 to allow users to input BLOB values. In older versions of CQL, inputting BLOB as string was supported for

convenience. It is now deprecated and will be removed in a future version. It is still supported only to allow smoother transition to a `BLOB` constant. Updating the client code to switch to `BLOB` constants should be done as soon as possible.

Boolean

A `boolean` data type is also very intuitive. It is merely a single byte of either `0x00`, which means `False`, or `0x01`, which means `True`, in the internal storage.

Decimal

A `decimal` data type can store a variable-precision decimal, basically a `BigDecimal` data type in Java.

Double

The `double` data type is a double-precision 64-bit IEEE 754 floating point in its internal storage.

Float

The `float` data type is a single-precision 32-bit IEEE 754 floating point in its internal storage.

Note

BigDecimal, double, or float?

The difference between `double` and `float` is obviously the length of precision in the floating point value. Both `double` and `float` use binary representation of decimal numbers with a radix which is in many cases an approximation, not an absolute value. `double` is a 64-bit value while `float` is an even shorter 32-bit value. Therefore, we can say that `double` is more precise than `float`. However, in both cases, there is still a possibility of loss of precision which can be very noticeable when working with either very big numbers or very small numbers.

On the contrary, `BigDecimal` is devised to overcome this loss of precision discrepancy. It is an exact way of representing numbers. Its disadvantage is slower runtime performance.

Whenever you are dealing with money or precision is a must, `BigDecimal` is the best choice (or `decimal` in CQL native data types), otherwise `double` or `float` should be good enough.

Inet

The `inet` data type is designed for storing IP address values in **IP Version 4 (IPv4)** and **IP Version 6 (IPv6)** format. The IP address, `192.168.0.1`, in the example record is stored as four bytes internally; `192` is stored as `0xc0`, `168` as `0xa8`, `0` as `0x00`, and `1` as `0x01`, respectively. It should be noted that regardless of the IP address being stored is IPv4 or IPv6, the port number is *not* stored. We need another column to store it if required.

We can also store an IPv6 address value. The following `UPDATE` statement changes the `inetfield` to an IPv6 address `2001:0db8:85a3:0042:1000:8a2e:0370:7334`, as shown in the following screenshot:


```

kan@ubuntu: ~
1 | ABC | 1000000000 | 0x414243 | True | 1.0 |
1.1235 | 1.1235 | 192.168.0.1 | 1 | ABC | 2014-05-01 01:02:
03+0800 | 84763d40-1a1e-11e4-8449-2d63f07021c6 | 60903075-d9e1-404f-86dc-9670f42
ea10b | ABC | 1

(1 rows)

cqlsh:packt>
cqlsh:packt>
cqlsh:packt> UPDATE table01 SET inetfield = '2001:0db8:85a3:0042:1000:8a2e:0370:
7334' WHERE rowkey = '1';
cqlsh:packt> SELECT inetfield FROM table01;

inetfield
-----
2001:db8:85a3:42:1000:8a2e:370:7334

(1 rows)

cqlsh:packt> 

```

```

kan@ubuntu: ~
Using default limit of 100
Using default cell limit of 100
-----
RowKey: 1
=> (name=, value=, timestamp=1406967910163000)
=> (name=asciifield, value=414243, timestamp=1406967910163000)
=> (name=bigintfield, value=000000003b9aca00, timestamp=1406967910163000)
=> (name=blobfield, value=414243, timestamp=1406967910163000)
=> (name=booleanfield, value=01, timestamp=1406967910163000)
=> (name=decimalfield, value=000000010a, timestamp=1406967910163000)
=> (name=doublefield, value=3ff1f9add3739636, timestamp=1406967910163000)
=> (name=floatfield, value=3f8fcd68, timestamp=1406967910163000)
=> (name=inetfield, value=20010db885a3004210008a2e03707334, timestamp=1406979926
467000)
=> (name=intfield, value=00000001, timestamp=1406967910163000)
=> (name=textfield, value=414243, timestamp=1406967910163000)
=> (name=timestampfield, value=00000145b395fef8, timestamp=1406967910163000)
=> (name=timestampfield, value=84763d401a1e11e484492d63f07021c6, timestamp=140696
7910163000)
=> (name=uuidfield, value=60903075d9e1404f86dc9670f42ea10b, timestamp=1406967910
163000)
=> (name=varcharfield, value=414243, timestamp=1406967910163000)
=> (name=varintfield, value=01, timestamp=1406967910163000)

1 Row Returned.
Elapsed time:34 msec(s).
[default@packt]

```

::: Comparison of the sample row in cqlsh and Cassandra CLI in inetfield :::

Note

Internet Protocol Version 6

Internet Protocol Version 6 (IPv6) is the latest version of the **Internet Protocol (IP)**. It was developed by the IETF to deal with the long-anticipated problem of IPv4 address exhaustion.

IPv6 uses a 128-bit address whereas IPv4 uses 32-bit address. The two protocols are not designed to be interoperable, making the transition to IPv6 complicated.

IPv6 addresses are usually represented as eight groups of four hexadecimal digits separated by colons, such as 2001:0db8:85a3:0042:1000:8a2e:0370:7334.

In `cg1sh`, the leading zeros of each group of four hexadecimal digits are removed. In Cassandra's internal storage, the IPv6 address value consumes 16 bytes.

Int

The `int` data type is a primitive 32-bit signed integer.

Text

The `text` data type is a UTF-8 encoded string accepting Unicode characters. As shown previously, the byte values of `\ " ABC \"`, `0x41`, `0x42`, and `0x43`, are stored internally. We can test the `text` field with non-ASCII characters by updating the `textField` as shown in the following screenshot:

The `text` data type is a combination of non-ASCII and ASCII characters. The four non-ASCII characters are represented as their 3-byte UTF-8 values, `0xe8b584`, `0xe6ba90`, `0xe68f90`, and `0xe4be9b`.

However, the ASCII characters are still stored as byte values, as shown in the screenshot:

```

kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> UPDATE table01 SET textfield='资源提供ABC'
... WHERE rowkey='1';
cqlsh:packt>
cqlsh:packt> SELECT textfield FROM table01;

textfield
-----
资源提供ABC

(1 rows)

cqlsh:packt>
kan@ubuntu: ~
=> (name=bigintfield, value=000000003b9aca00, timestamp=1406967910163000)
=> (name=blobfield, value=414243, timestamp=1406967910163000)
=> (name=booleanfield, value=01, timestamp=1406967910163000)
=> (name=decimalfield, value=000000010a, timestamp=1406967910163000)
=> (name=doublefield, value=3ff1f9add3739636, timestamp=1406967910163000)
=> (name=floatfield, value=3f8fcd68, timestamp=1406967910163000)
=> (name=inetfield, value=20010db885a3004210008a2e03707334, timestamp=1406979926467000)
=> (name=intfield, value=00000001, timestamp=1406967910163000)
=> (name=textfield, value=e8b584e6ba90e68f90e4be9b414243, timestamp=1406999832979000)
=> (name=timestampfield, value=00000145b395fef8, timestamp=1406967910163000)
=> (name=timeuuidfield, value=84763d401a1e11e484492d63f07021c6, timestamp=1406967910163000)
=> (name=uuidfield, value=60903075d9e1404f86dc9670f42ea10b, timestamp=1406967910163000)
=> (name=varcharfield, value=414243, timestamp=1406967910163000)
=> (name=varintfield, value=01, timestamp=1406967910163000)

1 Row Returned.
Elapsed time: 128 msec(s).
[default@packt]

```

... Experiment of the textfield data type ...

Timestamp

The value of the `timestampfield` is encoded as a 64-bit signed integer representing a number of milliseconds since the standard base time known as the *epoch*: January 1, 1970, at 00:00:00 GMT. A `timestamp` data type can be entered as an integer for CQL input, or as a string literal in ISO 8601 formats. As shown in the following screenshot, the internal value of May 1, 2014, 16:02:03, in the +08:00 timezone is `0x00000145b6cdf878` or 1,398,931,323,000 milliseconds since the epoch:

```

packt@ubuntu: ~
Using default limit of 100
Using default cell limit of 100
-----
RowKey: 1
=> (name=, value=, timestamp=1416811483812312)
=> (name=asciifield, value=414243, timestamp=1416811483812312)
=> (name=bigintfield, value=000000003b9aca00, timestamp=1416811483812312)
=> (name=blobfield, value=414243, timestamp=1416811483812312)
=> (name=booleanfield, value=01, timestamp=1416811483812312)
=> (name=decimalfield, value=000000010a, timestamp=1416811483812312)
=> (name=doublefield, value=3ff1f9add3739636, timestamp=1416811483812312)
=> (name=floatfield, value=3f8fcd68, timestamp=1416811483812312)
=> (name=inetfield, value=c0a80001, timestamp=1416811483812312)
=> (name=intfield, value=00000001, timestamp=1416811483812312)
=> (name=textfield, value=414243, timestamp=1416811483812312)
=> (name=timestampfield, value=00000145b395fef8, timestamp=1416811483812312)
=> (name=timeuuidfield, value=5f96213073a511e4a62fa92bc9056ee6, timestamp=1416811483812312)
=> (name=uuidfield, value=62a866ba149b4eac8c9ef5400f52dfec, timestamp=1416811483812312)
=> (name=varcharfield, value=414243, timestamp=1416811483812312)
=> (name=varintfield, value=01, timestamp=1416811483812312)

1 Row Returned.
Elapsed time: 90 msec(s).
[default@packt]

```

... Experiment of the timestamp data type ...

A `timestamp` data type contains a date portion and a time portion in which the time of the day can be omitted if only the value of the date is wanted. Cassandra will use 00:00:00 as the default for the omitted time of day.

Note

ISO 8601

ISO 8601 is the international standard for representation of dates and times. Its full reference number is ISO 8601:1988 (E), and its title is "Data elements and interchange formats – Information interchange – Representation of dates and times."

ISO 8601 describes a large number of date/time formats depending on the desired level of granularity. The formats are as follows. Note that the "T" appears literally in the string to indicate the beginning of the time element.

- Year: YYYY (e.g. 1997)
- Year and month: YYYY-MM (e.g. 1997-07)
- Date: YYYY-MM-DD (e.g. 1997-07-16)
- Date plus hours and minutes: YYYY-MM-DDThh:mmTZD (e.g. 1997-07-16T19:20+01:00)
- Date plus hours, minutes and seconds: YYYY-MM-DDThh:mm:ssTZD (e.g. 1997-07-16T19:20:30+01:00)
- Date plus hours, minutes, seconds and a decimal fraction of a second: YYYY-MM-DDThh:mm:ss.sTZD (e.g. 1997-07-16T19:20:30.45+01:00)

Where:

- YYYY = four-digit year
- MM = two-digit month (01=January, etc.)
- DD = two-digit day of month (01 through 31)
- hh = two digits of hour (00 through 23) (am/pm NOT allowed)
- mm = two digits of minute (00 through 59)
- ss = two digits of second (00 through 59)
- s = one or more digits representing a decimal fraction of a second
- TZD = time zone designator (Z or +hh:mm or -hh:mm)

Times are expressed either in **Coordinated Universal Time (UTC)** with a special UTC designator "Z" or in local time together with a time zone offset in hours and minutes. A time zone offset of "+/-hh:mm" indicates the use of a local time zone which is "hh" hours and "mm" minutes ahead/behind of UTC.

If no time zone is specified, the time zone of the Cassandra coordinator node handling the write request is used. Therefore the best practice is to specify the time zone with the timestamp rather than relying on the time zone configured on the Cassandra nodes to avoid any ambiguities.

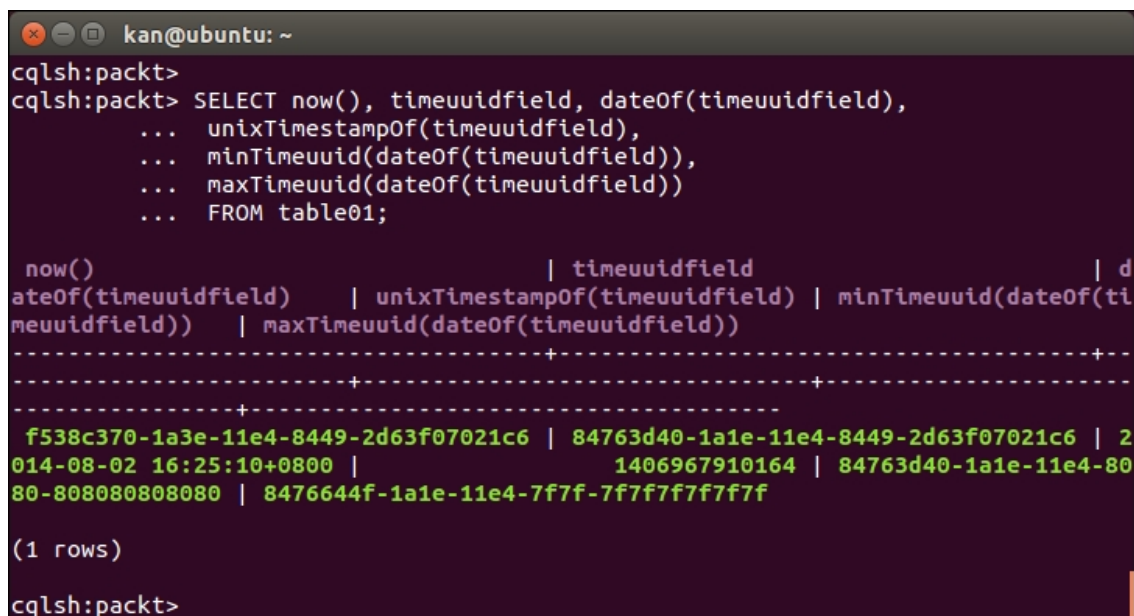
Timeuuid

A value of the `timeuuid` data type is a Type 1 UUID which includes the time of its generation and is sorted by timestamp. It is therefore ideal for use in applications requiring conflict-free timestamps. A valid `timeuuid` uses the time in 100 intervals since 00:00:00.00 UTC (60 bits), a clock sequence number for prevention of duplicates (14 bits), and the IEEE 801 MAC address (48 bits) to generate a unique identifier, for example, `74754ac0-e13f-11e3-a8a3-a92bc9056ee6`.

CQL v3 offers a number of functions to make the manipulation of `timeuuid` handy:

- **dateOf()**: This is used in a `SELECT` statement to extract the timestamp portion of a `timeuuid` column
- **now()**: This is used to generate a new unique `timeuuid`
- **minTimeuuid()** and **maxTimeuuid()**: These are used to return a result similar to a UUID given a conditional time component as its argument
- **unixTimestampOf()**: This is used in a `SELECT` statement to extract the timestamp portion as a raw 64-bit integer timestamp of a `timeuuid` column :::

The following figure uses `timeuuidfield` of `table01` to demonstrate the usage of these `timeuuid` functions:



```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> SELECT now(), timeuuidfield, dateOf(timeuuidfield),
...     unixTimestampOf(timeuuidfield),
...     minTimeuuid(dateOf(timeuuidfield)),
...     maxTimeuuid(dateOf(timeuuidfield))
...     FROM table01;

now() | timeuuidfield | d
ateOf(timeuuidfield) | unixTimestampOf(timeuuidfield) | minTimeuuid(dateOf(ti
meuuidfield)) | maxTimeuuid(dateOf(timeuuidfield))
-----+-----+-----+-----+-----+
f538c370-1a3e-11e4-8449-2d63f07021c6 | 84763d40-1a1e-11e4-8449-2d63f07021c6 | 2
014-08-02 16:25:10+0800 | 1406967910164 | 84763d40-1a1e-11e4-80
80-808080808080 | 8476644f-1a1e-11e4-7f7f-7f7f7f7f7f7f

(1 rows)
cqlsh:packt>
```

::: Demonstration of timeuuid functions :::

Note

Timestamp or Timeuuid?

Timestamp is suitable for storing date and time values. TimeUUID, however, is more suitable in those cases where a conflict free, unique timestamp is needed.

UUID

The `UUID` data type is usually used to avoid collisions in values. It is a 16-byte value that accepts a type 1 or type 4 UUID. CQL v3.1.6 or later versions provide a function called `uuid()` to easily generate random type 4 UUID values.

Note

Type 1 or type 4 UUID?

Type 1 uses the MAC address of the computer that is generating the `UUID` data type and the number of 100-nanosecond intervals since the adoption of the Gregorian calendar, to generate UUIDs. Its uniqueness across computers is guaranteed if MAC addresses are not duplicated; however, given the speed of modern processors, successive invocations on the same machine of a naive implementation of a type 1 generator might produce the same `UUID`, negating the property of uniqueness.

Type 4 uses random or pseudorandom numbers. Therefore, it is the recommended type of `UUID` to be used.

Varchar

Basically `varchar` is identical to `text` as evident by the same `UTF8Type` in the source code.

Varint

A `varint` data is used to store integers of arbitrary precision.

Counter

A `counter` data type is a special kind of column whose user-visible value is a 64-bit signed integer (though this is more complex internally) used to store a number that incrementally counts the occurrences of a particular event. When a new value is written to a given counter column, it is added to the previous value of the counter.

A counter is ideal for counting things quickly in a distributed environment which makes it invaluable for real time analytical tasks. The `counter` data type was introduced in Cassandra Version 0.8. Counter column tables must use `counter` data type. Counters can be stored in dedicated tables only, and you cannot create an index on a counter column.

Tip

Counter type don'ts

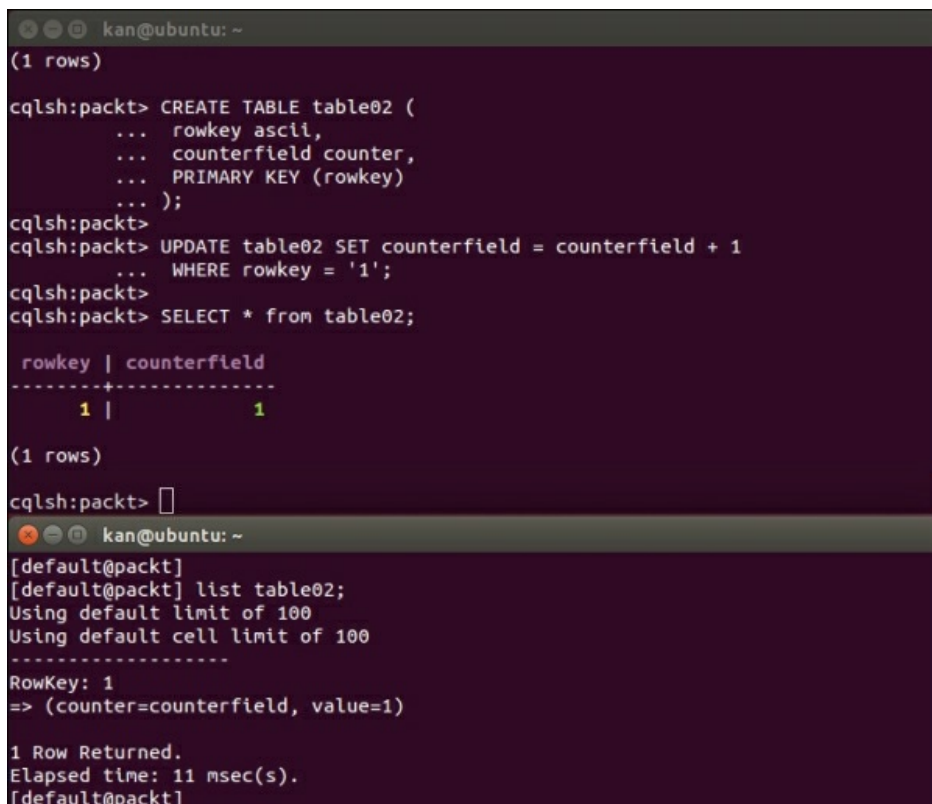
- Don't assign the `counter` data type to a column that serves as the primary key
- Don't use the `counter` data type in a table that contains anything other than `counter` data types and primary keys
- Don't use the `counter` data type to generate sequential numbers for surrogate keys; use the `timeuuid` data type instead :::

We use a `CREATE TABLE` statement to create a counter table. However, `INSERT` statements are not allowed on counter tables and so we must use an `UPDATE` statement to update the counter column as shown in the following screenshot.

Cassandra uses `counter` instead of `name` to indicate that the column is of a counter data type. The counter value is stored in the value of the column.

This is a very good article that explains the internals of how a counter works in a distributed environment <http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counters>.

The following screenshot shows that counter value is stored in the value of the column:



```
kan@ubuntu: ~
(1 rows)

cqlsh:packt> CREATE TABLE table02 (
...   rowkey ascii,
...   counterfield counter,
...   PRIMARY KEY (rowkey)
... );
cqlsh:packt>
cqlsh:packt> UPDATE table02 SET counterfield = counterfield + 1
... WHERE rowkey = '1';
cqlsh:packt>
cqlsh:packt> SELECT * from table02;

 rowkey | counterfield
-----+-----
      1 |           1

(1 rows)

cqlsh:packt>

kan@ubuntu: ~
[default@packt]
[default@packt] list table02;
Using default limit of 100
Using default cell limit of 100
-----
RowKey: 1
=> (counter=counterfield, value=1)

1 Row Returned.
Elapsed time: 11 msec(s).
[default@packt]
```

Collections

Cassandra also supports collections in its data model to store a small amount of data. Collections are a complex type that can provide tremendous flexibility. Three collections are supported: Set, List, and Map. The type of data stored in each of these collections requires to be defined, for example, a set of timestamp is defined as `set<timestamp>`, a list of text is defined as `list<text>`, a map containing a text key and a text value is defined as `map<text, text>`, and so on. Also, only native data types can be used in collections.

Cassandra reads a collection in its entirety and the collection is not paged internally. The maximum number of items of a collection is 64K and the maximum size of an item is 64K.

To better demonstrate the CQL support on these collections, let us create a table in the `fenago` keyspace with columns of each collection and insert some data into it, as shown in the following screenshot:

```
kan@ubuntu: ~  
cqlsh:packt> CREATE TABLE table03 (  
... rowkey ascii,  
... setfield set<text>,  
... listfield list<text>,  
... mapfield map<text, text>,  
... PRIMARY KEY (rowkey)  
... );  
cqlsh:packt>  
cqlsh:packt> INSERT INTO table03  
... (rowkey, setfield, listfield, mapfield)  
... VALUES  
... ('1', {'Lemon', 'Orange', 'Apple'},  
... ['Lemon', 'Orange', 'Apple'],  
... {'fruit1': 'Apple', 'fruit3': 'Orange', 'fruit2': 'Lemon'});  
cqlsh:packt>  
cqlsh:packt> SELECT * from table03;  
  
rowkey | listfield | setfield | mapfield  
-----+-----+-----+-----  
1 | ['Lemon', 'Orange', 'Apple'] | {'fruit1': 'Apple', 'fruit2': 'Lemon', 'fruit3': 'Orange'} | {'Apple', 'Lemon', 'Orange'}  
  
(1 rows)  
cqlsh:packt>   
  
kan@ubuntu: ~  
[default@packt] list table03;  
Using default limit of 100  
Using default cell limit of 100  
-----  
RowKey: 1  
=> (name=, value=, timestamp=1406989055148000)  
=> (name=listfield:bfdabec01a4f11e484492d63f07021c6, value=4c656d6f6e, timestamp=1406989055148000)  
=> (name=listfield:bfdabec11a4f11e484492d63f07021c6, value=4f72616e6765, timestamp=1406989055148000)  
=> (name=listfield:bfdabec21a4f11e484492d63f07021c6, value=4170706c65, timestamp=1406989055148000)  
=> (name=mapfield:667275697431, value=4170706c65, timestamp=1406989055148000)  
=> (name=mapfield:667275697432, value=4c656d6f6e, timestamp=1406989055148000)  
=> (name=mapfield:667275697433, value=4f72616e6765, timestamp=1406989055148000)  
=> (name=setfield:4170706c65, value=, timestamp=1406989055148000)  
=> (name=setfield:4c656d6f6e, value=, timestamp=1406989055148000)  
=> (name=setfield:4f72616e6765, value=, timestamp=1406989055148000)  
  
1 Row Returned.  
Elapsed time: 87 msec(s).  
[default@packt]   

```

Note

How to update or delete a collection?

CQL also supports updation and deletion of elements in a collection. You can refer to the relevant information in DataStax's documentation at http://www.datastax.com/documentation/cql/3.1/cql/cql_using/use_collections_c.html.

As in the case of native data types, let us walk through each collection below.

Set

CQL uses sets to keep a collection of unique elements. The benefit of a set is that Cassandra automatically keeps track of the uniqueness of the elements and we, as application developers, do not need to bother on it.

CQL uses curly braces (`{}`) to represent a set of values separated by commas. An empty set is simply `{}`. In the previous example, although we inserted the set as

`{ 'Lemon', 'Orange', 'Apple' }` , the input order was not preserved. Why?

The reason is in the mechanism of how Cassandra stores the set. Internally, Cassandra stores each element of the set as a single column whose column name is the original column name suffixed by a colon and the element value. As shown previously, the ASCII values of 'Apple' , 'Lemon' , and 'Orange' are `0x4170706c65` , `0x4c656d666e` , and `0x4f72616e6765` , respectively. So they are stored in three columns with column names, `setfield:4170706c65` , `setfield:4c656d666e` , and `setfield:4f72616e6765` . By the built-in order column-name-nature of Cassandra, the elements of a set are sorted automatically.

List

A list is ordered by the natural order of the type selected. Hence it is suitable when uniqueness is not required and maintaining order is required.

CQL uses square brackets (`[]`) to represent a list of values separated by commas. An empty list is `[]` . In contrast to a set, the input order of a list is preserved by Cassandra. Cassandra also stores each element of the list as a column. But this time, the columns have the same name composed of the original column name (`listfield` in our example), a colon, and a UUID generated at the time of update. The element value of the list is stored in the value of the column.

Map

A map in Cassandra is a dictionary-like data structure with keys and values. It is useful when you want to store table-like data within a single Cassandra row.

CQL also uses curly braces (`{}`) to represent a map of keys and values separated by commas. Each key-value pair is separated by a colon. An empty map is simply represented as `{}` . Conceivably, each key/value pair is stored in a column whose column name is composed of the original map column name followed by a colon and the key of that pair. The value of the pair is stored in the value of the column. Similar to a set, the map sorts its items automatically. As a result, a map can be imagined as a hybrid of a set and a list.

User-defined type and tuple type

Cassandra 2.1 introduces support for **User-Defined Types (UDT)** and tuple types.

User-defined types are declared at the keyspace level. A user-defined type simplifies handling a group of related properties. We can define a group of related properties as a type and access them separately or as a single entity. We can map our UDTs to application entities. Another new type for CQL introduced by Cassandra 2.1 is the tuple type. A tuple is a fixed-length set of typed positional fields without labels.

We can use user-defined and tuple types in tables. However, to support future capabilities, a column definition of a user-defined or tuple type requires the `frozen` keyword. Cassandra serializes a frozen value having multiple components into a single value. This means we cannot update parts of a UDT value. The entire value must be overwritten. Cassandra treats the value of a frozen UDT like a `BLOB` .

We create a UDT called `contact` in the `fenago` keyspace and use it to define `contactfield` in `table04` . Moreover, we have another column, `tuplefield` , to store a tuple in a row. Pay attention to the syntax of the `INSERT` statement for UDT and tuple. For UDT, we may use a dotted notation to retrieve a component of the UDT column, such as `contactfield.facebook` in our following example. As shown in `cassandra-cli` , `contactfield` is stored as a single value, `0000000162000000163000000076440642e636f6d` .

The value concatenates each UDT component in sequence with the format, a length of 4 bytes indicating the length of the component value and the component value itself. So, for `contactfield.facebook` , `0x00000001` is the length and `0x62` is the byte value of 'a' . Cassandra applies the same treatment to a tuple:


```
packt@ubuntu: ~
cqlsh:packt>
cqlsh:packt>
cqlsh:packt> CREATE TYPE contact (
...     facebook text,
...     twitter text,
...     email text
... );
cqlsh:packt>
cqlsh:packt> CREATE TABLE table04 (
...     rowkey ascii PRIMARY KEY,
...     contactfield frozen<contact>,
...     tuplefield frozen<tuple<int, text>>
... );
cqlsh:packt>
cqlsh:packt> INSERT INTO table04 (rowkey, contactfield, tuplefield)
...     VALUES ('a', {facebook:'b',twitter:'c',email:'d@d.com'},
...     (1,'e'));
cqlsh:packt>
cqlsh:packt> SELECT contactfield, contactfield.facebook, tuplefield FROM table04
;

contactfield                                     | contactfield.facebook | tuple
field
-----+-----+-----
{facebook: 'b', twitter: 'c', email: 'd@d.com'} | b | (1,
'e')

(1 rows)
cqlsh:packt>

packt@ubuntu: ~
[default@packt]
[default@packt] list table04;
Using default limit of 100
Using default cell limit of 100
-----
RowKey: a
=> (name=, value=, timestamp=1414889697969626)
=> (name=contactfield, value=00000001620000000163000000076440642e636f6d, timesta
mp=1414889697969626)
=> (name=tuplefield, value=00000004000000010000000165, timestamp=141488969796962
6)

1 Row Returned.
Elapsed time: 86 msec(s).
[default@packt]
```

Experiment of user-defined and tuple types

Further information can be found at DataStax's documentation, available at the following links:

http://www.datastax.com/documentation/cql/3.1/cql/cql_using/cqlUseUDT.html

- <http://www.datastax.com/documentation/developer/python-driver/2.1/python-driver/reference/tupleTypes.html>

Summary

This lab is the second part of Cassandra data modeling. We have learned the basics of Cassandra Query Language (CQL), which offers a SQL-like language to implement a Cassandra data model and operate the data inside. Then a very detailed walkthrough, with ample examples of native data types, more advanced collections, and new user-defined and tuple types, was provided to help you know how to select appropriate data types for your data models. The internal storage of each data type was also explained to let you know how Cassandra implements its data types.

In the next lab, we will learn another important element of a Cassandra query indexes.