**ASSIGNMENT 32.1**

**Explain in brief**

**● What is the difference between memstore and hfile in HBase?**

**Memstore**

MemStore: is the write cache. It stores new data which has not yet been written to disk. It is sorted before writing to disk

There is one MemStore per CF.

Memstore is kept in RS main memory

**Hfile**

Hfiles store the rows as sorted KeyValues on disk

HBase uses multiple HFiles per column family

HFiles are written to HDFS

* **Describe compactions in HBase.**

Apache Hbase is an distributed data store optimized for read performance and the optimized read performance come from having only one file per Column Family.Hbase idea comes from Google File system and based on log structures merge tree.

One file per Column Family is not always possible during period of heavy writes, that is the reason Hbase try to combine the HFiles to reduce the maximum number of disk seeks needed for a read. This process is called compaction.

Compaction, the process by which HBase cleans up after itself, comes in two flavors: major and minor

**Minor compactions**: combine a configurable number of smaller HFiles into one larger HFile. You can tune the number of HFiles to compact and the frequency of a minor compaction. Minor compactions are important because without them, reading a particular row can require many disk reads and cause slow overall performance.

**Major Compactions:** reads all the Store files for a Region and writes to a single Store file.

* **List and explain the logical entities in HBase**

1.Tables

2.row

3.Column Family

4.Column Qualifier

5.cell

5.Versiom

**Tables** – The HBase Tables are more like logical collection of rows stored in separate partitions called Regions. As shown above, every Region is then served by exactly one Region Server. The figure above shows a representation of a Table.

**Rows** – A row is one instance of data in a table and is identified by a rowkey. Rowkeys are unique in a Table and are always treated as a byte[].

**Column Families** – Data in a row are grouped together as Column Families. Each Column Family has one more Columns and these Columns in a family are stored together in a low level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it’s important that proper care be taken when designing Column Families in table. The table above shows Customer and Sales Column Families. The Customer Column Family is made up 2 columns – Name and City, whereas the Sales Column Families is made up to 2 columns – Product and Amount.

**Columns** – A Column Family is made of one or more columns. A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon – example: columnfamily:columnname. There can be multiple Columns within a Column Family and Rows within a table can have varied number of Columns.

**Cell** – A Cell stores data and is essentially a unique combination of rowkey, Column Family and the Column (Column Qualifier). The data stored in a Cell is called its value and the data type is always treated as byte[].

**Version** – The data stored in a cell is versioned and versions of data are identified by the timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3.

* **What will happen if we do not create a row key while inserting the data?**

Actually it is not possible to insert data without rowkey because the Hbase stores data as a key value pair where Key is row KEY which is a unique identifier for each row.

* **How can filters be applied in HBase and what are the benefits?**

When reading data from HBase using Get or Scan operations, you can use custom filters to return a subset of results to the client. While this does not reduce server-side IO, it does reduce network bandwidth and reduces the amount of data the client needs to process. Filters are generally used using the Java API, but can be used from HBase Shell for testing and debugging purposes.

Different types of filters applied in hbase are

**KeyOnlyFilter**

It gives only key of keyvalue pair. No arguments needed.

**FirstKeyOnlyFilter**

This filter doesntt take any arguments. It returns only the first key-value from each row.

**PrefixFilter**

This filter takes one argument a prefix of a row key. It returns only those key-values present in a row that starts with the specified row prefix

**ColumnPrefixFilter**

This returns all the key value pairs of the columns with specific prefix mentioned in the argumernt. It takes only one argument.

**MultipleColumnPrefixFilter**

Similar to prefix filter but it takes multiple arguments for prefixes and returns all the rows that which are matching with any of the prefix arguments.

**ColumnCountGetFilte**r

This filter takes one argument a limit. It returns the first limit number of columns in the table.

**PageFilter**

This filter takes one argument a page size. It returns page size number of rows from the table.

**InclusiveStopFilter**

It returns all key-values present in rows up to and including the specified row. The specified row is mentioned in the argument.

**Family Filter**

It compares using the operartor. It compares the whole data with given column family for the given operator if the value is true then it will return that key value.

**Value Filter**

This is similar to family filter except here it compares the value and not the column family.

**SingleColumnValueFilter**

This filter takes a column family, a qualifier, a compare operator and a comparator. If the specified column is not found – all the columns of that row will be emitted. If the column is found and the comparison with the comparator returns true, all the columns of the row will be emitted. If the condition fails, the row will not be emitted

* **What are the data model operations in hBase?**

The four primary data model operations are Get, Put, Scan, and Delete. Operations are applied via HTable instances.

* **Get**

Get returns attributes for a specified row. Gets are executed via HTable.get.

* **Put**

Put either adds new rows to a table (if the key is new) or can update existing rows (if the key already exists). Puts are executed via HTable.put (writeBuffer) or HTable.batch (non-writeBuffer).

* **Scans**

Scan allow iteration over multiple rows for specified attributes.

The following is an example of a on an HTable table instance. Assume that a table is populated with rows with keys "row1", "row2", "row3", and then another set of rows with the keys "abc1", "abc2", and "abc3". The following example shows how startRow and stopRow can be applied to a Scan instance to return the rows beginning with "row".

HTable htable = ... // instantiate HTable

Scan scan = new Scan();

scan.addColumn(Bytes.toBytes("cf"),Bytes.toBytes("attr"));

scan.setStartRow( Bytes.toBytes("row")); // start key is inclusive

scan.setStopRow( Bytes.toBytes("row" + (char)0)); // stop key is exclusive

ResultScanner rs = htable.getScanner(scan);

try {

for (Result r = rs.next(); r != null; r = rs.next()) {

// process result...

} finally {

rs.close(); // always close the ResultScanner!

}

* **Delete**

Delete removes a row from a table. Deletes are executed via HTable.delete.

HBase does not modify data in place, and so deletes are handled by creating new markers called tombstones. These tombstones, along with the dead values, are cleaned up on major compactions.

* **How can MapReduce be used with HBase?**

Apache MapReduce is a software framework used to analyze large amounts of data, and is the framework used most often with Apache Hadoop. HBase can be used as a data source, TableInputFormat, and data sink, TableOutputFormat or MultiTableOutputFormat, for MapReduce jobs. MapReduce jobs using Hbase can be done by subclass TableMapper and/or TableReducer

Example for reducer class that extends tablereducer

public static class Reducer1 extends TableReducer<ImmutableBytesWritable, IntWritable, ImmutableBytesWritable> {

public void reduce(ImmutableBytesWritable key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

Put put = new Put(key.get());

put.add(Bytes.toBytes("details"), Bytes.toBytes("total"), Bytes.toBytes(sum));

System.out.println(String.format("stats : key : %d, count : %d", Bytes.toInt(key.get()), sum));

context.write(key, put);

}

}

* **What is regionserver?**

Regions are nothing but tables that are split up and spread across the region servers.

Region server

The region servers have regions that -

* Communicate with the client and handle data-related operations.
* Handle read and write requests for all the regions under it.
* Decide the size of the region by following the region size thresholds.

When we take a deeper look into the region server, it contain regions and stores as shown below:



The store contains memory store and HFiles. Memstore is just like a cache memory. Anything that is entered into the HBase is stored here initially. Later, the data is transferred and saved in Hfiles as blocks and the memstore is flushed.