Data Serialization – Avro & Sequence files in BigData

# Scope

The scope of this project is to demonstrate the data serialization features of Avro and Sequence files and their role in Big Data/ Hadoop environment.

The project focuses on learning a new data serialization format AVRO and to work with sequence files API’s.

# Data Serialization/deserialization in Hadoop

Hadoop‘s native library provides Writables for data serialization (converting object data into byte stream) and deserialization (converting byte stream data to object data) and also it provides support for Sequence Files which will store the data in binary format. These are the only two mechanisms provided by hadoop for data serialization.

The main drawback of these two mechanisms is that Writables and SequenceFiles have only a Java API and they cannot be written or read in any other language. So any of the files created in hadoop with above two mechanisms can’t be read by any other third language which makes hadoop as a limited box. To address this drawback, Doug cutting created Avro, which is a language independent data structure.

# AVRO

## What is AVRO?

AVRO is a data format created by Apache to support the Hadoop eco system. Other data formats are XML, JSON, Protocol Buffers (by Google), and Thrift etc.

Avro stores data by putting data definition with the data allowing for Avro files to be read and interpreted by many different programs. It stores all of the data in a binary format making the files more compact, and will even add in markers to help Map Reduce jobs find where to break large files for more efficient processing.

## Why AVRO?

As a big data platform, Hadoop is designed to work with very large data sets, and it does so while following a divide and conquer approach towards data and its processing. In an ideal scenario, Hadoop prefers the data be divided into fewer but larger chunks that allow them to be processed efficiently with minimal overhead. Moreover, much of this data is semi-structured (for example: log files and sensor data) or unstructured (such as text from social or news service) and the ability to represent these efficiently and effectively (with meta-data for example) has proven to be quite helpful with data processing.

These imperatives have led to the creation of a variety of file formats for Hadoop, including the Avro storage format.

Avro Serialization Features:

* Avro is a language neutral data serialization system and it can be processed by many languages (currently C, C++, C#, Java, Python, and Ruby).
* Avro creates binary structured format that is both compressible and splittable, so, it can be efficiently used as the input to hadoop MapReduce jobs.
* Avro provides rich data structures, for example, we can create a record that contains an array, an enumerated type, and a sub record. These can be created in any language and can be processed in hadoop and the results can be fed to a third language.
* Avro schemas are defined in JSON. This facilitates implementation in languages that already have JSON libraries.
* In an Avro data file along with avro data, even schema is stored in a metadata section, and it makes the file self-describing.
* Avro is also used in RPC (Remote Procedure Call) and in this, the client and server exchange schemas in the connection handshake.
* Avro does not require that code be generated. Data is always accompanied by a schema that permits full processing of that data without code generation.

Comparison with Other Cross-Language Serialization Frameworks:

There are other serialization frameworks which provide language independent serialization mechanism. They are Protocol buffers (by google) and Thrift (by Apache).

These languages require code to be generated (for schema) to read or write the data files. However this is optional in Avro.

Schema is not stored with data in Thrift or Protocol Buffers but in Avro, since the schema is present when data is read, considerably less type information is need to be encoded with data.

Avro has rich schema resolution capabilities. The schema used to read data need not be identical to the schema that was used to write the data.

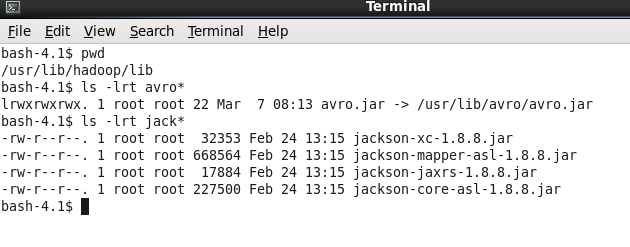
For example, a new, optional field may be added to a record by declaring it in the schema used to read the old data. New and old clients alike will be able to read the old data, while new clients can write new data that uses the new field. Conversely, if an old client sees newly encoded data, it will gracefully ignore the new field and carry on processing as it would have done with old data.

## Working with AVRO

Download the required Avro jar files from <http://avro.apache.org/releases.html>

AVRO requires JSON jars as well which can be downloaded from <http://www.json.org/>

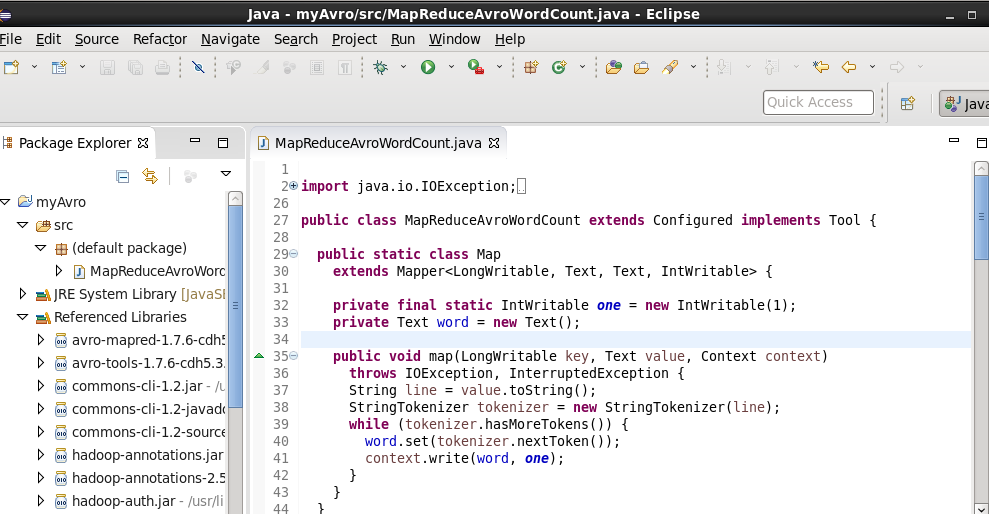
However as I was using the cloudera quickstart vm the required jar files for both avro and apache were already available under /usr/lib/hadoop/lib



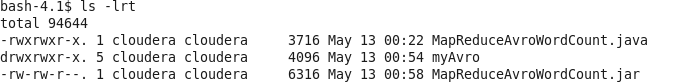
The use-case which I have used is to do the classic word count example using Avro schemas.

I have used the word count example to create a mapreduce job which creates a sample avro data file in hadoop distributed file system.

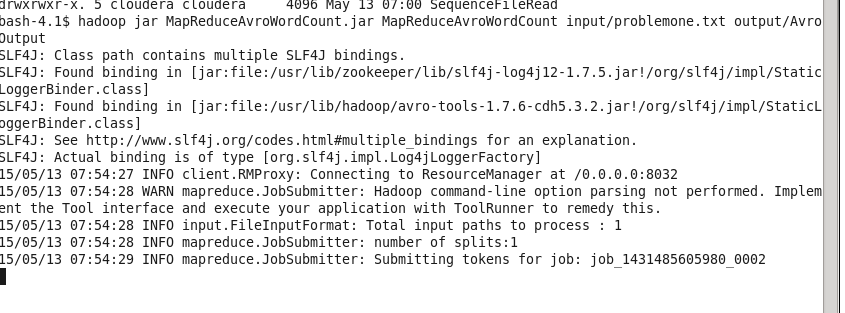
Using Eclipse IDE I have created a project and exported a jar file using the above code.

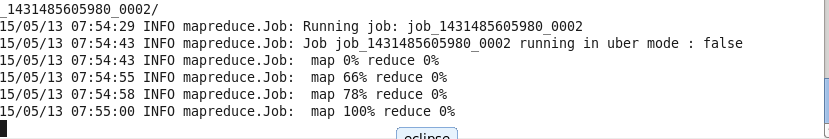


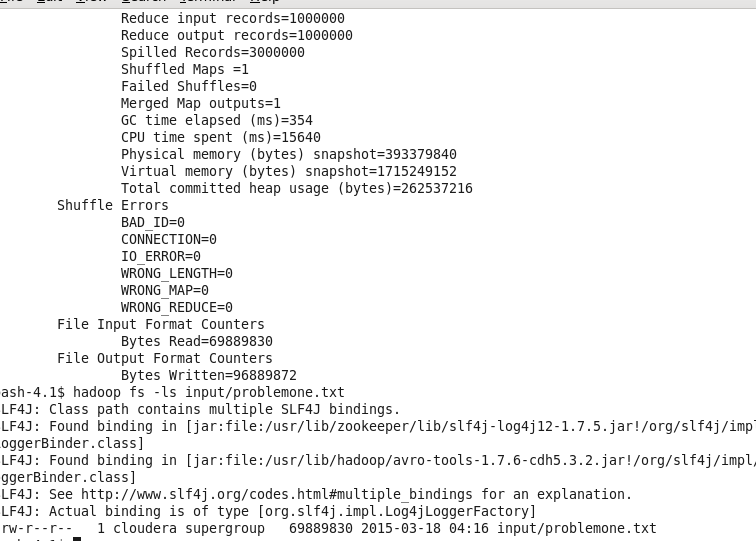
The jar file has been created as shown below:



Executing the above jar file

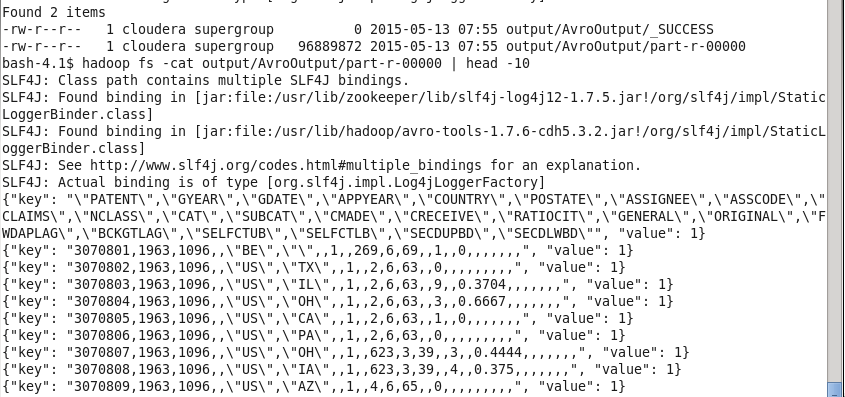




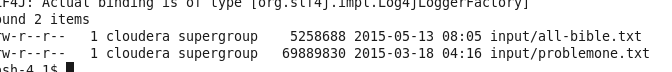


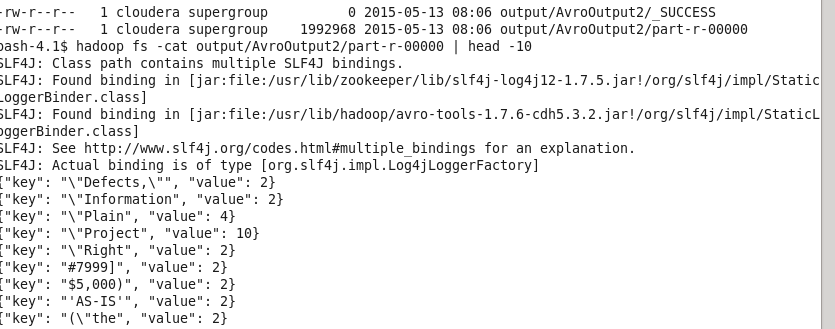
The input file is of size 690MB.

Following is the output



Repeating with the all-bible.txt file which is around 5MB





In the above code we are using AvroWrapper class to write pairs of <String, Integer> values and this pair is included in the reduce output key. Reducer’s output value is maintained as NullWritable as there is no need for it because, both string and its count are included in reducer’s output key part itself.

## Sequence Files

Advantages:

* As binary files, these are more compact than text files
* Provides optional support for compression at different levels – record, block.
* Files can be split and processed in parallel
* As HDFS and MapReduce are optimized for large files, Sequence Files can be used as containers for large number of small files thus solving hadoop’s drawback of processing huge number of small files.
* Extensively used in MapReduce jobs as input and output formats. Internally, the temporary outputs of maps are also stored using Sequence File format.

Limitations:

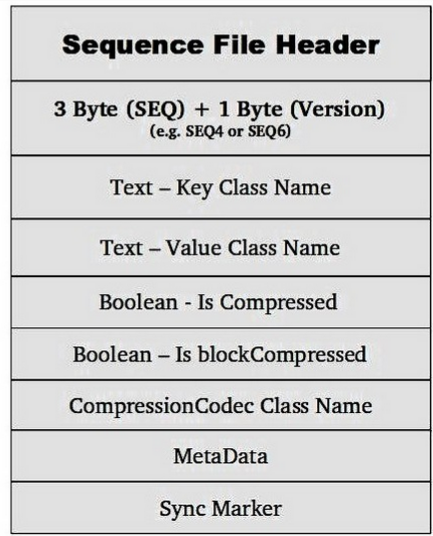
* Similar to other Hadoop files, SequenceFiles are append only.
* As these are specific to hadoop, as of now, there is only Java API available to interact with sequence files. Multi Language support is not yet provided.

Hadoop Sequence File Format:

Hadoop SequenceFile is a flat file consisting of binary key/value pairs. Based on compression type, there are 3 different SequenceFile formats:

* Uncompressed format
* Record Compressed format
* Block-Compressed format

A sequence file consists of a header followed by one or more records. All the above three formats uses the same header structure



SequenceFiles Java API:

Apache Hadoop provides various classes to create/read/sort SequenceFiles and below are some of the important classes useful in dealing with hadoop sequence files.

*SequenceFile – org.apache.hadoop.io.SequenceFile*

This is the main class to write/create sequence files and read sequence files. It provides SequenceFile.Writer, SequenceFile.Reader and SequenceFile.Sorter classes for writing, reading and sorting respectively.

For compressed sequence file creations there are special classes

*SequenceFile.RecordCompressWriter* & *SequenceFile.BlockCompressWriter*

But to create an instance of any of the above writer class flavors, we use one of the static methods createWriter(). There are several overloaded versions of it but they all require a minimum. Configuration object and varargs Writer.Option… object to specify the options to create the file with.

In the varargs Writer.Option, we need to specify at least file name, file system, key and value classes parameters to create the sequence file. Compression type, codec, write progress, and a Metadata instance to be stored in the SequenceFile header can be provided optionally.

Once we have a SequenceFile.Writer instance, then we can write key-value pairs, using the append() method. After finishing of writing, we need to call the close() method.

Similar to writer instance, SequenceFile.Reader instance is used to read the sequence files and it can read any of the SequenceFile formats created with above Writer instance.

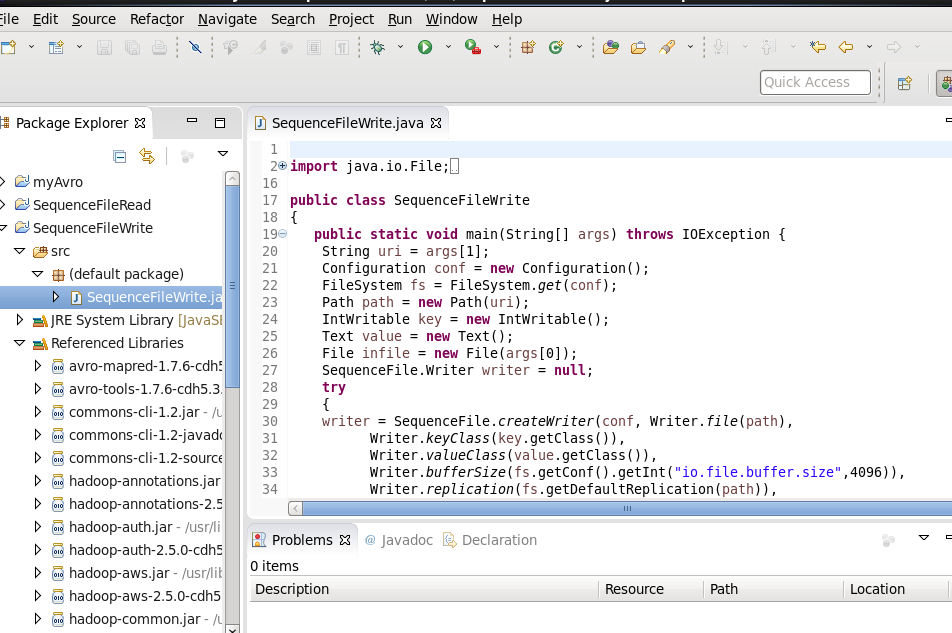
Instance of SequenceFile.Reader class can be created with one of its constructor methods.

Once we have Reader instance we can iterate over all records by repeatedly invoking one of its next() methods.

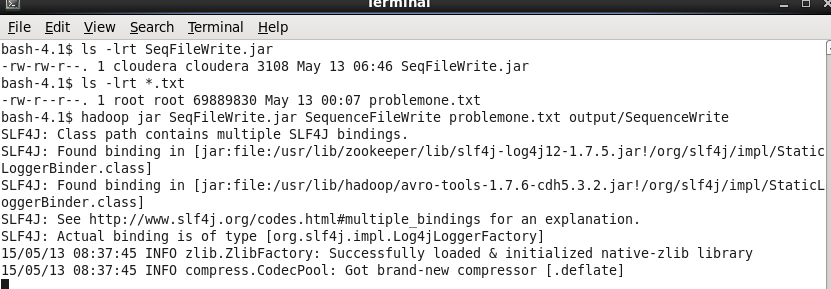
In the below example program, we are reading contents from a text file (problemone.txt) on local file system and writing it to sequence file on hadoop. Here, we are using integer counter as key and each line from input file as value in sequence file format’s <key, value>.

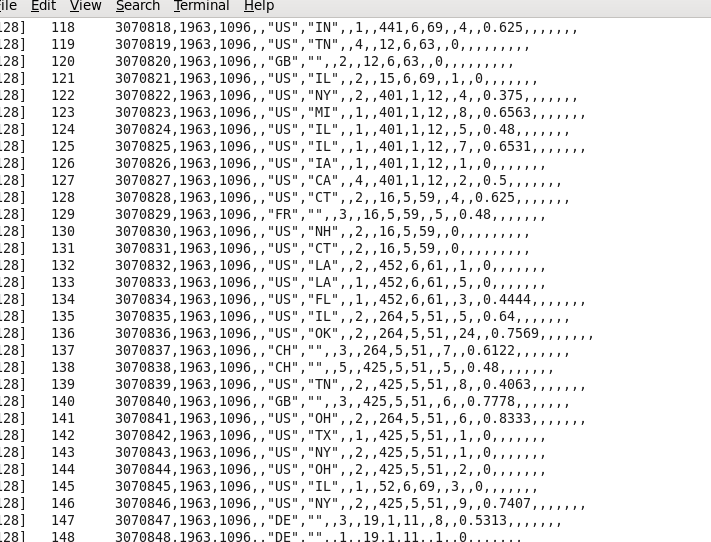
For verification of (key, value) pairs in sequence file, we are printing first 50 records onto console.

Jar file is created in eclipse

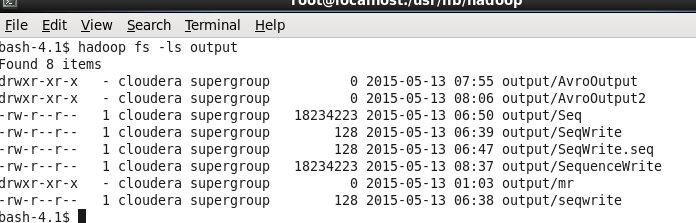


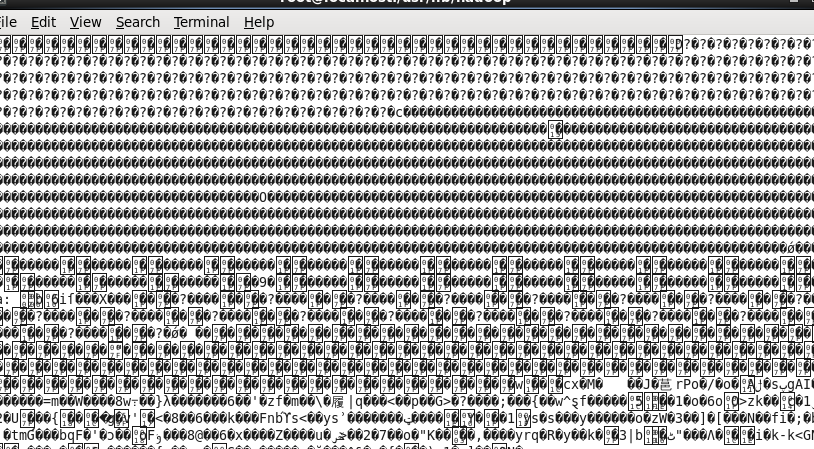
Executing the jar file on Hadoop





Checking the output and this should be a binary file



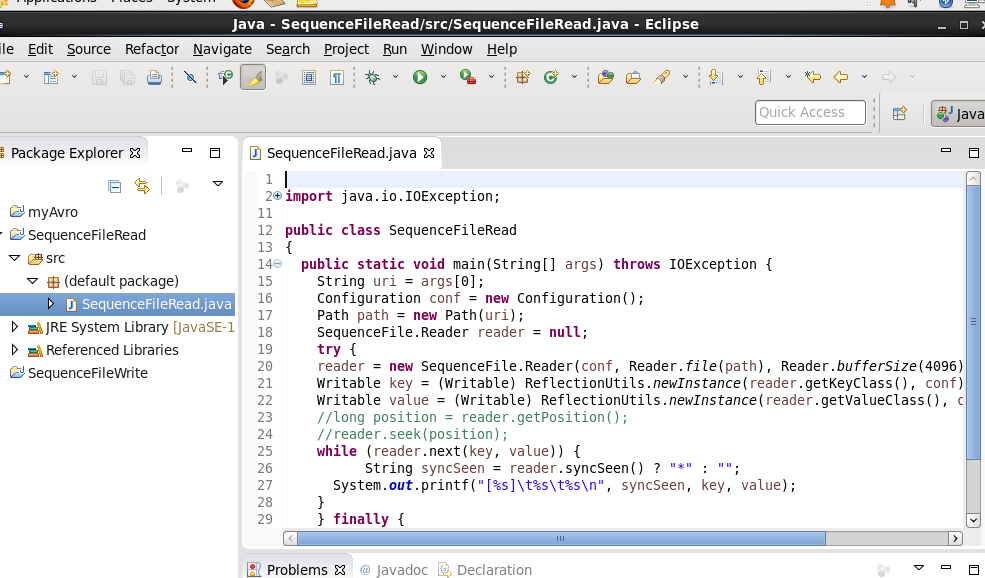


Reading SequenceFile Example:

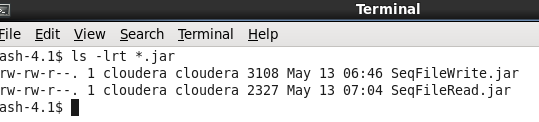
Now we will see how to read the above created sequence file through Hadoop API. We will create SequenceFile.Reader instance and use next(key, value) method to iterate over each record in the sequence file.

We didn’t mention compression type or codec to the sequence file that we used while creating it. By default reader instance will get these details from the file format itself and decompresses the file according to the codec found in the file format. Also note that, we have used getKeyClass() and getValueClass() methods on reader instance to retrieve the class names of (key,value) pairs in sequence file.

In the below program we are reading the contents of sequence file and printing them on console.

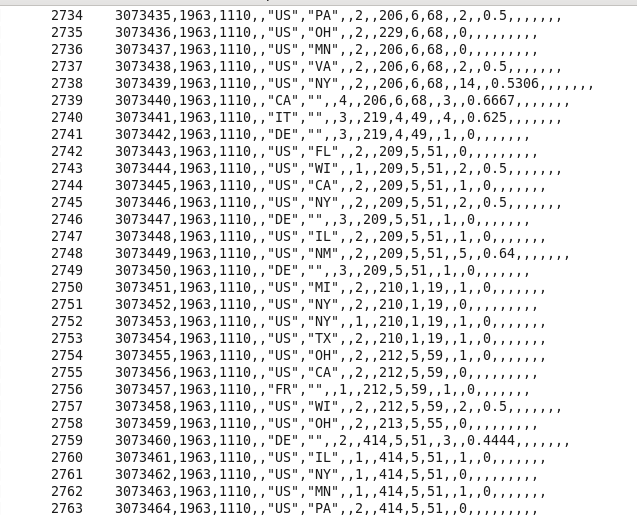


Executing the jar file



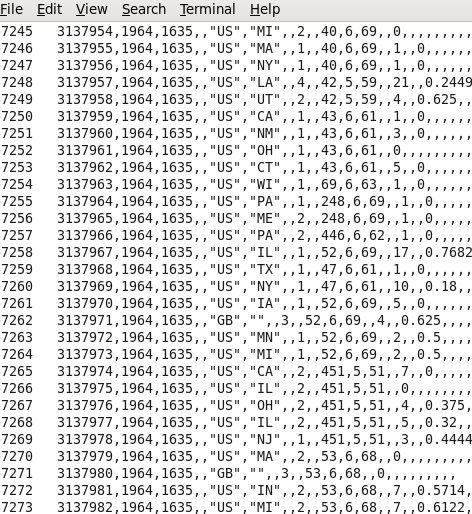


The output is stdout onto screen



The sequence file can be read directly from Hadoop fs itself using





# Conclusion

Using either Avro or Sequence Files reduces the time and space which are two crucial elements in any distributed/parallel processing environment. Being close to the binary data structure sequence files are faster but are very specific to the Hadoop environment thus Avro has an advantage over Sequence files due to its language independence.

The SequenceFile container and each Writable  implementation stored in it are only implemented in Java.  There is no  format specification independent of the Java implementation.  Avro data  files have a language-independent specification and are currently  implemented in C, Java, Ruby, Python, and PHP.  A Python-based  application can directly read and write Avro data files.

Language independence can be an advantage today however if you'd like to create or access data outside of MapReduce programs from non-Java applications.  Moreover, as the data platform expands, we'd like to be able to include more non-Java applications and to easily interchange data with these applications.

If a Writable class changes, if fields are added or removed, the type of a field is changed or the class is renamed, then data is usually unreadable.  A Writable implementation can explicitly manage versioning, writing a version number with each instance and handling older versions at read-time.  This is rare, but even then, it does not permit forward-compatibility (old code reading a newer version) nor branched versions.  Avro automatically handles field addition and removal, forward and backward compatibility, branched versioning, and renaming, all largely without any awareness by an application.