Using AVRO To Run Python Map Reduce Jobs

Overview

This article describes how AVRO can be used write hadoop map/reduce jobs in other languages. AVRO accomplishes this by providing a stock mapper /reducer implementation which
communicate with an external process which can in principle be executing a program written in any language compatible with AVRO. The subprocess is responsible for implementing the desired functionality for your application specific mapper and reducer. AVRO is used to serialize data and transport it via sockets back and forth between the external process and the java mapper/reducer daemons controlled by hadoop.

Currently, this feature is still being developed and is not part of any of the released versions of AVRO. Furthermore, python is the only language for which tethered jobs are actively being developed (AVRO-570). To obtain python bindings for tethered job you will need to download and apply the appropriate patch as described below.

Applying the patch

Checkout trunk. (The patch was last tested with revision 1158089)

```
svn co http://svn.apache.org/repos/asf/avro/trunk SVN_AVRO
```

(you can replace SVN_AVRO with where ever you want to check it out)

Download the latest patch from AVRO-570

Apply the patch

```
cd SVN_AVRO
patch -p0 < AVRO-570.patch
```

Replace AVRO-570.patch with the full path to the file where you saved the patch.

Build the java packages.

```
cd SVN_AVRO/lang/java
mvn package -DskipTests
mvn install
```

We skip the tests because some of the unittests on trunk (e.g AVRO-878) fail.

Build the python packages. **The unittests attempt to run a tethered map reduce job. It is therefore recommended that you run hadoop in standalone or pseudo distributed mode.**

```
cd SVN_AVRO/lang/py
ant test
```

There is a special ant target which runs only the python unit tests for tethered python map/reduce jobs

```
cd AVRO/lang/py
ant test-570
```

You can dump the output of the map reduce job run by the unit test as follows

```
export PYTHONPATH=SVN_AVRO/lang/py/build/src/
python SVN_AVRO/lang/py/build/src/avro/tool.py dump /tmp/mapred/out/part-00000.avpo
```

The output should be

```
{u'value': 1, u'key': u'brown'}
{u'value': 1, u'key': u'cow'}
{u'value': 1, u'key': u'dog'}
{u'value': 1, u'key': u'falls'}
{u'value': 1, u'key': u'fox'}
{u'value': 1, u'key': u'in'}
{u'value': 2, u'key': u'jumps'}
{u'value': 1, u'key': u'lazy'}
{u'value': 1, u'key': u'mainly'}
{u'value': 1, u'key': u'moon'}
{u'value': 1, u'key': u'on'}
{u'value': 2, u'key': u'over'}
{u'value': 1, u'key': u'plains'}
{u'value': 1, u'key': u'quick'}
{u'value': 1, u'key': u'rain'}
{u'value': 1, u'key': u'spain'}
{u'value': 6, u'key': u'the'}
```

Anatomy of a tethered map reduce job.

A tethered MR(map-reduce job) consists of a mapper and reducer as required for all MR jobs. The mapper is provided by the class **org.apache.avro.mapred.tether.TetherMapRunner** and the reducer **org.apache.avro.mapred.tether.TetherReducer**. Unlike a typical mapper/reducer implementation, **TetherMapRunner** and **TetherReducer** do not get overridden or otherwise customized for a particular application. Rather, the mapper and reducer serialize their input tuples using AVRO and then ship the data off to an external process (the tethered process). The tethered process implements the actual transformation performed for the map or reduce stage. After the tethered process transforms its input tuples into output tuples, it serializes the data and sends it back to **TetherMapRunner**/**TetherReducer**. Communication is handled using AVRO's RPC calls.

The java class **org.apache.avro.mapred.tether.TetheredProcess** manages the tethered subprocess. In particular it is responsible for starting the process when a mapper/reducer starts, and terminating the external process when the mapper/reducer is done processing tuples. The stdout and stderr of the external process is redirected to a log file to facilitate debugging.

Communication is handled using sockets. When the mapper/reducer daemon start, they grab an available socket on which they will listen for messages from the tethered process. We use environment variables to pass this socket to the tethered process so that the tethered process knows what socket the mapper/reducer daemon is listening on. Similarly, when the tethered process starts it grabs an available socket on which it will listen for messages coming from the mapper/reducer daemon. The tethered process identifies the port the mapper/reducer is listening on by reading the appropriate environment variable. The tethered process then issues an RPC call to the mapper/reducer daemon which tells the mapper/reducer daemon which port the tethered process is listening on.

There are two avro protocols which define the data structures and rpc calls used to communicate between the mapper/reducer daemons and the tethered process

```
SVN_AVRO/share/schemas/org/apache/avro/mapred/tether/InputProtocol.avpr
SVN_AVRO/share/schemas/org/apache/avro/mapred/tether/OutputProtocol.avpr
```

InputProtocol.avpr defines the messages that the mapper/reducer daemons send to the tethered process. In particular the **input** call is used to send a list of (key,value) pairs to the tethered process to be processed. The list of tuples are sent as an array of bytes. This byte array is then decoded by the tethered process using the appropriate schema.

OutputProtocol.avpr defines the messages that the tethered process sends to the mapper/reducer daemons.

The Python Side Of Things

The python API for tethered map reduce jobs make it easy to implement the desired transformations for the mapper/reducer in python code.

avro.tether.tether_task_runner.py - This is the main python script that gets executed by the tethered process. This script is responsible for executing the python code that will process the tuples for the mapper/reducer. When executed it takes a single argument which is the full *package.module.class* name of a subclass of **avro.tether.TetherTask**. This user defined subclass is where we put our application specific code for the mapper/reducer.

avro.tether.tether_task.TetherTask - This is a base class that defines the API for implementing MR jobs in python. You need to subclass this class in order to provide your application specific mapper and reducer. To define the MR job we basically need to overload 4 methods

1. Call TethetherTask._*init*_ with strings representing the appropriate schemas for the mapper input, mapper output, and reducer output.
2. overload TetherTask.map - This function is invoked once on each tuple passed to the mapper
3. overload TetherTask.reduce - This function is invoked once on each tuple passed to the reducer
4. overload TetherTask.reduceFlush - This function is invoked once for each group (i.e identical keys) of tuples processed by the reducer.

The schemas defining the output of the mapper and reducer will typically be records with one or more fields representing the key of the tuple and the remaining fields being the values.
Before the reduce phase, we need to group tuples according to a set of *key fields*. *We use the *order* property of a field to set the fields which act as the *key* Setting *order* to *ignore* tells the tethered job that the field is not used when ordering records and is therefore a value field. For the typical word count example, the schema would be

```
"""{"type":"record", "name":"Pair","namespace":"org.apache.avro.mapred","fields":[
             {"name":"key","type":"string"},
             {"name":"value","type":"long","order":"ignore"}]
             }"""
```

Submitting a tethered MR job

AVRO builds a bundled jar **avro-tools-1.6.0-SNAPSHOT-job.jar** suitable for submitting tethered jobs to hadoop.

To submit a tethered job, we submit this jar to hadoop

```
hadoop jar AVRO/tools/target/avro-tools-1.6.0-SNAPSHOT-job.jar tether ...
```

The executable inside the jar contains many subprograms. The command line argument
tether tells it to use the subprogram responsible for tethered map reduce jobs.

The usage for tether is

```
usage: tether
 -exec_args <arg>      A string containing the command line arguments to
                       pass to the tethered process. String should be
                       enclosed in quotes
 -exec_cached <arg>    (optional) boolean indicating whether or not the
                       exectuable should be distributed via distributed
                       cache
 -help                 print this message
 -in <arg>             comma-separated input paths
 -out <arg>            The output path.
 -outschema <arg>      schema file for output of reducer
 -outschemamap <arg>   (optional) map output schema file,  if different
                       from outschema
 -program <arg>        executable program, usually in HDFS
 -protocol <arg>       (optional) specifies the transport protocol 'http'
                       or 'sasl'
 -reduces <arg>        (optional) number of reducers
```

Submitting a python tethered job.

To submit a python tethered job we set the arguments for tether so that the tethered subprocess will execute a python program. For the typical word count example (see SVN_AVRO/lang/py/src/test/test_tether_word_count.py)

```
hadoop jar SVN_AVRO/tools/target/avro-tools-1.6.0-SNAPSHOT-job.jar tether --program python --in /INPUT/PATH --
out /OUTPUT/PATH --outschema /PATH/outschema.avsc --protocol http --exec_args AVRO_PYTHON_PACKAGE/tether
/tether_task_runner.py word_count_task.WordCountTask
```

By specifying "python" as the program, we tell **org.apache.avro.mapred.tether.TetheredProcess** to execute python in the tethered subprocess. This is the tethered program that will actually implement the mapper/reducer for our MR job. We then use **--exec_args** to specify which script should be executed in python.

The value of **--exec_args** is a space separated list of additional arguments to pass to the program (python in this case) which is executed in a subprocess. For a python job, the first argument is the script to be executed. As explained earlier, for tethered python programs we use **tether_task_runner.py** as the main entry point. The main script is provided as part of avro

```
AVRO_PYTHON_PACKAGE/tether/tether_task_runner.py
```

where AVRO_PYTHON_PACAKGE is the directory where avro is installed. Typically this will be something like

```
/usr/lib/python2.7/site-packages/avro
```

The second argument to --exec_args is the full dotted name, e.g

```
package.module.class
```

of the class that subclassess **TetherTask** and implements the functions map,reduce,reduceFlush which define the functionality for the mapper and reducer.

For the word count example this is,

```
word_count_task.WordCountTask
```

The package containing the class must be on your python path. In the next section we describe how to adjust the PYTHONPATH so that it can find your Task class

.

The option **--outschema** specifies the path to a schema file containing the output of the reducer. For the word count example the schema is just,

```
{"type":"record",
 "name":"Pair","namespace":"org.apache.avro.mapred","fields":[
     {"name":"key","type":"string"},
     {"name":"value","type":"long","order":"ignore"}
 ]
}
```

The **--protocol** option specifies which transport protocol the mapper/reducer java daemons will use to communicate with the tethered process. *Currently, only the http transport protocol is supported
in python, so you must use http when running python tethered jobs.*

Setting your python path.

The easiest way to set the python path for your python program is by creating a simple executable, shell script which sets the environment variable PYTHONPATH and then executes python using the arguments specified by **-exec_args**. You would then specify this shell script for the *-program * option and leave *-exec_args * blank.

For the word count example, we might create a script that looks like

```
#!/bin/bash
export PYTHONPATH=SVN_AVRO/lang/py/build/src:SVN_AVRO/lang/py/build/test
python SVN_AVRO/tether/tether_task_runner.py word_count_task.WordCountTask
```

Make sure the script is executable by the user under which map-reduce jobs run.

We could then submit the tethered job as follows,

```
hadoop jar SVN_AVRO/tools/target/avro-tools-1.6.0-SNAPSHOT-job.jar tether --program word_script --in /INPUT
/PATH --out /OUTPUT/PATH --outschema /PATH/outschema.avsc --protocol http
```

where **word_script** is the script we created above.

Coding the Word Count Example in Python

We briefly explain how the typical word count example can be coded as a tethered python job.
As explained above we need to create a subclass of TetherTask. The code for this example can be found in the avro package in

```
SVN_AVRO/lang/py/test/word_count_task.py
```

The first thing we do is overload *_init_*

```
def __init__(self):
  """
  """

  inschema="""{"type":"string"}"""
  midschema="""{"type":"record", "name":"Pair","namespace":"org.apache.avro.mapred","fields":[
            {"name":"key","type":"string"},
            {"name":"value","type":"long","order":"ignore"}]
            }"""
  outschema=midschema
  TetherTask.__init__(self,inschema,midschema,outschema)


  #keep track of the partial sums of the counts
  self.psum=0
```

The init method tells TetherTask what the schemas are for each stage. The input to the mapper is just a string representing a line of text. The output of the mapper is
a record containing two fields "key" and "value". As noted in the previous section, the names of the fields are irrelevant for determining the grouping of tuples with respect to the reduce stage. Here, we set "order" to "ignore" to tell the tethered job to ignore the "value" field when grouping tuples for processing in the reduce stage.

The final line of code initializes a member variable which will be used in our reduce function to keep track of the word counts.

The map function is pretty straightforward.

```
def map(self,record,collector):

  words=record.split()

  for w in words:
    logging.info("WordCountTask.Map: word={0}".format(w))
    collector.collect({"key":w,"value":1})
```

Since the input schema for the mapper is a string representing a line of text, all we do is split the line by spaces and output tuples (word,1) for each word.

The reduce function is

```
def reduce(self,record, collector):
  self.psum+=record["value"]
```

The reduce function is invoked once for each tuple. Hadoop is responsible for ordering the tuples by the "key". So all we do is increment the value of psum to keep track of how many times we have seen a given word. In this example, we don't output any tuples in the reduce function.

Finally we implement reduceFlush,

```
def reduceFlush(self,record, collector):
  collector.collect({"key":record["key"],"value":self.psum})

  #reset the sum
  self.psum=0
```

reduceFlush is invoked once for each group of tuples in the reduce phase. *When *reduceFlush* is invoked 1) reduce has already been invoked on the tuple stored in record, and b) the tuple in record is the last tuple with this key. So the next call to *reduce* will be on the first tuple with the next unique value for the key. In *reduceFlush* we output a tuple containing the current word and the current count. We then reset psum because the next time reduce is invoked it will be the first occurrence of that word.

Inspecting the output

To inspect the output of the word count map reduce job we can use the avro dump tool to dump the contents of the avro output files

```
python AVROPKG/tool.py dump OUTPATH/part-00000.avro
```

Here AVROPKG is the path where the python avro package is installed. Typically this will be something like

```
PYTHON_INSTALL_DIR/site-packages/avro
```

OUTPATH is the output directory specified when you submitted the tether job.