Input and Output Interfaces

Table of contents

1 Set Up

No HCatalog-specific setup is required for the HCatInputFormat and HCatOutputFormat interfaces.

2 HCatInputFormat

The HCatInputFormat is used with MapReduce jobs to read data from HCatalog-managed tables.

HCatInputFormat exposes a Hadoop 0.20 MapReduce API for reading data as if it had been published to a table.

2.1 API

The API exposed by HCatInputFormat is shown below. It includes:

- `setInput`
- `setOutputSchema`
- `getTableSchema`

To use HCatInputFormat to read data, first instantiate an `InputJobInfo` with the necessary information from the table being read and then call setInput with the `InputJobInfo`.

You can use the `setOutputSchema` method to include a projection schema, to specify the output fields. If a schema is not specified, all the columns in the table will be returned.

You can use the `getTableSchema` method to determine the table schema for a specified input table.

```
/**
 * Set the input to use for the Job. This queries the metadata server with
 * the specified partition predicates, gets the matching partitions, puts
 * the information in the conf object. The inputInfo object is updated with
 * information needed in the client context
 * @param job the job object
 * @param inputJobInfo the input info for table to read
 * @throws IOException the exception in communicating with the metadata server
 */
public static void setInput(Job job,
    InputJobInfo inputJobInfo) throws IOException;

/**
 * Set the schema for the HCatRecord data returned by HCatInputFormat.
 * @param job the job object
 * @param hcatSchema the schema to use as the consolidated schema
 */
public static void setOutputSchema(Job job,HCatSchema hcatSchema)
  throws IOException;
```

```
/**
 * Get the HCatTable schema for the table specified in the HCatInputFormat.setInput
 * call on the specified job context. This information is available only after
 * HCatInputFormat.setInput has been called for a JobContext.
 * @param context the context
 * @return the table schema
 * @throws IOException if HCatInputFormat.setInput has not been called
 *                     for the current context
 */
public static HCatSchema getTableSchema(JobContext context)
  throws IOException;
```

3 HCatOutputFormat

HCatOutputFormat is used with MapReduce jobs to write data to HCatalog-managed tables.

HCatOutputFormat exposes a Hadoop 0.20 MapReduce API for writing data to a table. When a MapReduce job uses HCatOutputFormat to write output, the default OutputFormat configured for the table is used and the new partition is published to the table after the job completes.

3.1 API

The API exposed by HCatOutputFormat is shown below. It includes:

- `setOutput`
- `setSchema`
- `getTableSchema`

The first call on the HCatOutputFormat must be `setOutput`; any other call will throw an exception saying the output format is not initialized. The schema for the data being written out is specified by the `setSchema` method. You must call this method, providing the schema of data you are writing. If your data has the same schema as the table schema, you can use HCatOutputFormat.getTableSchema() to get the table schema and then pass that along to setSchema().

```
/**
 * Set the information about the output to write for the job. This queries the metadata
 * server to find the StorageHandler to use for the table. It throws an error if the
 * partition is already published.
 * @param job the job object
 * @param outputJobInfo the table output information for the job
 * @throws IOException the exception in communicating with the metadata server
 */
@SuppressWarnings("unchecked")
public static void setOutput(Job job, OutputJobInfo outputJobInfo) throws IOException;

/**
 * Set the schema for the data being written out to the partition. The
 * table schema is used by default for the partition if this is not called.
```

```
   * @param job the job object
   * @param schema the schema for the data
   * @throws IOException
   */
  public static void setSchema(final Job job, final HCatSchema schema) throws IOException;

  /**
   * Get the table schema for the table specified in the HCatOutputFormat.setOutput call
   * on the specified job context.
   * @param context the context
   * @return the table schema
   * @throws IOException if HCatOutputFormat.setOutput has not been called
   *                     for the passed context
   */
  public static HCatSchema getTableSchema(JobContext context) throws IOException;
```

4 HCatRecord

HCatRecord is the type supported for storing values in HCatalog tables.

The types in an HCatalog table schema determine the types of objects returned for different fields in HCatRecord. This table shows the mappings between Java classes for MapReduce programs and HCatalog data types:

HCatalog Data Type	Java Class in MapReduce	Values
TINYINT	java.lang.Byte	-128 to 127
SMALLINT	java.lang.Short	-2^{15} to $2^{15}-1$ (-32,768 to 32,767)
INT	java.lang.Integer	-2^{31} to $2^{31}-1$ (-2,147,483,648 to 2,147,483,647)
BIGINT	java.lang.Long	-2^{63} to $2^{63}-1$ (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
BOOLEAN	java.lang.Boolean	true or false
FLOAT	java.lang.Float	single-precision floating-point value
DOUBLE	java.lang.Double	double-precision floating-point value
BINARY	byte[]	binary data
STRING	java.lang.String	character string
STRUCT	java.util.List	structured data

HCatalog Data Type	Java Class in MapReduce	Values
ARRAY	java.util.List	values of one data type
MAP	java.util.Map	key-value pairs

5 Running MapReduce with HCatalog

Your MapReduce program needs to be told where the Thrift server is. The easiest way to do this is to pass the location as an argument to your Java program. You need to pass the Hive and HCatalog jars to MapReduce as well, via the -libjars argument.

```
export HADOOP_HOME=<path_to_hadoop_install>
export HCAT_HOME=<path_to_hcat_install>
export HIVE_HOME=<path_to_hive_install>
export LIB_JARS=$HCAT_HOME/share/hcatalog/hcatalog-core-0.5.0.jar,
$HIVE_HOME/lib/hive-metastore-0.10.0.jar,
$HIVE_HOME/lib/libthrift-0.7.0.jar,
$HIVE_HOME/lib/hive-exec-0.10.0.jar,
$HIVE_HOME/lib/libfb303-0.7.0.jar,
$HIVE_HOME/lib/jdo2-api-2.3-ec.jar,
$HIVE_HOME/lib/slf4j-api-1.6.1.jar

export HADOOP_CLASSPATH=$HCAT_HOME/share/hcatalog/hcatalog-core-0.5.0.jar:
$HIVE_HOME/lib/hive-metastore-0.10.0.jar:
$HIVE_HOME/lib/libthrift-0.7.0.jar:
$HIVE_HOME/lib/hive-exec-0.10.0.jar:
$HIVE_HOME/lib/libfb303-0.7.0.jar:
$HIVE_HOME/lib/jdo2-api-2.3-ec.jar:
$HIVE_HOME/conf:$HADOOP_HOME/conf:
$HIVE_HOME/lib/slf4j-api-1.6.1.jar

$HADOOP_HOME/bin/hadoop --config $HADOOP_HOME/conf jar <path_to_jar>
<main_class> -libjars $LIB_JARS <program_arguments>
```

This works but Hadoop will ship libjars every time you run the MapReduce program, treating the files as different cache entries, which is not efficient and may deplete the Hadoop distributed cache.

Instead, you can optimize to ship libjars using HDFS locations. By doing this, Hadoop will reuse the entries in the distributed cache.

```
bin/hadoop fs -copyFromLocal $HCAT_HOME/share/hcatalog/hcatalog-core-0.5.0.jar /tmp
bin/hadoop fs -copyFromLocal $HIVE_HOME/lib/hive-metastore-0.10.0.jar /tmp
bin/hadoop fs -copyFromLocal $HIVE_HOME/lib/libthrift-0.7.0.jar /tmp
bin/hadoop fs -copyFromLocal $HIVE_HOME/lib/hive-exec-0.10.0.jar /tmp
bin/hadoop fs -copyFromLocal $HIVE_HOME/lib/libfb303-0.7.0.jar /tmp
bin/hadoop fs -copyFromLocal $HIVE_HOME/lib/jdo2-api-2.3-ec.jar /tmp
bin/hadoop fs -copyFromLocal $HIVE_HOME/lib/slf4j-api-1.6.1.jar /tmp

export LIB_JARS=hdfs:///tmp/hcatalog-core-0.5.0.jar,
hdfs:///tmp/hive-metastore-0.10.0.jar,
```

```
hdfs:///tmp/libthrift-0.7.0.jar,
hdfs:///tmp/hive-exec-0.10.0.jar,
hdfs:///tmp/libfb303-0.7.0.jar,
hdfs:///tmp/jdo2-api-2.3-ec.jar,
hdfs:///tmp/slf4j-api-1.6.1.jar

# (Other statements remain the same.)
```

Authentication

If a failure results in a message like "2010-11-03 16:17:28,225 WARN hive.metastore ... - Unable to connect metastore with URI thrift://..." in /tmp/<username>/hive.log, then make sure you have run "kinit <username>@FOO.COM" to get a Kerberos ticket and to be able to authenticate to the HCatalog server.

5.1 Read Example

The following very simple MapReduce program reads data from one table which it assumes to have an integer in the second column, and counts how many different values it sees. That is, it does the equivalent of "`select col1, count(*) from $table group by col1;`".

```
public class GroupByAge extends Configured implements Tool {

    public static class Map extends
            Mapper<WritableComparable, HCatRecord, IntWritable, IntWritable> {

        int age;

        @Override
        protected void map(
                WritableComparable key,
                HCatRecord value,
                org.apache.hadoop.mapreduce.Mapper<WritableComparable, HCatRecord,
                        IntWritable, IntWritable>.Context context)
                throws IOException, InterruptedException {
            age = (Integer) value.get(1);
            context.write(new IntWritable(age), new IntWritable(1));
        }
    }

    public static class Reduce extends Reducer<IntWritable, IntWritable,
    WritableComparable, HCatRecord> {


      @Override
      protected void reduce(
              IntWritable key,
              java.lang.Iterable<IntWritable> values,
              org.apache.hadoop.mapreduce.Reducer<IntWritable, IntWritable,
                      WritableComparable, HCatRecord>.Context context)
              throws IOException, InterruptedException {
          int sum = 0;
          Iterator<IntWritable> iter = values.iterator();
          while (iter.hasNext()) {
```

```
            sum++;
            iter.next();
          }
          HCatRecord record = new DefaultHCatRecord(2);
          record.set(0, key.get());
          record.set(1, sum);

          context.write(null, record);
        }
    }

    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        args = new GenericOptionsParser(conf, args).getRemainingArgs();

        String inputTableName = args[0];
        String outputTableName = args[1];
        String dbName = null;

        Job job = new Job(conf, "GroupByAge");
        HCatInputFormat.setInput(job, InputJobInfo.create(dbName,
                inputTableName, null));
        // initialize HCatOutputFormat

        job.setInputFormatClass(HCatInputFormat.class);
        job.setJarByClass(GroupByAge.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(WritableComparable.class);
        job.setOutputValueClass(DefaultHCatRecord.class);
        HCatOutputFormat.setOutput(job, OutputJobInfo.create(dbName,
                outputTableName, null));
        HCatSchema s = HCatOutputFormat.getTableSchema(job);
        System.err.println("INFO: output schema explicitly set for writing:"
                + s);
        HCatOutputFormat.setSchema(job, s);
        job.setOutputFormatClass(HCatOutputFormat.class);
        return (job.waitForCompletion(true) ? 0 : 1);
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new GroupByAge(), args);
        System.exit(exitCode);
    }
}
```

Notice a number of important points about this program:

1. The implementation of Map takes HCatRecord as an input and the implementation of Reduce produces it as an output.
2. This example program assumes the schema of the input, but it could also retrieve the schema via HCatOutputFormat.getOutputSchema() and retrieve fields based on the results of that call.

3. The input descriptor for the table to be read is created by calling InputJobInfo.create. It requires the database name, table name, and partition filter. In this example the partition filter is null, so all partitions of the table will be read.
4. The output descriptor for the table to be written is created by calling OutputJobInfo.create. It requires the database name, the table name, and a Map of partition keys and values that describe the partition being written. In this example it is assumed the table is unpartitioned, so this Map is null.

To scan just selected partitions of a table, a filter describing the desired partitions can be passed to InputJobInfo.create. To scan a single partition, the filter string should look like: "`ds=20120401`" where the datestamp "`ds`" is the partition column name and "`20120401`" is the value you want to read (year, month, and day).

5.2 Filter Operators

A filter can contain the operators 'and', 'or', 'like', '()', '=', '<>' *(not equal)*, '<', '>', '<=' and '>='.

For example:

- `ds > "20110924"`
- `ds < "20110925"`
- `ds <= "20110925" and ds >= "20110924"`

5.3 Scan Filter

Assume for example you have a web_logs table that is partitioned by the column "`ds`". You could select one partition of the table by changing

```
HCatInputFormat.setInput(job, InputJobInfo.create(dbName, inputTableName, null));
```

to

```
HCatInputFormat.setInput(job,
                         InputJobInfo.create(dbName, inputTableName, "ds=\"20110924\""));
```

This filter must reference only partition columns. Values from other columns will cause the job to fail.

5.4 Write Filter

To write to a single partition you can change the above example to have a Map of key value pairs that describe all of the partition keys and values for that partition. In our example web_logs table, there is only one partition column (`ds`), so our Map will have only one entry. Change

```
HCatOutputFormat.setOutput(job, OutputJobInfo.create(dbName, outputTableName, null));
```

to

```
Map partitions = new HashMap<String, String>(1);
partitions.put("ds", "20110924");
HCatOutputFormat.setOutput(job, OutputJobInfo.create(dbName, outputTableName, partitions));
```

To write multiple partitions simultaneously you can leave the Map null, but all of the partitioning columns must be present in the data you are writing.