Abstract— The Apache Cassandra Project develops a highly scalable, eventually consistent, distributed, structured second-
generation distributed database. Cassandra is a key-value store. 
Cassandra brings together the distributed systems technologies 
from Dynamo and the data model from Google's BigTable. Like 
Dynamo, Cassandra is eventually consistent. Like BigTable, 
Cassandra provides a ColumnFamily-based data model richer 
than typical key/value systems.

In this paper, I briefly illustrate the technology behind 
Cassandra and its working.

I. INTRODUCTION

Cassandra brings together Dynamo's fully distributed 
design and Bigtable's ColumnFamily based data model. 
Cassandra is basically a one hop DHT i.e. O(1) DHT. It is 
eventually consistent with tunable trade-offs between 
consistency and latency. It is more than a simple DHT because 
in Cassandra the values are not opaque, but they are structured 
into columns and columnFamilies and they are indexed. 

Following are the key features of Cassandra

Distributed and Decentralized:
Cassandra is decentralized, meaning that every node is 
identical; no Cassandra node performs certain organizing 
operations distinct from any other node. Cassandra features a 
peer-to-peer protocol and use gossip to maintain and keep in 
sync a list of nodes. As a result of this, there is no single point 
of failure.

Horizontally scalable:
Cassandra allows us to scale extensive structured datasets 
horizontally over inexpensive commodity hardware. The was 
Cassandra does this is by allowing new nodes to participate by 
getting copies of a part or all of the data and start serving 
requests without major disruptions. No restarting of processes, 
no manual rebalancing required.

High availability:
Cassandra is highly available. We can replace the failed 
nodes without any downtime and can replicate data to multiple 
data centers to improve local performance.

Tunable Consistency:
Cassandra trades off some consistency in order to achieve 
total availability. But Cassandra can allow you to tune the 
level of consistency you need, in balance of the availability.

II. CASSANDRA DATA MODEL

Cassandra is based on a Key-Value model like BigTable. 
Unlike a typical RDBMS which consists of rows tables and 
columns, Cassandra consists of column families. A column 
family is a set of key-value pairs. Drawing an analogy with 
relational databases, column family can be thought of as a 
table and a key-value pair as a record in a table.

There are two levels of nesting:

i. At the first level the value of a record is in turn a 
sequence of key-value pairs. These nested key-
value pairs are called columns where key is the 
name of the column. In other words you can say 
that a record in a column family has a key and 
consists of columns. This level of nesting is 
mandatory. A record must contain at least one 
column.

![Column families](image)

ii. At the second level, which is arbitrary, the value of a 
nested key-value pair can be a sequence of key-
value pairs as well. When the second level of 
nesting is presented, outer key-value pairs are 
called super columns with key being the name of 
the super column and inner key-value pairs are 
called columns.
The names of both columns and super columns can be used in two ways: as names or as values (usually reference value).

i. First, names can play the role of attribute names. For example, the name of a column in a record about User can be Email. That is how we used to think about columns in relational databases.

ii. Second, names can also be used to store values! For example, column names in a record which represent Blog can be identifiers of the posts of this blog and the corresponding column values are posts themselves. You can really use column (or super column) names to store some values because (a) theoretically there is no limitation on the number of columns (or super columns) for any given record and (b) names are byte arrays so that you can encode any value in it.

III. PIECES OF DATA MODEL WITH EXAMPLES

A. Column:
The column is the lowest/smallest increment of data. It’s a triplet that contains a name, a value and a timestamp.

```
{ // this is a column
    name: "emailAddress",
    value: "msoparia@indiana.edu ",
    timestamp: 7168680886
}
```

B. SuperColumn:
A SuperColumn is a tuple w/ a binary name & a value which is a map containing an unbounded number of Columns–keyed by the Column’s name.

```
{ // this is a SuperColumn
    name: "homeAddress",
    // with an infinite list of Columns
    value: {
        // note the keys is the name of
        // the Column
        street: {name: "street", value: "2602 Eastgate Lane",
            timestamp: 7168680886},
        city: {name: "city", value: "Bloomington",
            timestamp: 7168680886},
        zip: {name: "zip", value: "47408",
            timestamp: 7168680886},
    }
}
```

C. ColumnFamily:
A ColumnFamily is a structure that contains an infinite number of Records. Each Record has a client supplied (that means you) key & contains a map of Columns. Again, the keys in the map are the names of the Columns and the values are the Columns themselves.

```
UserProfile = { // this is a ColumnFamily
    msoparia: {   // this is the key to
        // this Row inside the CF
        // now we have an infinite # of
        // columns in this row
        username: "msoparia",
        email: "msoparia@indiana.edu ",
        phone: "(812) 222-6666"
    }, // end row
    rini2711: {   // this is the key to
        // another row in the CF
        // now we have another infinite #
        // of columns in this row
        username: "rini2711",
        email: "rini2711@gmail.com",
        phone: "(982) 064-9580"
    }, // end row
}
```

D. Super ColumnFamily:
A ColumnFamily can be of type Standard or Super. When a ColumnFamily is of type Super, each Row contains a map of SuperColumns. The map is keyed with the name of each SuperColumn and the value is the SuperColumn itself. Since this ColumnFamily is of type Super, there are no Standard ColumnFamily’s in there.

```
AddressBook = {
    msoparia: {
        friend1: {street: "3rd street",
            zip: "47401", city: "Bloomington", state: "IN"},
    }
}
```
Friend2: {street: "10th street", 
zip: "3345", 
city: "NYC", 
state: "NY"},

... 

rini2711: {
    peter: {street: "Dunn ave", 
            zip: "47331", 
city: "LA", 
state: "CA"},
},

IV. SYSTEM ARCHITECTURE

A. Partitioning:

One of the key design features for Cassandra is the ability to scale incrementally. This requires, the ability to dynamically partition the data over the set of nodes (i.e., storage hosts) in the cluster. Cassandra partitions data across the cluster using consistent hashing and has an option of two types of Partitioners to do so:

i. RandomPartitioner: This partitioner randomly distributes the key-value pairs over the network, resulting in a good load balancing. Compared to OrderPreservingPartitioner, more nodes have to be accessed to get a number of keys. Random partitioner uses 128bit namespace and MD5 algorithm.

ii. OrderPreservingPartitioner: This partitioner distributes the key-value pairs in a natural way so that similar keys are not far away. The advantage is that, lesser nodes have to be accessed. The drawback is the uneven distribution of the key-value pairs. In this kind of partitioning Tokens determine the namespace and keys are arranged lexicographically.

Using OPP provides you with two obvious advantages over RP:

i. We can perform range slices. That is we can scan over ranges of our rows as though we were moving a cursor through a traditional index. For example, if we are using user ids as our keys, we could scan over the rows for users whose names begin with R e.g. ramus, rohan, rini etc.

ii. We can store real time full text indexes inside Cassandra, which are built using the aforementioned feature.

iii. We can scan over our data to recover/delete orphaned keys.

B. Replication

Cassandra uses replication to achieve high availability and durability. Each data item is replicated at N hosts, where N is the replication factor configured “per-instance”. Each key, k, is assigned to a coordinator node. The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 nodes in the ring. Cassandra provides the client with various options for how data needs to be replicated. Cassandra provides various replication policies such as “Rack Unaware”, “Rack Aware” (within a datacenter) and “Datacenter Aware”. Replicas are chosen based on the replication policy chosen by the application. If certain application chooses “Rack Unaware” replication strategy then the non-coordinator replicas are chosen by picking N-1 successors of the coordinator on the ring. For “Rack Aware” and “Datacenter Aware” strategies the algorithm is slightly more involved. All nodes on joining the cluster contact the leader who tells them for what ranges they are replicas for and leader makes a concerted effort to maintain the invariant that no node is responsible for more than N-1 ranges in the ring. The metadata about the ranges a node is responsible is cached locally at each node. This way a node that crashes and comes back up knows what ranges it was responsible for.

C. Bootstrapping:

When a node starts for the first time, it chooses a random token for its position in the ring. For fault tolerance, the mapping is persisted to disk locally. The token information is then gossiped around the cluster. This is how we know about all nodes and their respective positions in the ring. This enables any node to route a request for a key to the correct node in the cluster. In the bootstrap case, when a node needs to join a cluster, it reads its configuration file which contains a list of a few contact points within the cluster. We call these initial contact points, seeds of the cluster.

D. Scaling:

When a new node is added into the system, it gets assigned a token such that it can alleviate a heavily loaded node. This results in the new node splitting a range that some other node was previously responsible for. The Cassandra bootstrap algorithm is initiated from any other node in the system by an operator. The node giving up the data streams the data over to the new node using kernel copy techniques.

V. CONCLUSION

Cassandra can support a very high update throughput while delivering low latency. Cassandra is used by Digg, Twitter, Facebook and many other applications which have huge amount of data to manage.

REFERENCES