Data types

Apache Cassandra comes with many common types that can help you with efficient data storage and representation. As Cassandra is written in Java, the Cassandra data types correlate directly to Java data types:

CQL type	Java class	Description
`ascii`	`String`	US-ASCII (United States – American Standard Codes for Information Interchange) character string.
`bigint`	`Long`	64-bit signed long.
`blob`	`java.nio.ByteBuffer`	Supports storage of BLOBs (binary large objects).
`date`	`java.time.LocalDate`	32-bit integer denoting the number of days elapsed since January 1, 1970. Dates can also be specified in CQL queries as strings (such as July 22, 2018).
`decimal`	`java.math.BigDecimal`	Arbitrary precision floating-point numeric.
`double`	`Double`	64-bit floating-point numeric.
`float`	`Float`	32-bit floating-point numeric.
`inet`	`java.net.InetAddress`	String type for working with IP addresses. Supports both IPv4 and IPv4 addresses.
`int`	`Integer`	32-bit signed Integer.
`list`	`java.util.List`	A collection of ordered items of a specified type.
`map`	`java.util.Map`	A key/value collection that stores values by a unique key. Type can be specified for both key and value.
`set`	`java.util.Set`	A collection of unique items of a specified type.

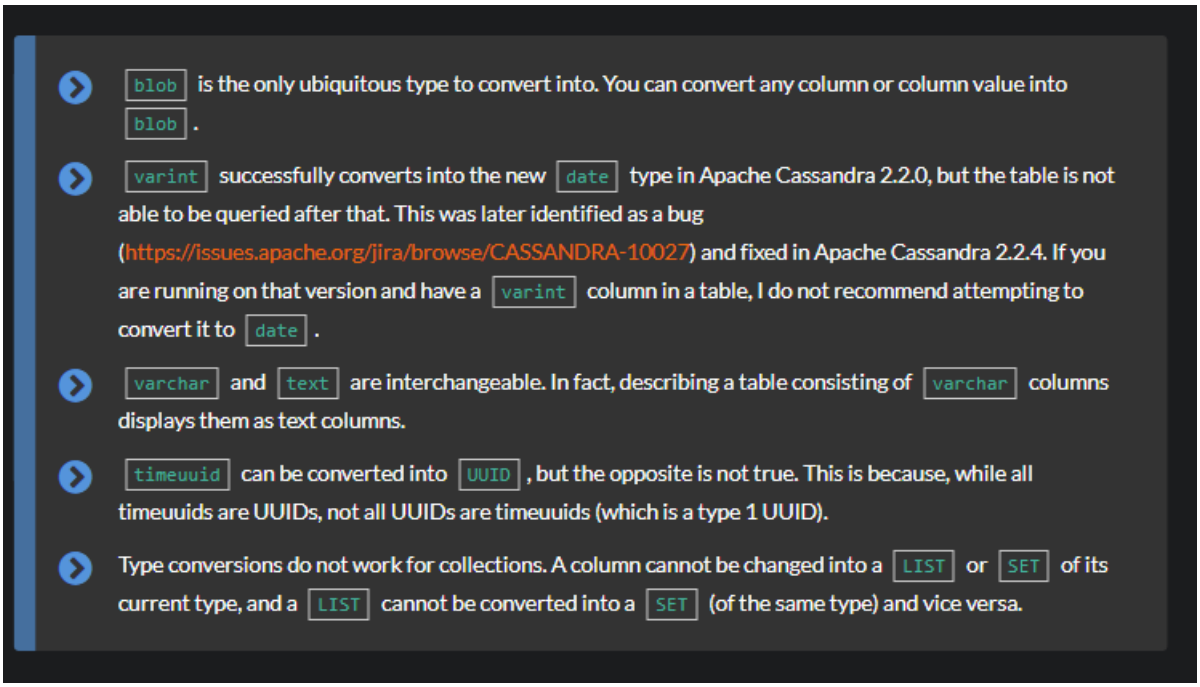
`smallint`	`Short`	16-bit integer.
`text`	`String`	UTF-8 character string.
`time`	`Long`	The current time in milliseconds elapsed since midnight of the current day.
`timestamp`	`java.util.Date`	Date/time with milliseconds, can also be specified in CQL queries as strings (such as `2018-07-22 22:51:13.442`).
`timeuuid`	`java.util.UUID`	Type 1 UUID (128-bit), used for storing times.
`tinyint`	`Byte`	8-bit integer.
`tuple`	`com.datastax.driver.core.TupleValue`	A tuple type, consisting of a collection of 2 to 3 values.
`uuid`	`java.util.UUID`	Type 4 (randomly-generated) UUID.
`varchar`	`String`	UTF-8 character string.
`varint`	`java.math.BigInteger`	Arbitrary precision integer.

Type conversion

One common point of consternation for many folks new to Cassandra is converting between types. This can be done either by altering the existing table, or coercing the results at query-time with `CAST` . While some of the types may seem interchangeable, issues can arise when converting between types where the target type cannot support the amount of precision required. The following type conversions can be done without issue:

Starting type	Compatible types
`ascii`	`text` , `varchar`
`bigint`	`timestamp` , `varint`
`date`	`timestamp`
`int`	`bigint`
`text`	`varchar`
`time`	`bigint`
`timestamp`	`bigint, varint, date`
`timeuuid`	`UUID`
`varchar`	`text`

Some notes about CQL data type conversion:



The primary key

The most important part of designing your data model is how you define the primary key of your tables. As mentioned previously in this lab, primary keys are built at table-creation time, and have the following options:

```
PRIMARY KEY ((<partition_key>[,additional <partition_key>])[,<clustering_keys])
```

Tables in Apache Cassandra may have (both) multiple partition and clustering keys. As previously mentioned, the partition keys determine which nodes are responsible for a row and its replicas. The clustering keys determine the on-disk sort order within a partition.

Designing a primary key

When designing a primary key, Cassandra modelers should have two main considerations:

- Does this primary key allow for even distribution, at scale?
- Does this primary key match the required query pattern(s), at scale?

Note that on the end of each statement is the phrase at scale. A model that writes all of its table's data into a single partition distributes evenly when you have a 3-node cluster. But it doesn't evenly distribute when you scale up to a 30-node cluster. Plus, that type of model puts your table in danger of approaching the limit of 2,000,000,000 cells per partition. Likewise, almost any model will support high-performing unbound queries (queries without a `WHERE` clause) when they have 10 rows across the (aforementioned) 3-node cluster. But increase that to 10,000,000,000 rows across 30 nodes, and watch the bad queries start to time out.

Apache Cassandra is a great tool for supporting large amounts of data at large scale. But simply using Cassandra for that task is not enough. Your tables must be designed to take advantage of Cassandra's storage and distribution model, or trouble will quickly ensue.

Selecting a good partition key

So, how do we pick a partition key that distributes well at scale? Minding your model's potential cardinality is the key. The cardinality of a partition key represents the number of possible values of the key, ranging from one to infinity.

Note

Avoid extreme ends of cardinality. Boolean columns should not be used as a single partition key (results in only two partitions). Likewise, you may want to avoid UUIDs or any kind of unique identifier as a single partition key, as its high cardinality will limit potential query patterns.

A bad partition key is easy to spot, with proper knowledge of the business or use case. For instance, if I wanted to be able to query all products found in a particular store, store would seem like a good partition key. However, if a retailer has 1,000,000 products, all of which are sold in all 3 of their stores, that's obviously not going to distribute well.

Let's think about a time-series model. If I'm going to keep track of security logs for a company's employees (with multiple locations), I may think about building a model like this:

```
CREATE TABLE security_logs_by_location (
  employee_id TEXT,
  time_in TIMESTAMP,
  location_id TEXT,
  mailstop TEXT,
  PRIMARY KEY (location_id, time_in, employee_id));
```

This will store the times that each employee enters by location. And if I want to query by a specific location, this may work for query requirements. The problem is that,

with each write, the `location_id` partition will get bigger and bigger. Eventually, too many employees will check in at a certain location, and the partition will get too big and become unable to be queried. This is a common Cassandra modeling anti-pattern, known as **unbound row growth**.

Note

Cardinality may also be an issue with `location_id` . If the company in question has 20-50 locations, this might not be so bad. But if it only has two, this model won't distribute well at all. That's where knowing and understanding the business requirements comes into play.

To fix unbound row growth, we can apply a technique called **bucketing**. Let's assume that we know that each building location will only have 300 employees enter each day. That means if we could partition our table by day, we would never have much more than 300 rows in each partition. We can do that by introducing a day bucket into the model, resulting in a composite partition key:

```
DROP TABLE security_logs_by_location;

CREATE TABLE security_logs_by_location (
 employee_id TEXT,
 time_in TIMESTAMP,
 location_id TEXT,
 day INT,
 mailstop TEXT,
 PRIMARY KEY ((location_id, day), time_in, employee_id));
```

Note

Creating a compound partition key is as simple as specifying the comma-delimited partition keys inside parentheses.

Now when I insert into this table and query it, it looks something like this:

```
 INSERT INTO security_logs_by_location (location_id,day,time_in,employee_id,mailstop)
VALUES ('MPLS2',20180723,'2018-07-23 11:04:22.432','aaronp','M266');

SELECT * FROM security_logs_by_location ;

 location_id | day      | time_in                         | employee_id | mailstop
-------------+----------+---------------------------------+-------------+----------
       MPLS2 | 20180723 | 2018-07-23 12:04:22.432000+0000 |      aaronp |     M266

(1 rows)
```

Now my application can write as many rows as it needs to, without worrying about running into too many cells per partition. The trade-off is that when I query this table, I'll need to specify the location and day. But business-reporting requirements typically (in this hypothetical use case) require querying data only for a specific day, so that will work.

Selecting a good clustering key

As mentioned previously, clustering determine the on-disk sort order for rows within a partition. But a good clustering key can also help to ensure uniqueness among the rows. Consider the table used in the preceding examples. This section will make more sense with more data, so let's start by adding a few more rows:

```
 INSERT INTO security_logs_by_location (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 9:04:59.377','tejam','M266');
INSERT INTO security_logs_by_location (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 7:17:38.268','jeffb','M266');
INSERT INTO security_logs_by_location (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 7:01:18.163','sandrak','M266');
INSERT INTO security_logs_by_location (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 6:49:11.754','samb','M266');
INSERT INTO security_logs_by_location (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 7:08:24.682','johno','M261');
INSERT INTO security_logs_by_location (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 7:55:45.911','tedk','M266');
```

Now, I'll query the table for all employees entering the `MPLS2` building between 6 AM and 10 AM, on July 23, 2018:

```
 SELECT * FROM security_logs_by_location
WHERE location_id='MPLS2'
AND day=20180723 AND time_in > '2018-07-23 6:00'
AND time_in < '2018-07-23 10:00';

 location_id | day      | time_in                          | employee_id | mailstop
-------------+----------+----------------------------------+-------------+----------
       MPLS2 | 20180723 | 2018-07-23 11:49:11.754000+0000 |        samb | M266
       MPLS2 | 20180723 | 2018-07-23 12:01:18.163000+0000 |     sandrak | M266
       MPLS2 | 20180723 | 2018-07-23 12:04:22.432000+0000 |      aaronp | M266
       MPLS2 | 20180723 | 2018-07-23 12:08:24.682000+0000 |       johno | M261
       MPLS2 | 20180723 | 2018-07-23 12:17:38.268000+0000 |       jeffb | M266
       MPLS2 | 20180723 | 2018-07-23 12:55:45.911000+0000 |        tedk | M266
       MPLS2 | 20180723 | 2018-07-23 14:04:59.377000+0000 |       tejam | M266

(7 rows)
```

Here are some things to note about the preceding result set:

- As required by the table's `PRIMARY KEY`, I have filtered my `WHERE` clause on the complete partition key (`location_id` and `day`)
- I did not specify the complete PRIMARY KEY, choosing to omit `employee_id`
- The results are sorted by `time_in`, in ascending order; I did not specify `ORDER BY`
- I specified a range on `time_in`, mentioning it twice in the `WHERE` clause, instructing Cassandra to return a range of data
- While Cassandra is aware of my system's time zone, the `time_in` timestamp is shown in UTC time

As per my clustering key definition, my result set was sorted by the `time_in` column, with the oldest value at the top. This is because, while I did clearly specify my clustering keys, I did not specify the sort order. Therefore, it defaulted to ascending order.

Additionally, I omitted the `employee_id` key. I can do that because I specified the keys that preceded it. If I opted to skip `time_in` and specify `employee_id`, this query would fail. There's more on that later.

So why make `employee_id` part of PRIMARY KEY? It helps to ensure uniqueness. After all, if two employees came through security at the exact same time, their writes to the table would conflict. Although unlikely, designating `employee_id` as the last clustering key helps to ensure that a last-write-wins scenario does not occur.

Another good question to ask would be, if Cassandra requires specific keys, how can range query be made to work. Recall that Apache Cassandra is built on a log-based storage engine (LSM tree). This means that building a table to return data in the order in which it is written actually coincides with how Cassandra was designed to work. Cassandra has problems when it is made to serve random reads but sequential reads actually work quite well.

Now assume that the requirements change slightly, in that the result set needs to be in descending order. How can we solve for that? Well, we could specify an `ORDER BY` clause at query time, but flipping the sort direction of a large result set can be costly for performance. What is the best way to solve that? By creating a table designed to serve that query naturally, of course:

```
CREATE TABLE security_logs_by_location_desc (
  employee_id TEXT,
  time_in TIMESTAMP,
  location_id TEXT,
  day INT,
  mailstop TEXT,
  PRIMARY KEY ((location_id, day), time_in, employee_id))
WITH CLUSTERING ORDER BY (time_in DESC, employee_id ASC);
```

If I duplicate my data into this table as well, I can run that same query and get my result set in descending order:

```
 INSERT INTO security_logs_by_location_desc (location_id,day,time_in,employee_id,mailstop)
  VALUES ('MPLS2',20180723,'2018-07-23 9:04:59.377','tejam','M266');
INSERT INTO security_logs_by_location_desc (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 7:17:38.268','jeffb','M266');
INSERT INTO security_logs_by_location_desc (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 7:01:18.163','sandrak','M266');
INSERT INTO security_logs_by_location_desc (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 6:49:11.754','samb','M266');
INSERT INTO security_logs_by_location_desc (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 7:08:24.682','johno','M261');
INSERT INTO security_logs_by_location_desc (location_id,day,time_in,employee_id,mailstop)
 VALUES ('MPLS2',20180723,'2018-07-23 7:55:45.911','tedk','M266');

SELECT * FROM security_logs_by_location_desc
WHERE location_id='MPLS2'
AND day=20180723
AND time_in > '2018-07-23 6:00' AND time_in < '2018-07-23 10:00';

 location_id | day      | time_in                         | employee_id | mailstop
-------------+----------+---------------------------------+-------------+----------
       MPLS2 | 20180723 | 2018-07-23 14:04:59.377000+0000 |       tejam | M266
       MPLS2 | 20180723 | 2018-07-23 12:55:45.911000+0000 |        tedk | M266
       MPLS2 | 20180723 | 2018-07-23 12:17:38.268000+0000 |       jeffb | M266
       MPLS2 | 20180723 | 2018-07-23 12:08:24.682000+0000 |       johno | M261
       MPLS2 | 20180723 | 2018-07-23 12:04:22.432000+0000 |      aaronp | M266
       MPLS2 | 20180723 | 2018-07-23 12:01:18.163000+0000 |     sandrak | M266
       MPLS2 | 20180723 | 2018-07-23 11:49:11.754000+0000 |        samb | M266

(7 rows)
```

In this way, it is clear how picking the right clustering keys (and sort direction) also plays a part in designing tables that will perform well at scale.