USING HADOOP WITH APACHE CASSANDRA

Overview of Cassandra

Cassandra is a noSQL opensource database.It was developed by Facebook to handle their unique needs to process enormous amounts of data.

To say that it is noSQL does not mean it is unstructured. Data in Cassandra is stored in the familiar row-and-column datasets as a regular SQL database. But there are no relations between the tables. And you can query and write data in Cassandra using CQL (Cassandra Query Language), which is very similar to regular SQL.

But just because it supports SQL does not mean you can do all SQL operations on it. In particular there are no JOIN or GROUP operations or anything that would require extension disk searching and calculation. Instead you are supposed to store data in Cassandra the same way that you would like it presented. That shift in thinking is a complete 180 degree turnaround from what people have trability been taught about RDBMS (relational database management systems).

Not-Normal

To illustrate what we mean, we need to discuss what it means to normalize data. Data in Cassandra is supposed to be not-normal and flattened. Let's illustrate that.

Suppose we have this student data:

Student number name city state zipcode

Someone designing an Oracle database would make two tables out of that:

Student number name zipcode

zipcode state city

The common element between the two tables is zipcode. Because when you know the zipcode where someone lives, you know their the city and the state. So why should you repeat that data on every single row? You don't. You make the data normal. But not with Cassandra.
Cassandra is not concerned with using lots of disk space, which is one reason you don't do normalization. (Yet its compression algorithm reduces storage 80%.) Instead Cassandra is all about speed and scale.

Architecture

There is no central hub or master-slave topology with Cassandra. Instead it is designed as a ring of nodes, with every node having the same role. The nodes communicate with each other in what is called **gossip**. Nodes can be added and taken down as needed. The user sets the replication level to indicate how many extra copies of the data to keep to maintain redundancy.

What makes Cassandra different from a regular RDBMS also is that one has to keep in mind how it physically stores and sorts data in order to properly use and fully understand it. That means understanding the CommitLog, Memtable, SSTable, Partitions, and Nodes.

To use an analogy, think of how waves come ashore at the beach. One large wave comes in followed by several smaller ones. Then one giant wave comes ashore and shifts everything around. You can think of the waves as Cassandra memory and the beach sand as Cassandra permanent disk storage.

Cassandra, like Hadoop, is designed to use low-cost commodity storage to deliver a distributed architecture. That means using the hard drives that are attached to the virtual and physical machines in the data center instead of some kind of storage array. Cassandra data is stored in storage as an **SSTable**. Writes are first written to a **CommitLog**, so that it can keep track of what changes it needs to apply. The writes are cached in structures called the **Memtable**. This cache builds up in size until it comes crashing ashore like a wave as Cassandra commits its changes to disk. So actual writes are relatively slow, but reads extremely fast.

There are several of these Memtables being held aloft at any one moment. So the structure of the underlying physical SSTable will be different from the Memtables at any one time. The SSTable is written to in waves, the same as a Linux writes pages of memory to disk when the swap file is full. That is different than a regular RDBMS which update a table each time there is an INSERT or other operation.

Partitions and Nodes

Now, think of a primary key on a database table. The primary key is some unique value coming from one or more fields. In Cassandra the first of these fields denotes the partition key. The other fields in the primary key indicate how data is sorted within that partition. Partitions indicate where data is physically stored (i.e., the node).

For example, you might have this data:

Primary Key	
Vehicle ID	Make
1	Ford
2	Ford
3	Chevrolet
4	Chevrolet
5	Chevrolet

Vehicle ID is the **partition key**. Make is a **clustering column**. This makes data retrieval very efficient if all the Ford vehicles are stored next to each other as it is sorted that way. In this example the vehicle ID is unique so it is not clear how that storage mechanism helps with efficiency. But consider that you can have a **composite partition key**. So suppose the vehicle ID is really the plant where the vehicle was made plus some number. So we would have all these vehicles made at the same plant and painted the same color stored close together to speed retrieval:

Plant	Vehicle ID	Color
Mexico	1	red
Mexico	1	red
USA	1	blue

Now, let's do some hands-on work with Cassandra to further illustrate these concepts. First we install the software.

Installation

Here is the installation on CentOS. One point to notice here is that on the Cassandra website, Cassandra makes their software available as a download and distributes it to Ubuntu repositories, but not for Yum. So Datastax does that.

Note:The private business Datastax has written most of the documentation for Cassandra on the internet. They used to make their own distribution of Cassandra but no longer do that. Now someone needs to step into the void and finish the documentation as the official Cassandra documentation is filled with plenty of To-Do comments noting which pages they need to complete. In fact they are even looking for persons who want to help finish the documentation. So use Datastax documentation for now.

First create a repo file with this content:

```
cat /etc/yum.repos.d/datastax.repo

name = DataStax Repo for Apache Cassandra
baseurl = http://rpm.datastax.com/community
enabled = 1
gpgcheck = 0
```

Then install these two items:

```
yum install dsc30
yum install cassandra30-tools
```

Now open the Cassandra shell:

```
cqlsh

Connected to Test Cluster at 127.0.0.1:9042.

Use HELP for help.
cqlsh>
```

Now lets create a table and put some data in it.

KeySpace

The first object to create is a **KeySpace**. This is a container for replication. Here we make a KeySpace called Library, as we are going to make a table to keep track of which books a person has checked out of the library in our little example.

The **class** and **replication_factor** determine how data is replicated. There are many options for that, as there are for partition dispersal. Here we say to make 3 copies of each data.

```
CREATE KEYSPACE Library
      WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' :
3 };
```

Tables

Now we create two tables: book and patron.

```
CREATE TABLE Library.book (
ISBN text,
copy int,
title text,
PRIMARY KEY (ISBN, copy)
 );

CREATE TABLE  Library.patron (
ssn int PRIMARY KEY,
checkedOut set <text>
);
```

There are some items to notice here:

- We use the field types text and int. Cassandra has the usual column types plus allows user-defined ones.
- The first table has 2 fields in the primary key. The second table has 1.
- The second table has a set collection column. A set is a collection of unique elements, like {1,2,3}. Cassandra also supports a list, whose elements do not need to be unique, like {1,2,2,3} and a map, where the key has to be unique, like {key->value}.
- Other than that the syntax looks exactly like regular Oracle SQL.

Add Data

Let's add some data. The commands look just like regular SQL. The table name is prefixed by the Keyspace, just like in a regular SQL database the table is denoted database.table.

```
INSERT INTO  Library.book (ISBN, copy, title) VALUES('1234',1, 'Bible');
INSERT INTO  Library.book (ISBN, copy, title) VALUES('1234',2, 'Bible');
INSERT INTO  Library.book (ISBN, copy, title) VALUES('1234',3, 'Bible');
INSERT INTO  Library.book (ISBN, copy, title) VALUES('5678',1, 'Koran');
INSERT INTO  Library.book (ISBN, copy, title) VALUES('5678',2, 'Koran');
```

Now list the values:

```
select * from Library.book;


 isbn | copy | title
------+------+-------
 5678 |    1 | Koran
 5678 |    2 | Koran
 1234 |    1 | Bible
 1234 |    2 | Bible
 1234 |    3 | Bible
```

Next we add data to the patron table. The books that the library patron has checked out is a set: {'1234','5678'}.

In a normalized database, the books and the library patron would be kept in separate tables. But here we flatten everything into one structure to avoid having to do SQL JOIN and other operations that would take time. So even though we put details about the books in the book table we could have added the title, page count, etc. as in the patron table too, in perhaps a tuple or other data structure. Again that is completely the opposite of what programmers have traditionally been taught when they design database tables.

So add some data and print it out:

```
INSERT INTO Library.patron (ssn, checkedOut) values (123,{'1234','5678'});
cqlsh> select * from Library.patron;


 ssn | checkedout
-----+------------------
 123 | {'1234', '5678'}

 INSERT INTO Library.patron (ssn, checkedOut) values (123,{‘1234’,’5678’});
```

Now, Cassandra is all about performance. So you cannot query fields in the collections column without first making an index:

```
create index on Library.patron (checkedOut);
```

Now we can we can show which patrons have check out book number 1234 with a query. Note that we use the contains operator.

```
select ssn from Library.patron where checkedOut contains '1234';

 ssn
-----
 123
```

This is an overview of Cassandra to get you started. Now you might want to investigate some of the APIs as there are Cassandra APIs for many programming languages. Also you could read use cases as because there are so many big data databases now it would be helpful to see which type people have used in what situation.