**ASSIGNMENT 22.3**

* **Sequence File Format**
* **NLine Input Format**
* **DB Input Format**
* **DB Output Format**

**Sequence File Format:**

[SequenceFile](http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/io/SequenceFile.html) is a flat file consisting of binary key/value pairs.

It is extensively used in [MapReduce](https://wiki.apache.org/hadoop/MapReduce) as input/output formats.

It is also worth noting that, internally, the temporary outputs of maps are stored using SequenceFile.

The SequenceFile provides a Writer, Reader and Sorter classes for writing, reading and sorting respectively.

There are 3 different SequenceFile formats:

1. Uncompressed key/value records.
2. Record compressed key/value records - only 'values' are compressed here.
3. Block compressed key/value records - both keys and values are collected in 'blocks' separately and compressed.

* Sequence file format is one of the hadoop specific file format and stores the serialized data hence the transferring of the file becomes very easy if stored in the sequence file format.
* Hadoop does not work very well with a lot of small files hence in order to solve this the sequence files are used as the container to store the small files and convert them into one large file.
* Sequence files are flat files containing key, value pairs.
* Every Sequence file has got a header which describes the about the file.
* The header contains information such as key/value class names, version, file format, metadata about the file and sync marker to denote the end of the header.

**Advantages:**

1. A smaller number of memory needed on NameNode. Continue with the 10,000 100KB files example,
   * Before using SequenceFile, 10,000 objects occupy about 4.5MB of RAM in NameNode.
   * After using SequenceFile, 1GB SequenceFile with 8 HDFS blocks, these objects occupy about 3.6KB of RAM in NameNode.
2. SequenceFile is splittable, so is suitable for MapReduce.
3. SequenceFile is compression supported.

**NLineInputFormat:**

* NLineInputFormat in Hadoop is another form of TextInputFormat where the keys are byte offset of the line and values are contents of the line.
* Each mapper receives a variable number of lines of input with TextInputFormat and KeyValueTextInputFormat and the number depends on the size of the split and the length of the lines.
* And if we want our mapper to receive a fixed number of lines of input, then we use NLineInputFormat.
* N is the number of lines of input that each mapper receives. By default (N=1), each mapper receives exactly one line of input.
* If N=2, then each split contains two lines. One mapper will receive the first two Key-Value pairs and another mapper will receive the second two key-value pairs.

**Changes to be made in driver:**

job.setInputFormatClass(NLineInputFormat.class);

NLineInputFormat.addInputPath(job, inputPath);

job.getConfiguration().setInt("mapreduce.input.lineinputformat.linespermap", 1);

**DBInputFormat:**

* DBInputFormat in Hadoop is an Input Format that reads data from a relational database, using JDBC.
* As it doesn’t have portioning capabilities, so we need to careful not to swamp the database from which we are reading to many mappers.
* So it is best used for loading relatively small datasets, perhaps for joining with large datasets from [HDFS](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) using MultipleInputs.
* Here Key is LongWritables while Value is DBWritable

**These are the changes that are to be made in driver to use db input format**

DBConfiguration.configureDB(<Configuration>,<Driver>,<Connection String>,<UserName>,<Password>);

job.setInputFormatClass(DBInputFormat.class);

DBOutputFormat.setOutput(<Job>,<Output Table>,<List of table columns>);

**Changes in mapper code**

public class DBInputFormatMapextends Mapper<LongWritable, DBInputWritable, Text, NullWritable>

protected void map(

LongWritable id,

DBInputWritable value, Context context)

**Implementation of the db input format.**

public void

readFields(ResultSet rs) throws SQLException

{

id = rs.getInt(1);

name = rs.getString(2);

location = rs.getString(3);

**}**

**Similar modifications have to be done in db output format**

**Driver code**

DBConfiguration.configureDB(<Configuration>,<Driver>,<Connection String>,<UserName>,<Password>);

job.setInputFormatClass(DBInputFormat.class);

DBOutputFormat.setOutput(<Job>,<Output Table>,<List of table columns>);

**Mapper code**

public class DBOutputFormatMap

extends Mapper<LongWritable, Text, DBOutputWritable, NullWritable>protected void map(LongWritableid, Text value, Context context)

context.write(new DBOutputWritable(Integer.parseInt(lineArray[1]), lineArray[2], lineArray[3]), NullWritable.get())

**Implementation of db output format**

public void write(PreparedStatementps) throws SQLException

{

ps.setInt(1, id);

ps.setString(2, name);

ps.setString(3, location);

}