CASSANDRA

Arash Akhlaghi, Badrinath Jayakumar, Wa’el Belkasim

Instructor: Dr. Rajshekhar Sunderraman

CSC 8711 Project Report
Introduction

The relational model was brought by E.F. Codd’s 1970 paper which made data modeling and application programming much easier. In addition to that, it is used for client server programming and for storing structured data on the cloud. After the arrival of web 2.0 applications, data stores need to large scale OLTP /OLAP application loads. In this circumstance, the relational model does not provide good results. This is the gap which NoSQL systems attempt to fill, by providing a more scalable solution with high availability.

History of Cassandra

This illustrates the brief history of Cassandra. Cassandra is one of the top level projects in Apache and it is open source.
**Cassandra**

Cassandra is a column oriented, eventually consistent, distributed storage system for managing very large amounts of structured data. The Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency.

**Why Cassandra?**

At Facebook, the inbox search system needed to handle a very high write throughput, billions of writes per day and was also required to scale to a very large amount of users. To keep search latency down, data has to be replicated to data centers which are used by different users in different geographic locations.

**What is eventually consistent?**

Building reliable distributed systems on a global scale demands trade-offs between consistency and availability. Consistency in a nutshell means that when something is written, it is expected that all reads after the write will have access to that written data. In Cassandra, due to its distributed nature, there are no such hard guarantees. However, we can say that it eventually reaches a consistent state because all data is eventually replicated across the distributed data store.

**CAP Theorem**

Brewer’s theorem states that in any given system, you can strongly support only two of the following three:

- Consistency
- Availability
- Partition Tolerance.

Figure 2.0 illustrates that there is no overlapping segment where all three are obtainable.
The CAP theorem visualized as a Venn diagram.

The list of companies using Cassandra is vast and constantly growing. This list includes:

- **Twitter** is using Cassandra for analytics.
- **Mahalo** uses it for its primary near-time data store.
- **Facebook** still uses it for inbox search, though they are using a proprietary fork.
• **Digg** uses it for its primary near-time data store.
• **Rackspace** uses it for its cloud service, monitoring, and logging.
• **Reddit** uses it as a persistent cache.
• **Cloudkick** uses it for monitoring statistics and analytics.
• **Ooyala** uses it to store and serve near real-time video analytics data.
• **SimpleGeo** uses it as the main data store for its real-time location infrastructure.
• **Onespot** uses it for a subset of its main data store.

## Interfacing with Cassandra

Users can interact with Cassandra in multiple ways.

1. **Command Line Interface (CLI)**
   The latest version is 1.2.4. For the purpose of learning, we worked on this project in CLI.
2. **Cassandra Query Language (CQL)**
   It supports subset of SQL features. Here standard DDL, DML commands can be used. Specific functions like Group by and Order by are not supported. The latest version is 3.0. Because of trivialness, this report will not discuss CQL in detail.

The **DataStax Community Package** is a software package which supports both the CLI and CQL together and is a quick way to interface with Cassandra.

## Architecture Overview

Cassandra was designed with the understanding that system/hardware failures will and do occur. Due to this, Cassandra was developed as a peer to peer distributed system where all nodes serve the same functions, meaning there is **no single point of failure**. One of Cassandra’s greatest strength is its availability and scaling, it achieves this through a fully distributed system where data is replicated across multiple nodes according to user settings.

Data reading and writing is abstracted away from the application, which allows the application to read/write to any node in the system and always expect that the data is replicated across multiple nodes. According to the tunable consistency setting, the user can also achieve
complete consistency, a compromise of consistency and speed, or little to no consistency. Figure 4.0 illustrates the life cycle of a data write in Cassandra at a high level.

1. Client Writes to any Cassandra Node
2. Coordinator Node replicates to nodes and Zones
3. Nodes return ack to coordinator
4. Coordinator returns ack to client
5. Data written to internal commit log disk

If a node goes offline, hinted handoff completes the write when the node comes back up.

Requests can choose to wait for one node, a quorum, or all nodes to ack the write

SSTable disk writes and compactions occur asynchronously

**Fig 4.0** – Cassandra write cycle as shown by Netflix’s Cassandra production rollout team.

In figure 4.0 the clients are a set of applications or users, and the surrounding nodes are the Cassandra peer to peer nodes. Once the client initiates a system request to read/write, it selects a node based on region, cluster, or just randomness. Since Cassandra is decentralized, any node can function as a coordinator node, it is simply the first node which the client connects to for a request. After which the request will be initiate on that node, and replicated to other nodes through Cassandra’s peer-to-peer Gossip protocol. In the case of figure 4.0, the coordinator node chooses to wait for 3 nodes to return an acknowledgement before responding to the client. This is determined by the consistency setting, which allows the administrator to choose the level of consistency and availability during a write. The data is first written to the commit log, in memory, after which Cassandra manages the compression and disk writing asynchronously according to the Memtable and SSTable settings, for the best performance.
Strengths of Cassandra

- Gigabyte to Petabyte scalability
- Flexible Schema Design
- Linear Performance of gains through node addition
- Cassandra Query Language (like SQL)
- Data Compression
- No need for separate caching layer
- Tunable data consistency

Map Reduce

Cassandra does not support the map reduce programming model. However, it’s possible to perform map reduce operations by integrating Cassandra with Hadoop. In this report, we will not be focusing on this non-core functionality.

Data Model - Bottom up approach

For people coming from traditional RDBMs, the Cassandra data model can be strange, confusing and maybe even a bit difficult to understand. There are some terms such as keyspace completely new in Cassandra and some terms such as column does not match the meaning in the RDBMs.

Before we dig into some of key data model concepts in Cassandra following bottom up approach we would like to illustrate how Cassandra data model can be mapped to RDBMs. This illustration is only for the purpose of giving some rough idea to developers coming from RDBMs world and making it easier for them to understand Cassandra. They should consider this as an exact match.
Cluster

It is not probably the best choice if you only have one node in Cassandra. Usually a user creates several nodes that will form a ring and uses a peer-to-peer protocol for data replication. So in case of one node going down the node having the data replica responds to the query. The ring is the outer most structure in Cassandra. They call this ring cluster.

Keyspaces

The key space is the outermost container for data in Cassandra. It resembles the database in relational database. A cluster can hold several key spaces. In Cassandra when you create key space there are set of basic attributes you can set for it as follow:

a) **Replication Factor**: The number of nodes holding the replica of data
b) **Replica Placement Strategy**: The strategy the replication should happen
c) **Column Families**: Like a database having several tables a Keyspace can define many column families

Column

The basic building block in Cassandra is a column. The column is constructed of three parts: column name, column value and a timestamp. The column name and value are simply
saved as Java byte arrays. The timestamp is implementation of Cassandra IClock interface. The timestamp is used for internal Cassandra use and is not accessible or editable by the user.

![Column](image)

**Fig 5.0 - Column in Cassandra**

Here’s an example of a column you might define, represented with JSON notation just for clarity of structure:

```json
{
  "name": "email",
  "value": "me@example.com",
  "timestamp": 1274654183103300
}
```

**Column Family**

A column family is a container for an ordered collection of rows, each of which is itself an ordered collection of columns. As you can see in the Figure 1.0 the column family resembles the table in a relational database. Though there are several reasons that we should not go too far in this comparison. First, Cassandra is considered schema free. You define the column families but you don’t define the columns. You can freely add any column to any column family at any time, depending on your needs. Second, a column family has two attributes: a name and a comparator. The comparator value indicates how columns will be sorted when they are returned to you in a query—according to long, byte, UTF8, or other ordering.

A collection of these columns constitutes a row in Cassandra. The primary key or unique identifier of each row will be called row-key. So a column family will be number of rows and each row can have variable length of columns. In Cassandra the design of many aspects including the data model concentrates on performance so anything that could help better performance has been considered such as variable length of columns.
There are several options that we can define for each column family that we are going to mention briefly here:

**keys_cached:** The number of locations to keep cached per SSTable. SSTable stands for sorted string table and are internal structure saving data on disk in Cassandra.

**rows_cached:** The number of rows that their whole content will be cached.

**Comment:** This is just a standard comment that helps you remember important things about your column family definitions.

**read_repair_chance:** This is a value between 0 and 1 that represents the probability that read repair operations will be performed when a query is performed without a specified quorum, and it returns the same row from two or more replicas and at least one of the replicas appears to be out of date. You may want to lower this value if you are performing a much larger number of reads than writes.

**preload_row_cache:** Specifies whether you want to prepopulate the row cache on server startup.
Read and Write in Cassandra

Fig 7.0 The Read and Write Process in Cassandra

Fig 7.0 shows the mechanism that Cassandra uses to read and write data. The design considerations are for the purpose of better performance. The figure clearly shows the division of memory and disk. For writes, Cassandra uses commit log to perform the quick operation then the data goes from Commit log to memtable. Memtable is volatile and purpose of high read operations. When memtable reaches a certain thresholds gets flushed to SSTable on disk. For read, Cassandra uses memtable to access the data mostly. Due to such design, Cassandra gives great performance in most operations especially on writes. The following graph compares Cassandra with some other No-SQL databases.

Fig 8.0 Cassandra Operation Speed Comparisons
Tunable Data Consistency

Cassandra is designed for high availability and partition failure tolerance. There is a bit sacrifices regarding consistency but the user based on its needs can tune Cassandra and put more weight on consistency. Consequently this might have a bit adverse effect on performance. Cassandra gives several options regarding this and is based on operations.

**Strategy for Writes:**

**ZERO:** The write operation will return immediately to the client before the write is recorded; the write will happen asynchronously in a background thread, and there are no guarantees of success.

**ANY:** Ensure that the value is written to a minimum of one node, allowing hints to count as a write.

**ONE:** Ensure that the value is written to the commit log and memtable of at least one node before returning to the client.

**QUORUM:** Ensure that the write was received by at least a majority of replicas ((replication factor / 2) + 1).

**ALL:** Ensure that the number of nodes specified by replication factor received the write before returning to the client. If even one replica is unresponsive to the write operation, fail the operation.

**Strategy for Reads:**

**ZERO:** Unsupported. You cannot specify CL.ZERO for read operations because it doesn’t make sense. This would amount to saying “give me the data from no nodes.”

**ANY:** Unsupported. Use CL.ONE instead.

**ONE:** Immediately return the record held by the first node that responds to the query. A background thread is created to check that record against the same record on other replicas. If any are out of date, a read repair is then performed to sync them all to the most recent value.

**QUORUM:** Query all nodes. Once a majority of replicas ((replication factor / 2) + 1) respond, return to the client the value with the most recent timestamp. Then, if necessary, perform a read repair in the background on all remaining replicas.
**ALL:** Query all nodes. Wait for all nodes to respond, and return to the client the record with the most recent timestamp. Then, if necessary, perform a read repair in the background. If any nodes fail to respond, fail the read operation.

**Client**

Cassandra has no drivers. JDBC abstracts database interactions from the developers’ point of view, and in the same way, the Cassandra client layer does the same but using a different mechanism. The client generation layer is provided Thrift API which is a low-level client. The following are high level clients for Cassandra.

- **Python:**
  - Pycassa
  - Telephus
- **Java:**
  - Firebrand:
    - PlayOrm:
    - Astyanax
    - Hector
    - Kundera
    - Pelops
    - Easy-Cassandra
    - Cassandrelle (Demoiselle Cassandra)
    - Feedly-Cassandra (ORM library)
- **Scala**
  - Cascal
- **Node.js**
  - Helenus
- **Clojure**
  - clj-hector
  - casyn
  - alia
  - hayt (CQL3 query generation)
- **.NET**
  - Aquiles
  - Cassandraemon
  - cassandra-sharp
In our application we make use of the Hector client. Hector is an open source project written in Java and released under the MIT license. It wraps the Thrift API and offers JMX capability, connection pooling, and failover.

**Features**

Hector is a well-supported and fully-featured Cassandra client, with many users and an active community.

**High-level object-oriented API**

Java developers see the Keyspace and Columnfamily interfaces are very natural to use and handy.
**Fail over support**

In hector, when the communicating node goes down, it is able to recover without user intervention. The API will help to service client requests by handing off the request to any other available working node. Thrift has no fail over support.

**Connection pooling**

Cassandra is known mainly for its scalability, if the client doesn’t support connection pooling, we end up having many new connections which will ultimately affect Cassandra’s speed. If there is no connection pooling, any NO SQL system becomes useless in terms of speed. Every client is expected to implement connection pooling. Hector’s connection pooling uses Apache’s GenericObjectPool.

**JMX support**

Cassandra makes liberal use of JMX, which comes in very handy for monitoring. Hector directly supports JMX by exposing metrics such as bad connections, available connections, idle connections, and other information.

**Demo Application**

The below figure shows the schema design of our application.

**Fig 9.0 Schema Diagram**
The following are the description about each column family.

**ODIPLAYER**

This column family stores different country’s player’s information. Here row key is Player ID which is unique. Name, country, images are the columns for the row key.

**PLAYERBYCOUNTRY**

This column family is created to make the retrieval of players based on country easier. Here row key is country name. It has a special column called Flag which stores the national flag of the country.

**MATCHESPLAYED**

This column family contains each player’s match history. Here row key is player id.

All column families are stored under the *keyspace* named *Cricket*.

The following CLI code creates Keyspace and Columnfamilies in Cassandra.

```sql
create keyspace Cricket
with strategy_options = {replication_factor:1}
and placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy';

use Cricket;
create column family ODIPLAYER
with comparator = UTF8Type
and default_validation_class = 'UTF8Type'
and key_validation_class = 'UTF8Type';

create column family PLAYERBYCOUNTRY
with comparator = UTF8Type
```
and default_validation_class = 'UTF8Type'
and key_validation_class = 'UTF8Type';

create column family MATCHESPLAYED
with comparator = UTF8Type
and default_validation_class = 'UTF8Type'
and key_validation_class = 'UTF8Type';

To run the application

Please make sure Cassandra 1.x.x is installed properly on your machine. Install Maven plugin in your eclipse. Please don’t change any default settings (like port, cluster name etc) in Cassandra. Run the above Key space and column families script. Now, the application is ready to go.

Note: Check the availability of the Cassandra server while running the application.

Our recommendations to use Cassandra

1. A Unix-based system is highly recommended. JNA (Java Native Access Library) for windows does not work.
2. If you prefer to use Java, then Maven plugin for eclipse is highly recommended to resolve dependency issues.
3. The DataStax tool is the easiest way to create clusters and nodes rather than directly changing the Cassandra.yaml configuration file.
Source Code

The following is the source code of our Cricket CURD (Create Update Read Delete) application. The whole application is built on these 4 operations.

Create
The following snippet adds matches for a player (referred by his id ). Mutator is the handler for the keyspace initialized with UTF-8 encoding.

```java
public static void addMatches(String pid1, String trophy, String venue,
                               String runs, String wkts)
{
    cluster = HFactory.getOrCreateCluster("Test Cluster", "localhost:9160");

    try {
        keyspace readkeyspace = HFactory.createKeyspace("Cricket", cluster);
        // Create a mutator object for this keyspace using utf-8 encoding
        Mutator<String> mutator = HFactory.createMutator(readkeyspace,
                                                            stringSerializer);
        int c = countNoofColumns("MATCHESPLAYED", pid1);
        int n = (c/4)+1;

        // mutator.addInsertion("ROW KEY ", "CF", NAME/VALUE PAIR);
        mutator.addInsertion(pid1, "MATCHESPLAYED",
                              HFactory.createStringColumn("Trophy"+n, trophy));
        mutator.addInsertion(pid1, "MATCHESPLAYED",
                              HFactory.createStringColumn("venue"+n, venue));
        mutator.addInsertion(pid1, "MATCHESPLAYED",
                              HFactory.createStringColumn("run scored"+n, runs));
        mutator.addInsertion(pid1, "MATCHESPLAYED",
                              HFactory.createStringColumn("wickets taken"+n, wkts));

        // everything was set. Execute stmt will run it.
        MutationResult mr = mutator.execute();
        System.out.println("Matches Inserted");
        System.out.println();
    }
    catch (Exception ex) {
        System.out.println("Error encountered while inserting data!!");
        ex.printStackTrace();
    }
}
```
Read

The following snippets will read all players for the given country name. Here interesting classes are `createSliceQuery` and `ColumnSliceIterator`.

`createSliceQuery`

It provides provision to navigate all name-value pair by row key.

`ColumnSliceIterator`

For a row key, instead of bringing all key value into memory, column slideIterator brings batch by batch.

```java
public static ArrayList<String> getPlayerbyCountry(String country) {
    ArrayList<String> al = new ArrayList<String>();
    ArrayList<String> name,country;
    cluster = HFactory.getOrCreateCluster("Test Cluster", "localhost:9160");
    pid= new ArrayList<String>();
    name= new ArrayList<String>();
    country= new ArrayList<String>();
    pimage = new ArrayList<String>();
    Keyspace readkeyspace = HFactory.createKeyspace("Cricket", cluster);
    SliceQuery<String, String, String> query = HFactory.createSliceQuery(readkeyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get()).setKey(country.trim()).setColumnFamily("PLAYERBYCOUNTRY");
    ColumnSliceIterator<String, String, String> iterator = new ColumnSliceIterator<String, String, String>(query,
        null, "\uFFFF", false);
    while (iterator.hasNext()) {
        HColumn<String, String> abc = iterator.next();
        if(!abc.getName().contains("Image"))
            al.add(abc.getName());
        System.out.println(abc.getName());
    }
    int len =al.size(); String s;
    for(int k =0 ;k<len;k++)
        { s= al.get(k);
    SliceQuery<String, String, String> query1 = HFactory.createSliceQuery(readkeyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get()).setKey(s.toString().trim()).setColumnFamily("ODIPLAYER");
    ColumnSliceIterator<String, String, String> iterator1 = new ColumnSliceIterator<String, String, String>(query1,
        null, "\uFFFF", false);
```
Update

The following snippets update player’s match details. Here, we had created column family template to update. This is one of many ways to update. Just to show other features in Hector API we used template here.

```java
public static void updateMatches(String pid1, String trophy, String venue, String runs, String wkts, int rno) {
    ArrayList<String> al = new ArrayList<String>();
    cluster = HFactory.getOrCreateCluster("Test Cluster", "localhost:9160");

    Keyspace readkeyspace = HFactory.createKeyspace("Cricket", cluster);

    SliceQuery<String, String, String> query = HFactory.createSliceQuery(readkeyspace, StringSerializer.get(), StringSerializer.get(), StringSerializer.get()).setKey(pid1).setColumnFamily("MATCHESPLAYED");

    ColumnSliceIterator<String, String, String> iterator = new ColumnSliceIterator<String, String, String>(query, null, "\uFFFF", false);
    while (iterator.hasNext()) {
        HColumn<String, String> abc1 = iterator.next();
        if(abc1.getName().contains("Name"))
            {
                name.add(abc1.getValue());
                System.out.println(abc1.getValue());
            }
        if(abc1.getName().contains("Country"))
            {
                country.add(abc1.getValue());
            }
        if(abc1.getName().trim().equals("Image"))
            {
                if(!pimage.contains(abc1.getValue()))
                    {
                        pimage.add(abc1.getValue());
                        System.out.println(abc1.getValue());
                    }
            }
    }
}
```
while (iterator.hasNext()) {
    HColumn<String, String> abc = iterator.next();
    if (abc.getName().contains("Trophy")) {
        al.add(abc.getName().substring(6));
    }
    System.out.println(abc.getName().substring(6));
}

ColumnFamilyTemplate<String, String> template =
    new ThriftColumnFamilyTemplate<String, String>(readkeyspace,
        "MATCHESPLAYED",
        StringSerializer.get(), StringSerializer.get());
ColumnFamilyUpdater<String, String> updater = template.createUpdater(pid1);
updater.setString("Trophy"+al.get(rno), trophy);
updater.setString("venue"+al.get(rno), venue);
updater.setString("run scored"+al.get(rno), runs);
updater.setString("wickets taken"+al.get(rno), wkts);
try {
    template.update(updater);
    System.out.println("a ODIPLAYER updated Successfully");
    System.out.println();
} catch (HectorException e) {
    System.out.println("Error encountered while updating data!!");
    e.printStackTrace();
}

Delete
Here, we used the sample template feature to delete matches for a particular player.

public static void deleteMatches(String pid1, int i) {
    ArrayList<String> al = new ArrayList<String>();
    cluster = HFactory.getOrCreateCluster("Test Cluster", "localhost:9160");
    Keyspace readkeyspace = HFactory.createKeyspace("Cricket", cluster);
    SliceQuery<String, String, String> query =
        HFactory.createSliceQuery(readkeyspace, StringSerializer.get(),
            StringSerializer.get(), StringSerializer.get().
            setKey(pid1).setColumnFamily("MATCHESPLAYED");
ColumnSliceIterator<String, String, String> iterator =
   new ColumnSliceIterator<String, String, String>(query, null, "\uFFFF", false);

while (iterator.hasNext()) {
   HColumn<String, String> abc = iterator.next();
   if (abc.getName().contains("Trophy")) {
       al.add(abc.getName().substring(6));
   }
   System.out.println(abc.getName().substring(6));
}

ColumnFamilyTemplate<String, String> template =
   new ThriftColumnFamilyTemplate<String, String>(readkeyspace, "MATCHESPLAYED",
   StringSerializer.get(), StringSerializer.get());

// template.deleteColumn ROWKEY,COLUMN NAME
template.deleteColumn(pid1, "Trophy"+al.get(i));
template.deleteColumn(pid1, "venue"+al.get(i));
template.deleteColumn(pid1, "run scored"+al.get(i));
template.deleteColumn(pid1, "wickets taken"+al.get(i));
System.out.println("a column deleted Successfully");
System.out.println();
Screen Shots

Fig. 10. Homepage

Country Names

India  Russia

England  Sri Lanka

Fig 11. List of Country Names
Fig. 12 Adding Country

Fig. 13 Players by Country
**Fig. 14** Player details

**Fig. 15** Add player
Conclusion

Cassandra is a robust solution for those requiring a reasonably consistent, highly available, and scalable fault-tolerant data store. Cassandra is also a great solution for those migrating from relational databases to No SQL due to the Cassandra Query Language, which is essentially a subset of SQL, making Cassandra more accessible than competitors. Cassandra maintains itself as a leader in speed and efficiency within the No SQL domain, so if the goal is to create an application with intensive and quick reads and writes, then Cassandra is the ideal solution.

References

2. Datastax Enterprise 3.0 Documentation
3. Cassandra - A Decentralized Structured Storage System Avinash Lakshman, Prashant Malik