CS 523 – BDT
Big Data Technology

Lesson 5

Apache Avro

Enjoy greater efficiency and accomplish more

Maharishi International
University

WHOLENESS OF THE LESSON

Apache Avro is Hadoop's preferred cross language data serialization system which serializes data in a format that has schema built-in. The serialized data is in a compact binary format that doesn't require proxy objects or code generation.

Science & Technology of Consciousness: Growth occurs through the process of transcending — transcending to deeper levels and then coming out.

Serialization & Deserialization

- ***Serialization*** - process of **converting object data** into **byte stream data** for transmission over a network across different nodes in a cluster or for persistent data storage.
- ***Deserialization -*** reverse process of turning a byte stream back into a series of structured objects.
- **Use of Serialization/Deserialization**
  - For interprocess communication and for persistent storage.
- In Hadoop, IPC between nodes in the system is implemented using *remote procedure calls* (RPCs) which uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message.

Desired Characteristics of an RPC Serialization Format

- ***Compact –*** making the best use of n/w bandwidth
- ***Fast –*** essential that there is as little performance overhead as possible for the serialization and deserialization process.
- ***Extensible -*** should be straightforward to evolve the protocol in a controlled manner for clients and servers.
  - For example, it should be possible to add a new argument to a method call and have the new servers accept messages in the old format (without the new argument) from old clients.
- ***Interoperable -*** to be able to support clients that are written in different languages to the server

Java Serialization

- Java Serialization is a general-purpose mechanism for serializing graphs of objects, so it has some overhead for serialization and deserialization operations.
- In Java Serialization process, it writes meta data (class name, field names and types and its super class) about each object being written to the stream! - **NOT very compact!**
- Subsequent instances of the same class write a reference handle to the first occurrence. That means, there is **state stored in the stream**.
- Though these reference handles occupy only 5 bytes, **they don't work well with random access**, since the referent class may occur at any point in the preceding stream.
- That's why it does not work well when it comes to sorting records in a serialized stream, since the first record of a particular class is distinguished and must be treated as a special case.

Java Serialization contd…

- **Deserialization procedure creates a new instance for each object** deserialized from the stream!
- Because of all these problems Java Serialization doesn't meet the criteria for a serialization format listed earlier: compact, fast, extensible, and interoperable.

Hadoop Serialization Needs

- In Big Data processing scenarios, we usually need to have a precise control over exactly how objects are written and read.
- With Java Serialization and even with RMI, you need to fight to get some control.
  - Effective, high-performance IPCs are critical to Hadoop. Need to precisely control how things like connections, timeouts and buffers are handled.
- **So, to achieve high performance serialization format, Hadoop uses its own Writables!**

Advantage of Hadoop Writables over Java Serialization

1. **Meta-info not stored**
   - In Java serialization, it writes meta data about the object to the stream.
   - In Hadoop Serialization, while defining '*Writable*', we (applications) know the expected class.
   - So **Writables do not store their type in the serialized representation**.
     - For example, while deserializing, if the input key is *LongWritable* instance then an empty *LongWritable* instance is asked to populate itself from the input data stream.
   - As no Meta info need to be stored, this results in more compact binary files, random access and high performance.

Advantage of Hadoop Writables over Java Serialization contd..

2. **No object creation overhead**
   - Hadoop's Writable-based serialization is capable to reduce the object-creation overhead by reusing the Writable objects, which is not possible with Java's serialization.
     - For a MapReduce job, which at its core serializes and deserializes billions of records of just a handful of different types, the savings gained by not having to allocate new objects are significant.

This thread is interesting!

Advantage of Hadoop Writables over Java Serialization

- Difference between serialization in Java and Hadoop.

```
smallInt serialized value using Java serializer
aced0005737200116a6176612e6c616e672e496e74656765
7212e2a0a4f781873802000149000576616c756578720106a6176612e
6c616e672e4e756d62657286ac951d0b94e08b020000787000000064
smallInt serialized value using IntWritable
00000064
```

- If the size of serialized data in Hadoop is like that of Java, then it will definitely become an overhead in the network.

Disadvantages of Hadoop Serialization

- To serialize Hadoop data, there are two ways:
  - Use the **Writable** classes, provided by Hadoop's native library.
  - Use **Sequence Files** which store the data in binary format.
- **Drawback**:- *Writables* and *SequenceFiles* have only a Java API and they cannot be written or read in any other language.
- Therefore, any of the files created in Hadoop with above two mechanisms cannot be read by any other third language!
- To address this drawback, Doug Cutting created Avro, which is a language independent data structure.
- Only Writables or Avro objects can be serialized or deserialized out of the box in Hadoop.

Apache Avro

- Apache Avro is a language-neutral data serialization system created by Doug Cutting to address the major downside of Hadoop Writables: lack of language portability.
- Avro provides:
  - Rich data structures
  - A compact, fast, binary data format
  - A container file, to store persistent data
  - Simple integration with dynamic languages
- Main uses of Apache Avro are for:
  - Data serialization/deserialization & Data exchange (RPC)

Apache Avro

- Avro currently supports languages such as Java, C, C++, C#, JavaScript, Perl, PHP, Python, and Ruby.
- Avro is also the best choice for Kafka.
- Avro data is described using a language-independent *schema*. Including schemas with Avro messages allows any application to deserialize the data.
- Serialized data is in a compact binary format that doesn't require proxy objects or code generation.
  - This means you can read and write data that conforms to a given schema even if your code has not seen that particular schema before.
- To achieve this, Avro assumes that the schema is always present - at both read and write time - which makes for a very compact encoding, since encoded values do not need to be tagged with a field identifier.

Avro Schema (.avsc)

- Avro data format (wire format and file format) is defined by Avro schemas.
- Avro accepts schemas as input. (.avsc files)
- A schema file can only contain a single schema definition.
- Schemas are written in JSON, and data is encoded using a binary format, but there are other options, too.
- Schemas are composed of primitive types (null, boolean, int, long, float, double, bytes, and string) and complex types (record, enum, array, map, union, and fixed).
- **Avro data plus schema is fully self-describing data format.**

Avro Schema

- Avro schema is written in JSON

```
// a simple three-element record
{“name”: “Employee”, “type”: “record”,
  “fields”: [
      {“name”: “id”, “type”: “int” },
      {“name”: “name”, “type”: “string” },
      {“name”: “dependents”, “type”:
          {“type”: “array”, “items”: “string”}},
  ]
}

// a linked list of strings or ints
{“name”: “MyList”, “type”: “record“,
“namespace”: “edu.mum.cs523.avro”
  “fields”:  [
      {“name”: “value”, “type”: [“string”, “int”]},
      {“name”: “next”, “type”: [“MyList”, “null”]}
  ]
}
```

- **namespace** – This field describes the name of the namespace in which the object resides.
- **name** – This field comes under the document as well as under the named fields.
  - In case of document, it describes the schema name. This schema name together with the namespace, uniquely identifies the schema within the store (**Namespace.SchemaName**).
  - In case of fields, it describes name of the field.
- **type** – This field comes under the document as well as under the named fields.
  - In case of document, it shows the type of the document, generally a *record* because there are multiple fields.
  - In case of fields, the type describes data type.

Avro Data Files (.avro)
(Avro Container Files)

- Avro *datafiles* are are compact, splittable and portable across different programming languages.
- Avro datafiles store only records (not key/value pairs).
- Avro *datafile* has a metadata section where the schema is stored, which makes the file self-describing.
  - Objects stored in Avro datafiles are described by this schema, rather than in the Java code of the implementation of a Writable object (as is the case for sequence files).
- The system can then generate types for different languages, which is good for interoperability.
  - For example, you can write a file in Python and read it in C.

Avro Data Files (.avro) contd..

- In Avro Data files, objects are stored in blocks that may be compressed.
- Synchronization markers are used between blocks to permit efficient splitting of files for MapReduce processing.
- By default, Avro data files are not compressed, but it is recommended to enable compression to reduce disk usage and increase read and write performance.
  - Deflate and Snappy compression is supported.
- Avro datafiles are widely supported across components in the Hadoop ecosystem (Pig, Hive, Spark, etc.), so they are a good default choice for a binary format.

Main Point

An Avro datafile (.avro) is language neutral and has a metadata section where the schema is stored, which makes the file self-describing. It also includes markers that can be used for splitting large datasets into subsets (HDFS blocks) which makes it very suitable for MapReduce processing.

Science & Technology of Consciousness: Veda is structured in self-referral consciousness; the universe is structured in the Veda, so the universe is also structured in self-referral consciousness; therefore it is obvious that self-referral consciousness is the source, course and goal of gaining knowledge.

Avro Data Types

- Avro provides rich data structures.
  - E.g., you can create a record that contains an array, an enumerated type, and a sub record.
- These datatypes can be created in any language, can be processed in Hadoop, and the results can be fed to a third language.
- For interoperability, implementations must support all Avro types.

Avro Primitive Data Types

- Avro defines a small number of primitive data types, which can be used to build application-specific data structures by writing schemas.

Type	Description	Schema
null	The absence of a value	"null"
boolean	A binary value	"boolean"
int	32-bit signed integer	"int"
long	64-bit signed integer	"long"
float	Single-precision (32-bit) IEEE 754 floating-point number	"float"
double	Double-precision (64-bit) IEEE 754 floating-point number	"double"
bytes	Sequence of 8-bit unsigned bytes	"bytes"
string	Sequence of Unicode characters	"string"

Avro Complex Data Types

- **array**
  - An ordered collection of objects. All objects in a particular array must have the same schema.
- **Schema example**

```
{
    "type": "array",
    "items": "long"
}
```

- **map**
  - An unordered collection of key-value pairs. Keys must be strings and values may be any type, although within a particular map, all values must have the same schema.
- **Schema example**

```
{
    "type": “map",
    “values": “string"
}
```

Avro Complex Data Types

- **record**
  - A collection of named fields of any type.
  - **Schema example**

```
{
    "type"        : "record",
    "name"        : "WeatherRecord",
    "doc"         : "A weather reading.",
    "fields": [
                {"name": "year",  "type": "int"},
                {"name": "temperature",  "type": "int"},
                {"name": "stationId",   "type": "string"}
              ]
 }
```

Avro Complex Data Types

- **enum**
  - A set of named values.
- **Schema example**

```
{
   "type": "enum",
   "name": "Cutlery",
   "doc": "An eating utensil.",
   "symbols": ["KNIFE",
               "FORK","SPOON"]
}
```

- **fixed**
  - A fixed number of 8-bit unsigned bytes.
  - **Schema example**

```
{
   "type": "fixed",
   "name": "Md5Hash",
   "size": 16
}
```

Avro Complex Data Types

- **union**
  - A union of schemas. A union is represented by a JSON array, where each element in the array is a schema.
  - Data represented by a union must match one of the schemas in the union.
  - **Schema example**

```
[
   "null",
   "string",
   {"type": "map",
              "values": "string"}
]
```

```
{
  "type" : "record",
  "namespace" : "myCompany",
  "name" : "EmpDetails ",
  "fields" :
  [
    { "name" : "experience", "type": ["int", "null"] },
    { "name" : "age", "type": "int" }
  ]
}
```

Language Mapping

- Each Avro language API has a representation for each Avro type that is specific to the language.
  - For example, Avro's double type is represented in C, C++, and Java by a double, in Python by a float, and in Ruby by a Float.
- All languages support a dynamic mapping, which can be used even when the schema is not known ahead of runtime. Java calls this the ***Generic* mapping.**
- In addition, the Java and C++ implementations can generate code to represent the data for an Avro schema.
- Code generation, which is called the ***Specific* mapping** in Java, is an optimization that is useful when you have a copy of the schema before you read or write data.

General Working of Avro

To use Avro, you need to follow the given workflow:

- **Step 1:** Create schemas. Here you need to design Avro schema according to your data and desired output.
- **Step 2:** Read the schemas into your program. It is done in two ways:
  - **By Generating a Class Corresponding to Schema** – Compile the schema using Avro. This generates a class file corresponding to the schema. (code generation)
  - **By Using Parsers Library –** You can directly read the schema using parsers library.
- **Step 3:** Serialize the data using the serialization API provided for Avro, which is found in the package org.apache.avro.generic.
- **Step 4:** Deserialize the data using deserialization API provided for Avro, which is found in the package org.apache.avro.generic.

Use of Class AvroJob
(org.apache.avro.mapreduce.AvroJob)

- AvroJob class has utility methods for configuring jobs that work with Avro.
- When using Avro data as MapReduce keys and values, data must be wrapped in a suitable **AvroWrapper** implementation.
- MapReduce keys must be wrapped in an **AvroKey** object, and MapReduce values must be wrapped in an **AvroValue** object.
  - E.g. in wordcount program, if instead of using a *Text* and *IntWritable* output value, you would like to use Avro data with a schema of "string" and "int", respectively, you may parameterize your reducer with *AvroKey<String>* and *AvroValue<Integer>* types.
- Then, use the `setOutputKeySchema()` and `setOutputValueSchema()` methods of ***AvroJob*** to set writer schemas for the records you will generate.

Station-Temp-Year Example

- The output needs to be in the following format:
  (stationId in ascending, temperature in descending order)

011990-99999	10	1950
011990-99999	8	1901
011990-99999	7	1930
……	…	…
012650-99999	12	1960
012650-99999	10	2015
……		

- Create an Avro schema in the above record format and emit Avro Key from the mapper adhering to this schema.

Station-Temp-Year Example

Weather.avsc

```
{
    "type"      : "record",
    "name"      : "WeatherRecord",
    "doc"       : "A weather reading.",
    "fields"    : [
                {"name": "stationId",   "type": "string"},
                {"name": "temperature", "type": "float"},
                {"name": "year",  "type": "int"}
                ]
 }
```

You can use this Avro schema to serialize a Java object (POJO) into bytes, and deserialize these bytes back into the Java object.

Station-Temp-Year Example Without Code Generation

- Data in Avro is always stored with its corresponding schema, meaning we can always read a serialized item regardless of whether we know the schema ahead of time or not.
- This allows us to perform serialization and deserialization without code generation.
- First, we use a Parser to read our schema definition and create a *Schema* object.

```
Schema SCHEMA = new Schema.Parser().parse(new File("weather.avsc"));
```

- Using this ***SCHEMA***, we'll create weather records from the given input file for creating mapper output.

Station-Temp-Year Example Without Code Generation contd..

- Since we're not using code generation, we use the interface *GenericRecord* (org.apache.avro.generic) to represent weather records.
- *GenericRecord* uses the *schema* to verify that we only specify valid fields.
  - If we try to set a non-existent field, we'll get an *AvroRuntimeException* when we run the program.
- Create the object of *GenericRecord* by instantiating *GenericData.Record* class. Pass the schema object to its constructor.

```
GenericRecord record = new GenericData.Record(SCHEMA);
```

```
record.put("stationId", utils.getStationId());
record.put("temperature", utils.getAirTemperature());
record.put("year", utils.getYearInt());

context.write(new AvroKey<GenericRecord>(record), NullWritable.get());
```

Compiling the Schema (with code generation)

- Code generation allows us to automatically create classes based on our previously-defined schema.
- Once we have defined the relevant classes, there is no need to use the schema directly in our programs.

- We use the avro-tools jar to generate code as follows:

```
avro-tools compile schema
/home/cloudera/cs523/Avro/weather.avsc
/home/cloudera/cs523/Avro
```

Avro's Built-in Sorting

- Avro defines a standard sort order for data. This permits data written by one system to be efficiently sorted by another system.
- This can be an important optimization, as sort order comparisons are sometimes the most frequent per-object operation.
- Note also that Avro binary-encoded data can be efficiently ordered without deserializing it to objects.
- Data items may only be compared if they have identical schemas.
- Pairwise comparisons are implemented recursively with a depth-first, left-to-right traversal of the schema. The first mismatch encountered determines the order of the items.

Avro's Built-in Sorting contd..

- Avro let's you control which fields in an Avro record are used for partitioning, sorting and grouping in MapReduce.
- By default, all the fields in an Avro *map output key* are used for partitioning, sorting and grouping in MapReduce.
- The output is sorted across all the fields, and that the sorting is in field ordinal order.
  - This is pretty much what you'd expect if you write your own complex Writable type, and your comparator compared all the fields in order.
- Field ordering is ascending by default, but you can make it descending by setting the value of the "order" field to "descending":

Excluding fields for partitioning, sorting and grouping

- Avro gives us the ability to indicate that specific fields should be ignored when performing ordering functions.

```
{"name": "temperature", "type": "float", "order": "ignore"}
```

- In MapReduce these fields are ignored for sorting, partitioning and grouping, which basically means that we have the ability to perform secondary sorting.

Schema Evolution

- An important aspect of data management is schema evolution. After the initial schema is defined, applications may need to evolve it over time.
- When this happens, it's critical for the downstream consumers to be able to handle data encoded with both the old and the new schema seamlessly.
- In a fast-changing environment it should be possible to
  - write data to file with a schema (writer's schema)
  - change the schema (reader's schema)
    - Add extra fields
    - Delete fields
    - Rename fields
  - and still read the written file with changed schema.

Schema Evolution in Avro

- A reader of Avro data, whether from an RPC or a file, can always parse that data because its schema is provided.
- But that schema may not be exactly the schema that was expected.
  - For example, if the data was written with a different version of the software than it is read, then records may have had fields added or removed.
  - Avro has rich *schema resolution* capabilities to take care of such problems.
- We call the schema used to write the data as the writer's schema, and the schema that the application expects is called as the reader's schema.

Schema Evolution in Avro contd..

- Writer's schema is always provided to Reader
- So Reader can compare:
  - The schema used to write with
  - The schema expected by application at Reader's side
- Fields that match (name and type) are read
- Fields written that don't match are skipped
- Expected fields not written can be identified

Schema Evolution in Avro contd..

- When a reader adds a new field to the schema, the new field must be given a default value. This prevents errors when writers using an old version of the schema creates new files that will be missing the new field.
  - 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.

Rules for Changing Schema at Reader's side

There are a few rules you need to remember if you are modifying schema that is already in use in your store:

- For best results, always provide a default value for the fields in your schema. This makes it possible to delete fields later on at the writers' side if you decide it is necessary.
- You cannot change a field's data type. If you have decided that a field should be some data type other than what it was originally created using, then add a whole new field to your schema that uses the appropriate data type.
- You cannot rename an existing field. However, if you want to access the field by some name other than what it was originally created using, add and use aliases for the field.

Allowed Schema Modifications

These are the modifications you can safely perform to your schema without any concerns:

- A field with a default value is added or removed
- A field's default value is added, or changed
- A field's doc attribute is changed, added or removed
- A field's order attribute is changed, added or removed
- Field or type aliases are added, or removed
- A non-union type may be changed to a union type that contains only the original type, or vice-versa

Main Point

Avro has rich schema resolution capabilities. Schema used to read data need not be identical to the schema that was used to write the data. When you add a new field to the schema, the new field must be given a default value. Schema evolution is one of the important features of Avro. Science & Technology of Consciousness: Expansion of happiness is the purpose of life, and evolution is the process through which it is fulfilled. Life begins in a natural way, it evolves, and happiness expands. The expansion of happiness carries with it the expansion of intelligence, power, creativity, and everything that may be said to be of significance in life.

Evolution of Data: AVRO

- When using Avro, one of the most important things is to manage its schemas and reason about how those schemas should evolve.
- **Advantages**:
  - Data is fully typed
  - Data can be compressed automatically
  - Schema (defined with JSON) comes along with the data
  - Documentation is embedded in the schema
  - Data can be read across any language
  - Schema can evolve over time, in a safe manner (schema evolution)
  - No need to compile any type of interface or protocol definition files in order to use the serialization features of the framework.
- **Disadvantages**:
  - Avro support for some languages may be lacking (but the main ones are fine)
  - Can't print the data without using the Avro tools (because it’s compressed and serialized)

Avro Serialization Recap

- Schema-based serialization utility
  - Using these schemas, you can store serialized values in binary format using less space.
  - These values are stored without any metadata.
- Schema language is in JSON
  - Each language already has JSON parser
- Each language implements data reader and writer
- Code generation is optional
  - Sometimes useful in statistically typed languages
- Data is untagged
  - Schema is required to read/write