1 Data Model

Project Model Overview

- HBase is using Column-families data model. Applications store data into an HBase table. Tables are made of rows and columns. All columns in HBase belong to a particular column family.

- HBase have several objects within the data model: collections, column families, table, rows, columns, cell, key value pairs.

Manipulating API by native Java client API:

Create a new table:

```
/* Creates a Configuration with HBase resources */
Configuration conf = HBaseConfiguration.create();

/* Provides an interface to manage HBase database table metadata
*/
HBaseAdmin admin = new HBaseAdmin(conf);

/* HTableDescriptor contains the details about an HBase table
(e.p. Column family)
   In this example, we input an argument to constructor to create
a table name school */
HTableDescriptor tableDescriptor = new
HTableDescriptor(TableName.valueOf("school"));

/* Add two new column family: dept, building */
tableDescriptor.addFamily(new HColumnDescriptor("school-info"));
tableDescriptor.addFamily(new HColumnDescriptor("dept-info"));

/* Create the table */
admin.createTable(tableDescriptor);

/* Create an object to interact with a HBase table */
HTable table = new HTable(conf, "school");
```

```
     /* Create an object (tool) to insert data, need argument to
specify key for the new row.
        In this example, our key(unique) would be "PSU" and "CS".*/
     Put put = new Put(Bytes.toBytes("PSU-CS"));

     /* Add values by the following format: add(<column-family>,
<column>, <data>
        Value are all byte arrays. */
     put.add(Bytes.toBytes("school-info"), Bytes.toBytes("s-name"),
Bytes.toBytes("PSU"));
     put.add(Bytes.toBytes("school-info"), Bytes.toBytes("s-city"),
Bytes.toBytes("Portland"));
     put.add(Bytes.toBytes("dept-info"), Bytes.toBytes("d-name"),
Bytes.toBytes("CS"));
     put.add(Bytes.toBytes("dept-info"), Bytes.toBytes("d-location"),
Bytes.toBytes("FAB"));

     /* Put the data into the table */
     table.put(put);

     /* Flush the commits to ensure the local buffer is submitted
successfully */
     table.flushCommits();

     /* close table */
     table.close();
```

Retrieve data from table:

```
     /* Create an object to interact with a HBase table */
     HTable table = new HTable(conf, "school");

     /* Create an object (tool) to retrieve data, need argument to
specify key for the new row.
        In this example, our key(unique) would be "PSU" and "CS".*/
     Get get = new Get(Bytes.toBytes("PSU-CS"));

     /* Describe column family that we want to get */
     get.addFamily(Bytes.toBytes("school-info"));
```

```
      get.addFamily(Bytes.toBytes("dept-info"));

      /* Specify how many versions we want to achieve, (1 would be the
last version) */
      get.setMaxVersions(1);

      /* Return the result from get */
      Result result = table.get(get);
```

Update data from table:

```
      /* To do update, we can also use "Put".
         Create an object (tool) to insert data, need argument to
specify key for the new row.
         In this example, our key(unique) would be "PSU" and "CS".*/
      Put put = new Put(Bytes.toBytes("PSU-CS"));

      /* Add values by the following format: add(<column-family>,
<column>, <data>
         Value are all byte arrays.
         The same as in create section, we overwrite PSU to OSU.
         Since we create "Put" Object by unique key, we know which row
      we are manipulating.*/
      put.add(Bytes.toBytes("school-info"), Bytes.toBytes("s-name"),
Bytes.toBytes("OSU"));

      /* Put the data into the table */
      table.put(put);

      /* Flush the commits to ensure the local buffer is submitted
successfully */
      table.flushCommits();

      /* close table */
      table.close();
```

Delete data from table:

```
      /* To do update, we can also use "Delete".
         Create an object (tool) to delete data, need argument to
specify key for the new row.
```

```
        In this example, our key(unique) would be "PSU" and "CS".*/
     Delete delete = new Delete(Bytes.toBytes("PSU-CS"));

     /* Two kinds of "Delete" method in HBase, delete column-families
or delete columns.
        In here, we give an example about delete a column-families
        In HBase, we can just delete a specific column-families in a
row instead of delete
        the whole row.*/
     delete.delete(Bytes.toBytes("school-info"));

     /* Put the data into the table */
     table.delete(delete);
     /* close table */
     table.close();
```

2 Query Support

Languages and Queries

- HBase does not have any query syntax but providing query-like capabilities with the HBase filter command, which provides similar functionality as SQL where clause. We will be able to use custom filters to return a set of subset of the scan results. For specific, HBase provide compound operators system that group clauses together in order to provide query-like capabilities.

- HBase does not support range queries syntax (since it does not support query syntax). Instead, HBase can do range queries over key values, via several mechanisms, such as RowFilter, or STARTROW and ENDROW on a scan. For example, It has RowFilter that compares each row key with the comparator using the compare operator and if the comparison returns true, it returns all the key-values in the row.

- HBase does not support query syntax. Unfortunately, HBase do not have any similar field that can provide the same functionality and implement directly by HBase. Instead, we have to use graph framework like Apache TinkerPop to build graph database build on top of HBase in order to visualize data. By using API, we will be able to

storing, querying the graph data represented by edge and vertices. The basic mechanism is that the graph database that implemented by TinkerPop take those unique key (row ids) as the combination of VertexID and EdgeID.

Automatic Query Optimization

- HBase does not support query syntax. Instead, we can optimize scans of HBase tables by modifying some properties such as caching, region size tuning. For example, we can optimize scanner.caching, which is set in file hbase-site.xml and is the number of rows that are fetched when calling next on a scanner if it is not served from local memory. While we have an unstructured data and our query looks for wide range rowkey lookups, high caching values enable faster scanners by using more memory.

3 Indexes

Indices Support

- Hbase uses Hfile to store tables on HDFS. Hfile has a block-indexed file format, which means data is stored in a sequence of blocks. An index is stored at the end of each block. When a read request occurs, the index is read for the block location, and then the requested data is read from the block.
- Row keys act as primary index, but indexing is not present in HBase in default. However, we can define codes or script to perform indexing in HBase.

Primary and Secondary Indices

- Row keys are the primary indices.
- Secondary indices are not built-in in Hbase, but they can be created by using “Coprocessor” strategy, which will require additional cluster space and updating on the processing cycle. This can also be overcome by writing MapReduce code, integrating with Apache SOLR and with Apache Phoenix.

Limitations

- An index cannot be created on an external table, custom encoded columns and columns that are the leading part or parts of a row key.

Management

- Since indices are not built-in in HBase, we have to implement and manage the indices manually.

4 Storage

By default, Hbase stores data in HDFS. HDFS is not modifiable, but we can append new data to HDFS. It is also possible to run Hbase on other distributed file system, such as Amazon s3, GFS. Hbase has “region servers” that maintains in-memory data. In-memory data is flushed to disk periodically.

5 Transactions and Concurrency Control

Consistency

Hbase is not an ACID database, but they do have certain properties that are derived from such a database that offer some guarantees of consistency, such as:

- Atomic mutation of data rows
- Evolving state of the data.
    - When a mutation happens on a row of data it will always change holistically to a new version. You will never have rows that revert to an old state.
- No interleaving of data when mutating rows (see number 2 below)

Consistency Guarantees

1) Any changes made to a row of data (insert, update) are atomic. If you do an operation on multiple rows at once, these changes are not atomic. You have the possibility of some succeeding and some failing. The outcome of the operation you make on a given row from your set will be atomic, but not all guaranteed to succeed or fail as a whole.

2) Any changes made to a row of data happen in a well defined order. This meaning that you will have intermingled results committed from writes against columns in different families. You will not have mutations of the same row by two different clients, across different column families, be mixed and committed as the final state of the row. For example, one user sets a=1 b=1 c=1 and user two sets a=2 b=2 c=2. You will never have a=1 b=2 c=1 (or any other combination). You will always have one or the other. One caveat to this is multi-row batch mutations, as backed up by part 1 above, since atomicy is not guaranteed in multi-row
3) You are always guaranteed to receive a full row of data from a read, as it exists in some point in history. You may not receive the most current row, but you will never receive intermingled results from different versions of that row.
4) A row of data will only move forward in time, with a newer version. You will never have a row version revert.
5) Scans of data are only guaranteed to return the state of the data as committed up to the time of the start of the scan. No partial mutations will exist in the returned data.
6) Any version of a cell that has been returned to a read operation is guaranteed to be durably stored

Consistency Implementation

HBase uses its own version of MVCC to control consistency across reads and writes. We can see the high level steps of the process when you do certain operations to see how MVCC is used.

Here are a few definitions that will help:
WriteNumber: the version number that will be used for the next write of a given row
ReadPoint: The current row version number that will be used for a read of that row

When a write is attempted:

1) lock the row(s), to guard against concurrent writes to the same row(s)
2) retrieve the current writenumber (version number for write)
3) apply changes to the WAL (Write Ahead Log)
4) apply the changes to the Memstore (using the acquired writenumber to tag the KeyValues)
5) commit the transaction, i.e. attempt to roll the Readpoint (version for reads) forward to the acquired Writenumber.

6) unlock the row(s)

When a read is attempted:

1) open the scanner
2) get the current readpoint
3) remove all scanned KeyValues with memstore timestamp > the readpoint, this gets rid of writes that take place after the start of the scan.
4) close the scanner (this is initiated by the client)

6 Scalability and Replication

Replication

- For replication. HBase needs to use Zookeeper which is already built-in to the HBase distribution. Zookeeper is a coordination service that HBase uses and the service provides distributed synchronization which is important for data replication across several clusters.
- The amount of clusters HBase should use for its implementation should be an odd number greater than one. The source cluster is the only cluster responsible for writing new data, the destination clusters only receives new data from the source clusters.
- HBase uses asynchronous replication and maintains a goal of consistency.
- Table names must be the same on source and destination clusters. Tables can be specified on the source cluster to be replicated on the destination without updating others.

Scalability

- HBase can handle petabytes of data. Large tables are distributed dynamically when they get to be too big, HBase is capable of autosharding, the sharding key is the row key.
- HBase has a set of regions, which is just a subset of the table data that is contiguous with the rows sorted together. One region is served by one and only one regional server, but a regional server can serve one to many regions for the purposes of load balancing across the system.

7 Platform/Deployment

a) Hbase is usually deployed in fully distributed mode in a onsite datacenter or in Cloud (single and multi). The Cloud infrastructure enables us to deploy Hbase without the need to have hardware or specific setup. Amazon EMR, Google Cloud

Platform, Microsoft Azure, ,CenturyLink Cloud IBM and Oracle are some of the cloud infrastructure that support Hbase.

b)
Standalone Mode: Hadoop is configured to run in a non-distributed or standalone mode, as a single Java process. There are no daemons running and everything runs in a single JVM instance & HDFS is not used. This supported by single multi core CPU
Pseudo-distributed mode: The Hadoop daemons run on a local machine, simulating a cluster on a small scale. Different Hadoop daemons run in different JVM instances, but on a single machine & HDFS is used instead of local File System. This is supported by single multi core CPU.
Fully-Distributed Mode: In the fully-distributed mode, all daemons are executed in separate nodes forming a multi-node cluster. This is supported by cloud infrastructure and Multicore CPUs running on several machines each machine running as a client or server.

c) To support language bindings and communication protocols Hbase uses Thrift. Thrift supports remote procedure call (RPC) framework and scalable cross-language services development. It supports languages such as Java, C, C++, C#, Cappuccino, Cocoa, Delphi, Erlang, Go, Haskell, Java, Node.js, Objective-C, OCaml, Perl, PHP, Python, Ruby and Smalltalk.

Refernce : http://hbase.apache.org/0.94/book/standalone_dist.html