GenericUDAFCaseStudy

Writing GenericUDAFs: A Tutorial

User-Defined Aggregation Functions (UDAFs) are an excellent way to integrate advanced data-processing into Hive. Hive allows two varieties of UDAFs: simple and generic. Simple UDAFs, as the name implies, are rather simple to write, but incur performance penalties because of the use of Java Reflection, and do not allow features such as variable-length argument lists. Generic UDAFs allow all these features, but are perhaps not quite as intuitive to write as Simple UDAFs.

This tutorial walks through the development of the `histogram()` UDAF, which computes a histogram with a fixed, user-specified number of bins, using a constant amount of memory and time linear in the input size. It demonstrates a number of features of Generic UDAFs, such as a complex return type (an array of structures), and type checking on the input. The assumption is that the reader wants to write a UDAF for eventual submission to the Hive open-source project, so steps such as modifying the function registry in Hive and writing `.q` tests are also included. If you just want to write a UDAF, debug and deploy locally, see this page.

NOTE: In this tutorial, we walk through the creation of a `histogram()` function. Starting with the 0.6.0 release of Hive, this appears as the built-in function `histogram_numeric()`.

Preliminaries

Make sure you have the latest Hive trunk by running `svn up` in your Hive directory. More detailed instructions on downloading and setting up Hive can be found at Getting Started . Your local copy of Hive should work by running `build/dist/bin/hive` from the Hive root directory, and you should have some tables of data loaded into your local instance for testing whatever UDAF you have in mind. For this example, assume that a table called `normal` exists with a single `double` column called `val`, containing a large number of random number drawn from the standard normal distribution.

The files we will be editing or creating are as follows, relative to the Hive root:

`ql/src/java/org/apache/hadoop/hive/ql/udf/generic/GenericUDAFHistogram.java`	the main source file, to be created by you.	
`ql/src/java/org/apache/hadoop/hive/ql/exec/FunctionRegistry.java`	the function registry source file, to be edited by you to register our new `histogram()` UDAF into Hive's built-in function list.	
`ql/src/test/queries/clientpositive/udaf_histogram.q`	a file of sample queries for testing `histogram()` on sample data, to be created by you.	
`ql/src/test/results/clientpositive/udaf_histogram.q.out`	the expected output from your sample queries, to be created by `ant` in a later step.	
`ql/src/test/results/clientpositive/show_functions.q.out`	the expected output from the SHOW FUNCTIONS Hive query. Since we're adding a new `histogram()` function, this expected output will change to reflect the new function. This file will be modified by `ant` in a later step.	

Writing the source

This section gives a high-level outline of how to implement your own generic UDAF. For a concrete example, look at any of the existing UDAF sources present in `ql/src/java/org/apache/hadoop/hive/ql/udf/generic/` directory.

Overview

At a high-level, there are two parts to implementing a Generic UDAF. The first is to write a *resolver* class, and the second is to create an *evaluator* class. The resolver handles type checking and operator overloading (if you want it), and helps Hive find the correct evaluator class for a given set of argument types. The evaluator class then actually implements the UDAF logic. Generally, the top-level UDAF class extends the abstract base class **org.apache.hadoop.hive.ql.udf.GenericUDAFResolver2**, and the evaluator class(es) are written as static inner classes.

Writing the resolver

The resolver handles type checking and operator overloading for UDAF queries. The type checking ensures that the user isn't passing a **double** expression where an **integer** is expected, for example, and the operator overloading allows you to have different UDAF logic for different types of arguments.

The resolver class must extend **org.apache.hadoop.hive.ql.udf.GenericUDAFResolver2** (see #Resolver Interface Evolution for backwards compatibility information). We recommend that you extend the AbstractGenericUDAFResolver base class in order to insulate your UDAF from future interface changes in Hive.

Look at one of the existing UDAFs for the *import*s you will need.

```
#!Java
public class GenericUDAFHistogramNumeric extends AbstractGenericUDAFResolver {
  static final Log LOG = LogFactory.getLog(GenericUDAFHistogramNumeric.class.getName());

  @Override
  public GenericUDAFEvaluator getEvaluator(GenericUDAFParameterInfo info) throws SemanticException {
    // Type-checking goes here!

    return new GenericUDAFHistogramNumericEvaluator();
  }

  public static class GenericUDAFHistogramNumericEvaluator extends GenericUDAFEvaluator {
    // UDAF logic goes here!
  }
}
```

The code above shows the basic skeleton of a UDAF. The first line sets up a Log object that you can use to write warnings and errors to be fed into the Hive log. The GenericUDAFResolver class has a single overridden method: **getEvaluator**, which receives information about how the UDAF is being invoked. Of most interest is info.getParameters(), which provides an array of type information objects corresponding to the SQL types of the invocation parameters. For the histogram UDAF, we want two parameters: the numeric column over which to compute the histogram, and the number of histogram bins requested. The very first thing to do is to check that we have exactly two parameters (lines 3-6 below). Then, we check that the first parameter has a primitive type, and not an array or map, for example (lines 9-13). However, not only do we want it to be a primitive type column, but we also want it to be numeric, which means that we need to throw an exception if a STRING type is given (lines 14-28). BOOLEAN is excluded because the "histogram" estimation problem can be solved with a simple COUNT() query. Lines 30-41 illustrate similar type checking for the second parameter to the histogram() UDAF – the number of histogram bins. In this case, we insist that the number of histogram bins is an integer.

```
#!Java
  public GenericUDAFEvaluator getEvaluator(GenericUDAFParameterInfo info) throws SemanticException {
    TypeInfo [] parameters = info.getParameters();
    if (parameters.length != 2) {
      throw new UDFArgumentTypeException(parameters.length - 1,
          "Please specify exactly two arguments.");
    }

    // validate the first parameter, which is the expression to compute over
    if (parameters[0].getCategory() != ObjectInspector.Category.PRIMITIVE) {
      throw new UDFArgumentTypeException(0,
          "Only primitive type arguments are accepted but "
          + parameters[0].getTypeName() + " was passed as parameter 1.");
    }
    switch (((PrimitiveTypeInfo) parameters[0]).getPrimitiveCategory()) {
    case BYTE:
    case SHORT:
    case INT:
    case LONG:
    case FLOAT:
    case DOUBLE:
      break;
    case STRING:
    case BOOLEAN:
    default:
      throw new UDFArgumentTypeException(0,
          "Only numeric type arguments are accepted but "
          + parameters[0].getTypeName() + " was passed as parameter 1.");
    }

    // validate the second parameter, which is the number of histogram bins
    if (parameters[1].getCategory() != ObjectInspector.Category.PRIMITIVE) {
      throw new UDFArgumentTypeException(1,
          "Only primitive type arguments are accepted but "
          + parameters[1].getTypeName() + " was passed as parameter 2.");
    }
    if( ((PrimitiveTypeInfo) parameters[1]).getPrimitiveCategory()
        != PrimitiveObjectInspector.PrimitiveCategory.INT) {
      throw new UDFArgumentTypeException(1,
          "Only an integer argument is accepted as parameter 2, but "
          + parameters[1].getTypeName() + " was passed instead.");
    }
    return new GenericUDAFHistogramNumericEvaluator();
  }
```

A form of operator overloading could be implemented here. Say you had two completely different histogram construction algorithms – one designed for working with integers only, and the other for double data types. You would then create two separate Evaluator inner classes, and depending on the type of the input expression, would return the correct one as the return value of the Resolver class.

CAVEAT: The histogram function is supposed to be used in a manner resembling the following: `SELECT histogram_numeric(age, 30) FROM employees;`, which means estimate the distribution of employee ages using 30 histogram bins. However, within the resolver class, there is no way of telling if the second argument is a constant or an integer column with a whole bunch of different values. Thus, a pathologically twisted user could write something like: `SELECT histogram_numeric(age, age) FROM employees;`, assuming that age is an integer column. Of course, this makes no sense whatsoever, but it would validate correctly in the Resolver type-checking code above.

We'll deal with this problem in the Evaluator.

Writing the evaluator

All evaluators must extend from the abstract base class org.apache.hadoop.hive.ql.udf.generic.GenericUDAFEvaluator. This class provides a few abstract methods that must be implemented by the extending class. These methods establish the processing semantics followed by the UDAF. The following is the skeleton of an Evaluator class.

```
#!Java
  public static class GenericUDAFHistogramNumericEvaluator extends GenericUDAFEvaluator {

    // For PARTIAL1 and COMPLETE: ObjectInspectors for original data
    private PrimitiveObjectInspector inputOI;
    private PrimitiveObjectInspector nbinsOI;

    // For PARTIAL2 and FINAL: ObjectInspectors for partial aggregations (list of doubles)
    private StandardListObjectInspector loi;


    @Override
    public ObjectInspector init(Mode m, ObjectInspector[] parameters) throws HiveException {
      super.init(m, parameters);
      // return type goes here
    }

    @Override
    public Object terminatePartial(AggregationBuffer agg) throws HiveException {
      // return value goes here
    }

    @Override
    public Object terminate(AggregationBuffer agg) throws HiveException {
      // final return value goes here
    }

    @Override
    public void merge(AggregationBuffer agg, Object partial) throws HiveException {
    }

    @Override
    public void iterate(AggregationBuffer agg, Object[] parameters) throws HiveException {
    }

    // Aggregation buffer definition and manipulation methods
    static class StdAgg implements AggregationBuffer {
    };

    @Override
    public AggregationBuffer getNewAggregationBuffer() throws HiveException {
    }

    @Override
    public void reset(AggregationBuffer agg) throws HiveException {
    }
  }
```

What do all these functions do? The following is a brief summary of each function, in (roughly) chronological order of being called. It's **very** important to remember that the computation of your aggregation must be arbitrarily divisible over the data. Think of it like writing a divide-and-conquer algorithm where the partitioning of the data is completely out of your control and handled by Hive. More formally, given any subset of the input rows, you should be able to compute a partial result, and also be able to merge any pair of partial results into another partial result. This naturally makes it difficult to port over many existing algorithms, but should guarantee researchers jobs for quite some time.

Function	Purpose
init	Called by Hive to initialize an instance of your UDAF evaluator class.
getNewAggregationBuffer	Return an object that will be used to store temporary aggregation results.
iterate	Process a new row of data into the aggregation buffer
terminatePartial	Return the contents of the current aggregation in a persistable way. Here persistable means the return value can only be built up in terms of Java primitives, arrays, primitive wrappers (e.g. Double), Hadoop Writables, Lists, and Maps. Do NOT use your own classes (even if they implement java.io.Serializable), otherwise you may get strange errors or (probably worse) wrong results.
merge	Merge a partial aggregation returned by **terminatePartial** into the current aggregation
terminate	Return the final result of the aggregation to Hive

For writing the `histogram()` function, the following is the strategy that was adopted.

getNewAggregationBuffer

The aggregation buffer for a histogram is a list of (x,y) pairs that represent the histogram's bin centers and heights. In addition, the aggregation buffer also stores two integers with the maximum number of bins (a user-specified parameter), and the current number of bins used. The aggregation buffer is initialized to a 'not ready' state with the number of bins set to 0. This is because Hive makes no distinction between a constant parameter supplied to a UDAF and a column from a table; thus, we have no way of knowing how many bins the user wants in their histogram until the first call to `iterate()`.

iterate

The first thing we do in `iterate()` is to check whether the histogram object in our aggregation buffer is initialized. If it is not, we parse our the second argument to `iterate()`, which is the number of histogram bins requested by the user. We do this exactly once and initialize the histogram object. Note that error checking is performed here – if the user supplied a negative number or zero for the number of histogram bins, a `HiveException` is thrown at this point and computation terminates.

Next, we parse out the actual input data item (a number) and add it to our histogram estimation in the aggregation buffer. See the `GenericUDAFHistogramNumeric.java` file for details on the heuristic used to construct a histogram.

terminatePartial

The current histogram approximation is serialized as a list of `DoubleWritable` objects. The first two doubles in the list indicate the maximum number of histogram bins specified by the user and number of bins current used. The remaining entries are (x,y) pairs from the current histogram approximation.

merge

At this point, we have a (possibly uninitialized) histogram estimation, and have been requested to merge it with another estimation performed on a separate subset of the rows. If **N** is the number of histogram bins specified by the user, the current heuristic first builds a histogram with all **2N** bins from both estimations, and then iteratively merges the closest pair of bins until only **N** bins remain.

terminate

The final return type from the `histogram()` function is an array of (x,y) pairs representing histogram bin centers and heights. These can be {{explode()}} ed into a separate table, or parsed using a script and passed to Gnuplot (for example) to visualize the histogram.

Modifying the function registry

Once the code for the UDAF has been written and the source file placed in `ql/src/java/org/apache/hadoop/hive/ql/udf/generic`, it's time to modify the function registry and incorporate the new function into Hive's list of functions. This simply involves editing `ql/src/java/org/apache/hadoop/hive/ql/exec/FunctionRegistry.java` to import your UDAF class and register it's name.

Please note that you will have to run the following command to update the output of the `show functions` Hive call:

```
ant test -Dtestcase=TestCliDriver -Dqfile=show_functions.q -Doverwrite=true
```

Compiling and running

```
ant package
build/dist/bin/hive
```

Creating the tests

System-level tests consist of writing some sample queries that operate on sample data, generating the expected output from the queries, and making sure that things don't break in the future in terms of expected output. Note that the expected output is passed through `diff` with the actual output from Hive, so nondeterministic algorithms will have to compute some sort of statistic and then only keep the most significant digits (for example).

These are the simple steps needed for creating test cases for your new UDAF/UDF:

1. Create a file in `ql/src/test/queries/clientpositive/udaf_XXXXX.q` where XXXXX is your UDAF's name.
2. Put some queries in the `.q` file – hopefully enough to cover the full range of functionality and special cases.
3. For sample data, put your own in `hive/data/files` and load it using `LOAD DATA LOCAL INPATH...`, or reuse one of the files already there (grep for LOAD in the queries directory to see table names).
4. `touch ql/src/test/results/clientpositive/udaf_XXXX.q.out`
5. Run the following command to generate the output into the `.q.out` result file.

```
ant test -Dtestcase=TestCliDriver -Dqfile=udaf_XXXXX.q -Doverwrite=true
```

6. Run the following command to make sure your test runs fine.

```
ant test -Dtestcase=TestCliDriver -Dqfile=udaf_XXXXX.q
```

Checklist for open source submission

- Create an account on the Hive JIRA (https:--issues.apache.org-jira-browse-HIVE), create an issue for your new patch under the `Query Processor` component. Solicit discussion, incorporate feedback.
- Create your UDAF, integrate it into your local Hive copy.
- Run `ant package` from the Hive root to compile Hive and your new UDAF.
- Create `.q` tests and their corresponding `.q.out` output.
- Modify the function registry if adding a new function.
- Run `ant checkstyle` and examine `build/checkstyle/checkstyle-errors.html`, ensure that your source files conform to the Sun Java coding convention (with the 100 character line length exception).
- Run `ant test`, ensure that tests pass.
- Run `svn up`, ensure no conflicts with the main repository.
- Run `svn add` for whatever new files you have created.
- Ensure that you have added `.q` and `.q.out` tests.
- Ensure that you have run the `.q` tests for all new functionality.
- If adding a new UDAF, ensure that `show_functions.q.out` has been updated. Run `ant test -Dtestcase=TestCliDriver -Dqfile=show_functions.q -Doverwrite=true` to do this.
- Run `svn diff > HIVE-NNNN.1.patch` from the Hive root directory, where NNNN is the issue number the JIRA has assigned to you.
- Attach your file to the JIRA issue, describe your patch in the comments section.
- Ask for a code review in the comments.
- Click **Submit patch** on your issue after you have completed the steps above.
- It is also advisable to **watch** your issue to monitor new comments.

Tips, Tricks, Best Practices

- Hive can have unexpected behavior sometimes. It is best to first run `ant clean` if you're seeing something weird, ranging from unexplained exceptions to strings being incorrectly double-quoted.
- When serializing the aggregation buffer in a `terminatePartial()` call, if your UDAF only uses a few variables to represent the buffer (such as average), consider serializing them into a list of doubles, for example, instead of complicated named structures.
- Strongly cast generics wherever you can.
- Abstract core functionality from multiple UDAFs into its own class. Examples are `histogram_numeric()` and `percentile_approx()`, which both use the same core histogram estimation functionality.
- If you're stuck looking for an algorithm to adapt to the terminatePartial/merge paradigm, divide-and-conquer and parallel algorithms are predictably good places to start.
- Remember that the tests do a `diff` on the expected and actual output, and fail if there is any difference at all. An example of where this can fail horribly is a UDAF like `ngrams()`, where the output is a list of sorted (word,count) pairs. In some cases, different sort implementations might place words with the same count at different positions in the output. Even though the output is correct, the test will fail. In these cases, it's better to output (for example) only the counts, or some appropriate statistic on the counts, like the sum.

Resolver Interface Evolution

Old interface org.apache.hadoop.hive.ql.udf.GenericUDAFResolver was deprecated as of the 0.6.0 release. The key difference between GenericUDAFResolver and GenericUDAFResolver2 interface is the fact that the latter allows the evaluator implementation to access extra information regarding the function invocation such as the presence of DISTINCT qualifier or the invocation with the wildcard syntax such as FUNCTION(*). UDAFs that implement the deprecated GenericUDAFResolver interface will not be able to tell the difference between an invocation such as FUNCTION() or FUNCTION (*) since the information regarding specification of the wildcard is not available. Similarly, these implementations will also not be able to tell the difference between FUNCTION(EXPR) vs FUNCTION(DISTINCT EXPR) since the information regarding the presence of the DISTINCT qualifier is also not available.

Note that while resolvers which implement the GenericUDAFResolver2 interface are provided the extra information regarding the presence of DISTINCT qualifier of invocation with the wildcard syntax, they can choose to ignore it completely if it is of no significance to them. The underlying data filtering to compute DISTINCT values is actually done by Hive's core query processor and not by the evaluator or resolver; the information is provided to the resolver only for validation purposes. The AbstractGenericUDAFResolver base class offers an easy way to transition previously written UDAF implementations to migrate to the new resolver interface without having to re-write the implementation since the change from implementing GenericUDAFResolver interface to extending AbstractGenericUDAFResolver class is fairly minimal. (There may be issues with implementations that are part of an inheritance hierarchy since it may not be easy to change the base class.)