Hadoop Integration Guide

HP Vertica Analytic Database

Software Version: 7.0.x

Document Release Date: 5/2/2018

Contents

Using the Vertica Connector for Hadoop Map Reduce

Apache Hadoop is a software platform for performing distributed data processing. Micro Focus has created an interface between Vertica and Hadoop that lets you take advantage of both platform's strengths. Using the interface, a Hadoop application can access and store data in an Vertica database.

You use the Vertica Connector for Hadoop Map Reduce when you want Hadoop to be able to read data from and write data to Vertica. If you want Vertica to import data stored in a Hadoop Distributed File System (HDFS), use the Vertica Connector for HDFS. See Using the Vertica Connector for HDFS.

Prerequisites

Before you can use the Vertica Connector for Hadoop Map Reduce, you must install and configure Hadoop and be familiar with developing Hadoop applications. For details on installing and using Hadoop, please see the Apache Hadoop Web site.

See Vertica 7.0.x Supported Platforms for a list of the versions of Hadoop and Pig that the connector supports.

How Hadoop and Vertica Work Together

Hadoop and Vertica share some common features—they are both platforms that use clusters of hosts to store and operate on very large sets of data. Their key difference is the type of data they work with best. Hadoop is best suited for tasks involving unstructured data such as natural language content. Vertica works best on structured data that can be loaded into database tables.

The Vertica Connector for Hadoop Map Reduce lets you use these two platforms together to take advantage of their strengths. With the connector, you can use data from Vertica in a Hadoop job, and store the results of a Hadoop job in an Vertica database. For example, you can use Hadoop to extract and process keywords from a mass of unstructured text messages from a social media site, turning the material into structured data that is then loaded into an Vertica database. Once you have loaded the data into Vertica, you can perform many different analytic queries on the data far faster than by using Hadoop alone.

For more on how Hadoop and Vertica work together, see the Vertica Map Reduce Web page.

Note: Vertica supports two different programming interfaces—user defined functions (UDFs), and external procedures—which can handle complex processing tasks that are hard to resolve using SQL. You should consider using either of these interfaces instead of Hadoop if possible, since they do not require the overhead of maintaining a Hadoop cluster.

Since code written to use either user defined functions or external procedures runs on the Vertica cluster, they can impact the performance of your database. In general, if your application has straightforward operation that is hard to do in SQL, you should consider using UDFs or external procedures rather than resorting to Hadoop. If your application needs to perform significant data extraction and processing, Hadoop is often a better solution, because the processing takes place off of the Vertica cluster.

Hadoop and Vertica Cluster Scaling

Nodes in the Hadoop cluster connect directly to Vertica nodes when retrieving or storing data, allowing large volumes of data to be transferred in parallel. If the Hadoop cluster is larger than the Vertica cluster, this parallel data transfer can negatively impact the performance of the Vertica database.

To avoid performance impacts on your Vertica database, you should ensure that your Hadoop cluster cannot overwhelm your Vertica cluster. The exact sizing of each cluster depends on how fast your Hadoop cluster generates data requests and the load placed on the Vertica database by queries from other sources. A good rule of thumb to follow is for your Hadoop cluster to be no larger than your Vertica cluster.

Vertica Connector for Hadoop Features

The Vertica Connector for Hadoop Map Reduce:

- gives Hadoop access to data stored in Vertica.
- lets Hadoop store its results in Vertica. The Connector can create a table for the Hadoop data if it does not already exist.
- lets applications written in Apache Pig access and store data in Vertica.
- works with Hadoop streaming.

The Connector runs on each node in the Hadoop cluster, so the Hadoop nodes and Vertica nodes communicate with each other directly. Direct connections allow data to be transferred in parallel, dramatically increasing processing speed.

The Connector is written in Java, and is compatible with all platforms supported by Hadoop.

Note: To prevent Hadoop from potentially inserting multiple copies of data into Vertica, the Vertica Connector for Hadoop Map Reduce disables Hadoop's speculative execution feature.

Installing the Connector

Follow these steps to install the Vertica Connector for Hadoop Map Reduce:

If you have not already done so, download theVertica Connector for Hadoop Map Reduce installation package from the myVertica portal. Be sure to download the package that is compatible with your version of Hadoop. You can find your Hadoop version by issuing the following command on a Hadoop node:

```
# hadoop version
```

You will also need a copy of the Vertica JDBC driver which you can also download from the myVertica portal.

You need to perform the following steps on **each node** in your Hadoop cluster:

1. Copy the Vertica Connector for Hadoop Map Reduce `.zip` archive you downloaded to a temporary location on the Hadoop node.

2. Copy the Vertica JDBC driver .jar file to the same location on your node. If you haven't already, you can download this driver from the myVertica portal.

3. Unzip the connector `.zip` archive into a temporary directory. On Linux, you usually use the command `unzip`.

4. Locate the Hadoop home directory (the directory where Hadoop is installed). The location of this directory depends on how you installed Hadoop (manual install versus a package supplied by your Linux distribution or Cloudera). If you do not know the location of this directory, you can try the following steps:

   - See if the HADOOP_HOME environment variable is set by issuing the command `echo $HADOOP_HOME` on the command line.

   - See if Hadoop is in your path by typing `hadoop classpath` on the command line. If it is, this command lists the paths of all the jar files used by Hadoop, which should tell you the location of the Hadoop home directory.

   - If you installed using a `.deb` or `.rpm` package, you can look in `/usr/lib/hadoop`, as this is often the location where these packages install Hadoop.

5. Copy the file `hadoop-vertica.jar` from the directory where you unzipped the connector archive to the `lib` subdirectory in the Hadoop home directory.

6. Copy the Vertica JDBC driver file (`vertica_`*`x.x.x`*`_jdk_5.jar`) to the `lib` subdirectory in the Hadoop home directory (`$HADOOP_HOME/lib`).

7. Edit the `$HADOOP_HOME/conf/hadoop-env.sh` file, and find the lines:

   ```
   # Extra Java CLASSPATH elements.  Optional.
   # export HADOOP_CLASSPATH=
   ```

   Uncomment the `export` line by removing the hash character (#) and add the absolute path of the JDBC driver file you copied in the previous step. For example:

   ```
   export HADOOP_CLASSPATH=$HADOOP_HOME/lib/vertica_x.x.x_jdk_5.jar
   ```

   This environment variable ensures that Hadoop can find the Vertica JDBC driver.

8. Also in the `$HADOOP_HOME/conf/hadoop-env.sh` file, ensure that the `JAVA_HOME` environment variable is set to your Java installation.

9. If you want your application written in Pig to be able to access Vertica, you need to:

   a. Locate the Pig home directory. Often, this directory is in the same parent directory as the Hadoop home directory.

   b. Copy the file named `pig-vertica.jar` from the directory where you unpacked the connector `.zip` file to the `lib` subdirectory in the Pig home directory.

   c. Copy the Vertica JDBC driver file (`vertica_`*`x.x.x`*`_jdk_5.jar`) to the `lib` subdirectory in the Pig home directory.

Accessing Vertica Data From Hadoop

You need to follow three steps to have Hadoop fetch data from Vertica:

- Set the Hadoop job's input format to be `VerticaInputFormat`.
- Give the `VerticaInputFormat` class a query to be used to extract data from Vertica.
- Create a `Mapper` class that accepts `VerticaRecord` objects as input.

The following sections explain each of these steps in greater detail.

Selecting VerticaInputFormat

The first step to reading Vertica data from within a Hadoop job is to set its input format. You usually set the input format within the `run()` method in your Hadoop application's class. To set up the input format, pass the job.setInputFormatClass method the VerticaInputFormat.class, as follows:

```
    public int run(String[] args) throws Exception {
        // Set up the configuration and job objects
        Configuration conf = getConf();
        Job job = new Job(conf);
```

(later in the code)

```
        // Set the input format to retrieve data from
        // Vertica.
        job.setInputFormatClass(VerticaInputFormat.class);
```

Setting the input to the `VerticaInputFormat` class means that the `map` method will get `VerticaRecord` objects as its input.

Setting the Query to Retrieve Data From Vertica

A Hadoop job that reads data from your Vertica database has to execute a query that selects its input data. You pass this query to your Hadoop application using the `setInput` method of the `VerticaInputFormat` class. The Vertica Connector for Hadoop Map Reduce sends this query to the Hadoop nodes which then individually connect to Vertica nodes to run the query and get their input data.

A primary consideration for this query is how it segments the data being retrieved from Vertica. Since each node in the Hadoop cluster needs data to process, the query result needs to be segmented between the nodes.

There are three formats you can use for the query you want your Hadoop job to use when retrieving input data. Each format determines how the query's results are split across the Hadoop cluster. These formats are:

- A simple, self-contained query.
- A parameterized query along with explicit parameters.
- A parameterized query along with a second query that retrieves the parameter values for the first query from Vertica.

The following sections explain each of these methods in detail.

Using a Simple Query to Extract Data From Vertica

The simplest format for the query that Hadoop uses to extract data from Vertica is a self-contained hard-coded query. You pass this query in a String to the `setInput` method of the `VerticaInputFormat` class. You usually make this call in the `run` method of your Hadoop job class. For example, the following code retrieves the entire contents of the table named allTypes.

```
// Sets the query to use to get the data from the Vertica database.
// Simple query with no parameters
VerticaInputFormat.setInput(job,
    "SELECT * FROM allTypes ORDER BY key;");
```

The query you supply must have an ORDER BY clause, since the Vertica Connector for Hadoop Map Reduce uses it to figure out how to segment the query results between the Hadoop nodes. When it gets a simple query, the connector calculates limits and offsets to be sent to each node in the Hadoop cluster, so they each retrieve a portion of the query results to process.

Having Hadoop use a simple query to retrieve data from Vertica is the least efficient method, since the connector needs to perform extra processing to determine how the data should be segmented across the Hadoop nodes.

Using a Parameterized Query and Parameter Lists

You can have Hadoop retrieve data from Vertica using a parametrized query, to which you supply a set of parameters. The parameters in the query are represented by a question mark (`?`).

You pass the query and the parameters to the `setInput` method of the `VerticaInputFormat` class. You have two options for passing the parameters: using a discrete list, or by using a `Collection` object.

Using a Discrete List of Values

To pass a discrete list of parameters for the query, you include them in the `setInput` method call in a comma-separated list of string values, as shown in the next example:

```
// Simple query with supplied parameters
VerticaInputFormat.setInput(job,
    "SELECT * FROM allTypes WHERE key = ?", "1001", "1002", "1003");
```

The Vertica Connector for Hadoop Map Reduce tries to evenly distribute the query and parameters among the nodes in the Hadoop cluster. If the number of parameters is not a multiple of the number of nodes in the cluster, some nodes will get more parameters to process than others. Once the connector divides up the parameters among the Hadoop nodes, each node connects to a host in the Vertica database and executes the query, substituting in the parameter values it received.

This format is useful when you have a discrete set of parameters that will not change over time. However, it is inflexible because any changes to the parameter list requires you to recompile your Hadoop job. An added limitation is that the query can contain just a single parameter, because the `setInput` method only accepts a single parameter list. The more flexible way to use parameterized queries is to use a collection to contain the parameters.

Using a Collection Object

The more flexible method of supplying the parameters for the query is to store them into a `Collection` object, then include the object in the `setInput` method call. This method allows you to build the list of parameters at run time, rather than having them hard-coded. You can also use multiple parameters in the query, since you will pass a collection of `ArrayList` objects to `setInput` statement. Each `ArrayList` object supplies one set of parameter values, and can contain values for each parameter in the query.

The following example demonstrates using a collection to pass the parameter values for a query containing two parameters. The collection object passed to `setInput` is an instance of the `HashSet` class. This object contains four `ArrayList` objects added within the `for` loop. This example just adds dummy values (the loop counter and the string `"FOUR"`). In your own application, you usually calculate parameter values in some manner before adding them to the collection.

Note: If your parameter values are stored in Vertica, you should specify the parameters using a query instead of a collection. See Using a Query to Retrieve Parameters for a Parameterized

Query for details.

```
// Collection to hold all of the sets of parameters for the query.
Collection<List<Object>> params = new HashSet<List<Object>>() {
};

// Each set of parameters lives in an ArrayList. Each entry
// in the list supplies a value for a single parameter in
// the query. Here, ArrayList objects are created in a loop
// that adds the loop counter and a static string as the
// parameters. The ArrayList is then added to the collection.
for (int i = 0; i < 4; i++) {
    ArrayList<Object> param = new ArrayList<Object>();
    param.add(i);
    param.add("FOUR");
    params.add(param);
}

VerticaInputFormat.setInput(job,
        "select * from allTypes where key = ? AND NOT varcharcol = ?",
        params);
```

Scaling Parameter Lists for the Hadoop Cluster

Whenever possible, make the number of parameter values you pass to the Vertica Connector for Hadoop Map Reduce equal to the number of nodes in the Hadoop cluster because each parameter value is assigned to a single Hadoop node. This ensures that the workload is spread across the entire Hadoop cluster. If you supply fewer parameter values than there are nodes in the Hadoop cluster, some of the nodes will not get a value and will sit idle. If the number of parameter values is not a multiple of the number of nodes in the cluster, Hadoop randomly assigns the extra values to nodes in the cluster. It does not perform scheduling—it does not wait for a nodes finish its task and become free before assigning additional tasks. In this case, a node could become a bottleneck if it is assigned the longer-running portions of the job.

In addition to the number of parameters in the query, you should make the parameter values yield roughly the same number of results. Ensuring each [parameter yields the same number of results helps prevent a single node in the Hadoop cluster from becoming a bottleneck by having to process more data than the other nodes in the cluster.

Using a Query to Retrieve Parameter Values for a Parameterized Query

You can pass the Vertica Connector for Hadoop Map Reduce a query to extract the parameter values for a parameterized query. This query must return a single column of data that is used as parameters for the parameterized query.

To use a query to retrieve the parameter values, supply the `VerticaInputFormat` class's `setInput` method with the parameterized query and a query to retrieve parameters. For example:

```
// Sets the query to use to get the data from the Vertica database.
// Query using a parameter that is supplied by another query
VerticaInputFormat.setInput(job,
        "select * from allTypes where key = ?",
        "select distinct key from regions");
```

When it receives a query for parameters, the connector runs the query itself, then groups the results together to send out to the Hadoop nodes, along with the parameterized query. The Hadoop nodes then run the parameterized query using the set of parameter values sent to them by the connector.

Writing a Map Class That Processes Vertica Data

Once you have set up your Hadoop application to read data from Vertica, you need to create a `Map` class that actually processes the data. Your `Map` class's `map` method receives `LongWritable` values as keys and `VerticaRecord` objects as values. The key values are just sequential numbers that identify the row in the query results. The `VerticaRecord` class represents a single row from the result set returned by the query you supplied to the `VerticaInput.setInput` method.

Working with the VerticaRecord Class

Your `map` method extracts the data it needs from the `VerticaRecord` class. This class contains three main methods you use to extract data from the record set:

- `get` retrieves a single value, either by index value or by name, from the row sent to the map method.
- `getOrdinalPosition` takes a string containing a column name and returns the column's number.
- `getType` returns the data type of a column in the row specified by index value or by name. This method is useful if you are unsure of the data types of the columns returned by the query. The types are stored as integer values defined by the `java.sql.Types` class.

The following example shows a `Mapper` class and `map` method that accepts `VerticaRecord` objects. In this example, no real work is done. Instead two values are selected as the key and value to be passed on to the reducer.

```
public static class Map extends
    Mapper<LongWritable, VerticaRecord, Text, DoubleWritable> {
    // This mapper accepts VerticaRecords as input.
    public void map(LongWritable key, VerticaRecord value, Context context)
                    throws IOException, InterruptedException {

        // In your mapper, you would do actual processing here.
        // This simple example just extracts two values from the row of
        // data and passes them to the reducer as the key and value.
        if (value.get(3) != null && value.get(0) != null) {
            context.write(new Text((String) value.get(3)),
            new DoubleWritable((Long) value.get(0)));
        }
    }
}
```

If your Hadoop job has a reduce stage, all of the `map` method output is managed by Hadoop. It is not stored or manipulated in any way by Vertica. If your Hadoop job does not have a reduce stage, and needs to store its output into Vertica, your `map` method must output its keys as `Text` objects and values as `VerticaRecord` objects.

Writing Data to Vertica From Hadoop

There are three steps you need to take for your Hadoop application to store data in Vertica:

- Set the output value class of your Hadoop job to `VerticaRecord`.
- Set the details of the Vertica table where you want to store your data in the `VerticaOutputFormat` class.
- Create a `Reduce` class that adds data to a `VerticaRecord` object and calls its write method to store the data.

The following sections explain these steps in more detail.

Configuring Hadoop to Output to Vertica

To tell your Hadoop application to output data to Vertica you configure your Hadoop application to output to the Vertica Connector for Hadoop Map Reduce. You will normally perform these steps in your Hadoop application's `run` method. There are three methods that need to be called in order to set up the output to be sent to the connector and to set the output of the `Reduce` class, as shown in the following example:

```
// Set the output format of Reduce class. It will
// output VerticaRecords that will be stored in the
// database.
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(VerticaRecord.class);
// Tell Hadoop to send its output to the Vertica
// Vertica Connector for Hadoop Map Reduce.
job.setOutputFormatClass(VerticaOutputFormat.class);
```

The call to `setOutputValueClass` tells Hadoop that the output of the `Reduce.reduce` method is a `VerticaRecord` class object. This object represents a single row of an Vertica database table. You tell your Hadoop job to send the data to the connector by setting the output format class to `VerticaOutputFormat`.

Defining the Output Table

Call the `VerticaOutputFormat.setOutput` method to define the table that will hold the Hadoop application output:

```
VerticaOutputFormat.setOutput(jobObject, tableName, [truncate, ["columnName1 dataType1"
[,"columnNamen dataTypen" ...]] );
```

jobObject	The Hadoop job object for your application.
tableName	The name of the table to store Hadoop's output. If this table does not exist, the Vertica Connector for Hadoop Map Reduce automatically creates it. The name can be a full database.schema.table reference.

truncate	A Boolean controlling whether to delete the contents of *tableName* if it already exists. If set to `true`, any existing data in the table is deleted before Hadoop's output is stored. If set to `false` or not given, the Hadoop output is added to the existing data in the table.
"*columnName1 dataType1*"	The table column definitions, where *columnName1* is the column name and *dataType1* the SQL data type. These two values are separated by a space. If not specified, the existing table is used.

The first two parameters are required. You can add as many column definitions as you need in your output table.

You usually call the `setOutput` method in your Hadoop class's `run` method, where all other setup work is done. The following example sets up an output table named mrtarget that contains 8 columns, each containing a different data type:

```
// Sets the output format for storing data in Vertica. It defines the
// table where data is stored, and the columns it will be stored in.
VerticaOutputFormat.setOutput(job, "mrtarget", true, "a int",
     "b boolean", "c char(1)", "d date", "f float", "t timestamp",
     "v varchar", "z varbinary");
```

If the truncate parameter is set to `true` for the method call, and target table already exists in the Vertica database, the connector deletes the table contents before storing the Hadoop job's output.

Note: If the table already exists in the database, and the method call truncate parameter is set to `false`, the Vertica Connector for Hadoop Map Reduce adds new application output to the existing table. However, the connector does not verify that the column definitions in the existing table match those defined in the `setOutput` method call. If the new application output values cannot be converted to the existing column values, your Hadoop application can throw casting exceptions.

Writing the Reduce Class

Once your Hadoop application is configured to output data to Vertica and has its output table defined, you need to create the `Reduce` class that actually formats and writes the data for storage in Vertica.

The first step your `Reduce` class should take is to instantiate a `VerticaRecord` object to hold the output of the `reduce` method. This is a little more complex than just instantiating a base object, since the `VerticaRecord` object must have the columns defined in it that match the out table's columns (see Defining the Output Table for details). To get the properly configured `VerticaRecord` object, you pass the constructor the configuration object.

You usually instantiate the `VerticaRecord` object in your `Reduce` class's `setup` method, which Hadoop calls before it calls the `reduce` method. For example:

```
// Sets up the output record that will be populated by
// the reducer and eventually written out.
public void setup(Context context) throws IOException,
        InterruptedException {
    super.setup(context);
    try {
        // Instantiate a VerticaRecord object that has the proper
        // column definitions. The object is stored in the record
        // field for use later.
        record = new VerticaRecord(context.getConfiguration());
     } catch (Exception e) {
          throw new IOException(e);
     }
}
```

Storing Data in the VerticaRecord

Your `reduce` method starts the same way any other Hadoop reduce method does—it processes its input key and value, performing whatever reduction task your application needs. Afterwards, your `reduce` method adds the data to be stored in Vertica to the `VerticaRecord` object that was instantiated earlier. Usually you use the `set` method to add the data:

`VerticaRecord.set(`*`column, value`*`);`

column	The column to store the value in. This is either an integer (the column number) or a `String` (the column name, as defined in the table definition). **Note:** The set method throws an exception if you pass it the name of a column that does not exist. You should always use a try/catch block around any set method call that uses a column name.
value	The value to store in the column. The data type of this value must match the definition of the column in the table.

Note: If you do not have the `set` method validate that the data types of the value and the column match, the Vertica Connector for Hadoop Map Reduce throws a `ClassCastException` if it finds a mismatch when it tries to commit the data to the database. This exception causes a rollback of the entire result. By having the `set` method validate the data type of the value, you can catch and resolve the exception before it causes a rollback.

In addition to the `set` method, you can also use the `setFromString` method to have the Vertica Connector for Hadoop Map Reduce convert the value from `String` to the proper data type for the column:

`VerticaRecord.setFromString(`*`column`*`, "`*`value`*`");`

column	The column number to store the value in, as an integer.
value	A `String` containing the value to store in the column. If the `String` cannot be converted to the correct data type to be stored in the column, `setFromString` throws an exception (`ParseException` for date values, `NumberFormatException` numeric values).

Your `reduce` method must output a value for every column in the output table. If you want a column to have a null value you must explicitly set it.

After it populates the `VerticaRecord` object, your `reduce` method calls the `Context.write` method, passing it the name of the table to store the data in as the key, and the `VerticaRecord` object as the value.

The following example shows a simple `Reduce` class that stores data into Vertica. To make the example as simple as possible, the code doesn't actually process the input it receives, and instead just writes dummy data to the database. In your own application, you would process the key and values into data that you then store in the `VerticaRecord` object.

```
public static class Reduce extends
    Reducer<Text, DoubleWritable, Text, VerticaRecord> {
    // Holds the records that the reducer writes its values to.
    VerticaRecord record = null;


    // Sets up the output record that will be populated by
    // the reducer and eventually written out.
    public void setup(Context context) throws IOException,
        InterruptedException {
        super.setup(context);
        try {
            // Need to call VerticaOutputFormat to get a record object
            // that has the proper column definitions.
            record = new VerticaRecord(context.getConfiguration());
        } catch (Exception e) {
            throw new IOException(e);
        }
    }


    // The reduce method.
    public void reduce(Text key, Iterable<DoubleWritable> values,
        Context context) throws IOException, InterruptedException {

        // Ensure that the record object has been set up properly. This is
        // where the results are written.
        if (record == null) {
            throw new IOException("No output record found");
        }
        // In this part of your application, your reducer would process the
        // key and values parameters to arrive at values that you want to
        // store into the database. For simplicity's sake, this example
        // skips all of the processing, and just inserts arbitrary values
        // into the database.
        //
        // Use the .set method to store a value in the record to be stored
        // in the database. The first parameter is the column number,
        // the second is the value to store.
        //
        // Column numbers start at 0.
        //
        // Set record 0 to an integer value, you
        // should always use a try/catch block to catch the exception.
```

```
            try {
                record.set(0, 125);
            } catch (Exception e) {
                // Handle the improper data type here.
                e.printStackTrace();
            }

            // You can also set column values by name rather than by column
            // number. However, this requires a try/catch since specifying a
            // non-existent column name will throw an exception.
            try {
                // The second column, named "b", contains a Boolean value.
                record.set("b", true);
            } catch (Exception e) {
                // Handle an improper column name here.
                e.printStackTrace();
            }

            // Column 2 stores a single char value.
            record.set(2, 'c');


            // Column 3 is a date. Value must be a java.sql.Date type.
            record.set(3, new java.sql.Date(
                Calendar.getInstance().getTimeInMillis()));


            // You can use the setFromString method to convert a string
            // value into the proper data type to be stored in the column.
            // You need to use a try...catch block in this case, since the
            // string to value conversion could fail (for example, trying to
            // store "Hello, World!" in a float column is not going to work).
            try {
                record.setFromString(4, "234.567");
            } catch (ParseException e) {
                // Thrown if the string cannot be parsed into the data type
                // to be stored in the column.
                e.printStackTrace();
            }

            // Column 5 stores a timestamp
            record.set(5, new java.sql.Timestamp(
                Calendar.getInstance().getTimeInMillis()));
            // Column 6 stores a varchar
            record.set(6, "example string");
            // Column 7 stores a varbinary
            record.set(7, new byte[10]);
            // Once the columns are populated, write the record to store
            // the row into the database.
            context.write(new Text("mrtarget"), record);
        }
    }
```

Passing Parameters to the Vertica Connector for Hadoop Map Reduce At Run Time

Specifying the Location of the Connector .jar File

Recent versions of Hadoop fail to find the Vertica Connector for Hadoop Map Reduce classes automatically, even though they are included in the Hadoop `lib` directory. Therefore, you need to manually tell Hadoop where to find the connector .jar file using the `libjars` argument:

```
hadoop jar myHadoopApp.jar com.myorg.hadoop.myHadoopApp \
    -libjars $HADOOP_HOME/lib/hadoop-vertica.jar \
    . . .
```

Specifying the Database Connection Parameters

You need to pass connection parameters to the Vertica Connector for Hadoop Map Reduce when starting your Hadoop application, so it knows how to connect to your database. At a minimum, these parameters must include the list of hostnames in the Vertica database cluster, the name of the database, and the user name. The common parameters for accessing the database appear in the following table. Usually, you will only need the basic parameters listed in this table in order to start your Hadoop application.

Parameter	Description	Required	Default
`mapred.vertica.hostnames`	A comma-separated list of the names or IP addresses of the hosts in the Vertica cluster. You should list all of the nodes in the cluster here, since individual nodes in the Hadoop cluster connect directly with a randomly assigned host in the cluster. The hosts in this cluster are used for both reading from and writing data to the Vertica database, unless you specify a different output database (see below).	Yes	none
`mapred.vertica.port`	The port number for the Vertica database.	No	5433
`mapred.vertica.database`	The name of the database the Hadoop application should access.	Yes	
`mapred.vertica.username`	The username to use when connecting to the database.	Yes	
`mapred.vertica.password`	The password to use when connecting to the database.	No	empty

You pass the parameters to the connector using the `-D` command line switch in the command you use to start your Hadoop application. For example:

```
hadoop jar myHadoopApp.jar com.myorg.hadoop.myHadoopApp \
    -libjars $HADOOP_HOME/lib/hadoop-vertica.jar \
    -Dmapred.vertica.hostnames=Vertica01,Vertica02,Vertica03,Vertica04 \
    -Dmapred.vertica.port=5433 -Dmapred.vertica.username=exampleuser \
    -Dmapred.vertica.password=password123 -Dmapred.vertica.database=ExampleDB
```

Parameters for a Separate Output Database

The parameters in the previous table are all you need if your Hadoop application accesses a single Vertica database. You can also have your Hadoop application read from one Vertica database and write to a different Vertica database. In this case, the parameters shown in the previous table apply to the input database (the one Hadoop reads data from). The following table lists the parameters that you use to supply your Hadoop application with the connection information for the output database (the one it writes its data to). None of these parameters is required. If you do not assign a value to one of these output parameters, it inherits its value from the input database parameters.

Parameter	Description	Default
`mapred.vertica.hostnames.output`	A comma-separated list of the names or IP addresses of the hosts in the output Vertica cluster.	Input hostnames
`mapred.vertica.port.output`	The port number for the output Vertica database.	5433
`mapred.vertica.database.output`	The name of the output database.	Input database name
`mapred.vertica.username.output`	The username to use when connecting to the output database.	Input database user name
`mapred.vertica.password.output`	The password to use when connecting to the output database.	Input database password

Example Vertica Connector for Hadoop Map Reduce Application

This section presents an example of using the Vertica Connector for Hadoop Map Reduce to retrieve and store data from an Vertica database. The example pulls together the code that has appeared on the previous topics to present a functioning example.

This application reads data from a table named allTypes. The mapper selects two values from this table to send to the reducer. The reducer doesn't perform any operations on the input, and instead inserts arbitrary data into the output table named mrtarget.

```
package com.vertica.hadoop;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.math.BigDecimal;
import java.sql.Date;
import java.sql.Timestamp;
// Needed when using the setFromString method, which throws this exception.
import java.text.ParseException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import com.vertica.hadoop.VerticaConfiguration;
import com.vertica.hadoop.VerticaInputFormat;
import com.vertica.hadoop.VerticaOutputFormat;
import com.vertica.hadoop.VerticaRecord;
// This is the class that contains the entire Hadoop example.
public class VerticaExample extends Configured implements Tool {

  public static class Map extends
      Mapper<LongWritable, VerticaRecord, Text, DoubleWritable> {
      // This mapper accepts VerticaRecords as input.
      public void map(LongWritable key, VerticaRecord value, Context context)
                      throws IOException, InterruptedException {

          // In your mapper, you would do actual processing here.
          // This simple example just extracts two values from the row of
          // data and passes them to the reducer as the key and value.
          if (value.get(3) != null && value.get(0) != null) {
              context.write(new Text((String) value.get(3)),
              new DoubleWritable((Long) value.get(0)));
          }
      }
  }


      public static class Reduce extends
          Reducer<Text, DoubleWritable, Text, VerticaRecord> {
          // Holds the records that the reducer writes its values to.
          VerticaRecord record = null;
```

```
    // Sets up the output record that will be populated by
    // the reducer and eventually written out.
    public void setup(Context context) throws IOException,
        InterruptedException {
        super.setup(context);
        try {
            // Need to call VerticaOutputFormat to get a record object
            // that has the proper column definitions.
            record = new VerticaRecord(context.getConfiguration());
        } catch (Exception e) {
            throw new IOException(e);
        }
    }


    // The reduce method.
    public void reduce(Text key, Iterable<DoubleWritable> values,
        Context context) throws IOException, InterruptedException {

        // Ensure that the record object has been set up properly. This is
        // where the results are written.
        if (record == null) {
            throw new IOException("No output record found");
        }
        // In this part of your application, your reducer would process the
        // key and values parameters to arrive at values that you want to
        // store into the database. For simplicity's sake, this example
        // skips all of the processing, and just inserts arbitrary values
        // into the database.
        //
        // Use the .set method to store a value in the record to be stored
        // in the database. The first parameter is the column number,
        // the second is the value to store.
        //
        // Column numbers start at 0.
        //
        // Set record 0 to an integer value, you
        // should always use a try/catch block to catch the exception.



        try {
            record.set(0, 125);
        } catch (Exception e) {
            // Handle the improper data type here.
            e.printStackTrace();
        }

        // You can also set column values by name rather than by column
        // number. However, this requires a try/catch since specifying a
        // non-existent column name will throw an exception.
        try {
            // The second column, named "b", contains a Boolean value.
            record.set("b", true);
            } catch (Exception e) {
                // Handle an improper column name here.
                e.printStackTrace();
            }

            // Column 2 stores a single char value.
            record.set(2, 'c');


            // Column 3 is a date. Value must be a java.sql.Date type.
            record.set(3, new java.sql.Date(
                Calendar.getInstance().getTimeInMillis()));


            // You can use the setFromString method to convert a string
            // value into the proper data type to be stored in the column.
            // You need to use a try...catch block in this case, since the
            // string to value conversion could fail (for example, trying to
            // store "Hello, World!" in a float column is not going to work).

            try {
                record.setFromString(4, "234.567");
            } catch (ParseException e) {
                // Thrown if the string cannot be parsed into the data type
                // to be stored in the column.
                e.printStackTrace();
            }

            // Column 5 stores a timestamp
            record.set(5, new java.sql.Timestamp(
                Calendar.getInstance().getTimeInMillis()));
            // Column 6 stores a varchar
            record.set(6, "example string");
            // Column 7 stores a varbinary
            record.set(7, new byte[10]);
            // Once the columns are populated, write the record to store
            // the row into the database.
            context.write(new Text("mrtarget"), record);
        }
    }

  @Override
  public int run(String[] args) throws Exception {
      // Set up the configuration and job objects
      Configuration conf = getConf();
      Job job = new Job(conf);
      conf = job.getConfiguration();
      conf.set("mapreduce.job.tracker", "local");
      job.setJobName("vertica test");
      // Set the input format to retrieve data from
      // Vertica.
      job.setInputFormatClass(VerticaInputFormat.class);

      // Set the output format of the mapper. This is the interim
      // data format passed to the reducer. Here, we will pass in a
      // Double. The interim data is not processed by Vertica in any
      // way.
```

```
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(DoubleWritable.class);

        // Set the output format of the Hadoop application. It will
        // output VerticaRecords that will be stored in the
        // database.
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(VerticaRecord.class);
        job.setOutputFormatClass(VerticaOutputFormat.class);
        job.setJarByClass(VerticaExample.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);


        // Sets the output format for storing data in Vertica. It defines the
        // table where data is stored, and the columns it will be stored in.
        VerticaOutputFormat.setOutput(job, "mrtarget", true, "a int",
            "b boolean", "c char(1)", "d date", "f float", "t timestamp",
            "v varchar", "z varbinary");
        // Sets the query to use to get the data from the Vertica database.
        // Query using a list of parameters.
        VerticaInputFormat.setInput(job, "select * from allTypes where key = ?",
            "1", "2", "3");
        job.waitForCompletion(true);
        return 0;
    }
    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new VerticaExample(),
        args);
        System.exit(res);
    }
}
```

Compiling and Running the Example Application

To run the example Hadoop application, you first need to set up the allTypes table that the example reads as input. To set up the input table, save the following Perl script as `MakeAllTypes.pl` to a location on one of your Vertica nodes:

```
#!/usr/bin/perl
open FILE, ">datasource" or die $!;
for ($i=0; $i < 10; $i++) {
    print FILE $i . "|" . rand(10000);
    print FILE "|one|ONE|1|1999-01-08|1999-02-23 03:11:52.35";
    print FILE '|1999-01-08 07:04:37|07:09:23|15:12:34 EST|0xabcd|';
    print FILE '0xabcd|1234532|03:03:03' . qq(\n);
}
close FILE;
```

Then follow these steps:

1. Connect to the node where you saved the `MakeAllTypes.pl` file.
2. Run the `MakeAllTypes.pl` file. This will generate a file named `datasource` in the current directory.

   Note: If your node does not have Perl installed, can can run this script on a system that does have Perl installed, then copy the `datasource` file to a database node.

3. On the same node, use vsql to connect to your Vertica database.
4. Run the following query to create the allTypes table:

   ```
   CREATE TABLE allTypes (key identity,intcol integer,
                                 floatcol float,
                                 charcol char(10),
                                 varcharcol varchar,
                                 boolcol boolean,
                                 datecol date,
                                 timestampcol timestamp,
                                 timestampTzcol timestamptz,
                                 timecol time,
                                 timeTzcol timetz,
                                 varbincol varbinary,
                                 bincol binary,
                                 numcol numeric(38,0),
                                 intervalcol interval
                         );
   ```

5. Run the following query to load data from the `datasource` file into the table:

   ```
   COPY allTypes COLUMN OPTION (varbincol FORMAT 'hex', bincol FORMAT 'hex')
   FROM '/path-to-datasource/datasource' DIRECT;
   ```

   Replace *`path-to-datasource`* with the absolute path to the `datasource` file located in the same directory where you ran `MakeAllTypes.pl`.

Compiling the Example (optional)

The example code presented in this section is based on example code distributed along with the Vertica Connector for Hadoop Map Reduce in the file `hadoop-vertica-example.jar`. If you just want to run the example, skip to the next section and use the `hadoop-vertica-example.jar` file that came as part of the connector package rather than a version you compiled yourself.

To compile the example code listed in Example Vertica Connector for Hadoop Map Reduce Application, follow these steps:

1. Log into a node on your Hadoop cluster.
2. Locate the Hadoop home directory. See Installing the Connector for tips on how to find this directory.
3. If it is not already set, set the environment variable `HADOOP_HOME` to the Hadoop home directory:

   ```
   export HADOOP_HOME=path_to_Hadoop_home
   ```

   If you installed Hadoop using an `.rpm` or `.deb` package, Hadoop is usually installed in `/usr/lib/hadoop`:

   ```
   export HADOOP_HOME=/usr/lib/hadoop
   ```

4. Save the example source code to a file named `VerticaExample.java` on your Hadoop node.
5. In the same directory where you saved `VerticaExample.java`, create a directory named classes. On Linux, the command is:

   ```
   mkdir classes
   ```

6. Compile the Hadoop example:

   ```
   javac -classpath \
    $HADOOP_HOME/hadoop-core.jar:$HADOOP_HOME/lib/hadoop-vertica.jar \
   -d classes VerticaExample.java \
   && jar -cvf hadoop-vertica-example.jar -C classes .
   ```

   Note: If you receive errors about missing Hadoop classes, check the name of the `hadoop-code.jar` file. Most Hadoop installers (including the Cloudera) create a symbolic link named hadoop-core.jar to a version specific `.jar` file (such as `hadoop-core-0.20.203.0.jar`). If your Hadoop installation did not create this link, you will have to supply the `.jar` file name with the version number.

When the compilation finishes, you will have a file named `hadoop-vertica-example.jar` in the same directory as the `VerticaExample.java` file. This is the file you will have Hadoop run.

Running the Example Application

Once you have compiled the example, run it using the following command line:

```
hadoop jar hadoop-vertica-example.jar \
   com.vertica.hadoop.VerticaExample \
```

```
  -Dmapred.vertica.hostnames=VerticaHost01,VerticaHost02,... \
  -Dmapred.vertica.port=portNumber \
  -Dmapred.vertica.username=userName \
  -Dmapred.vertica.password=dbPassword \
  -Dmapred.vertica.database=databaseName
```

This command tells Hadoop to run your application's `.jar` file, and supplies the parameters needed for your application to connect to your Vertica database. Fill in your own values for the hostnames, port, user name, password, and database name for your Vertica database.

After entering the command line, you will see output from Hadoop as it processes data that looks similar to the following:

```
12/01/11 10:41:19 INFO mapred.JobClient: Running job: job_201201101146_0005
12/01/11 10:41:20 INFO mapred.JobClient:  map 0% reduce 0%
12/01/11 10:41:36 INFO mapred.JobClient:  map 33% reduce 0%
12/01/11 10:41:39 INFO mapred.JobClient:  map 66% reduce 0%
12/01/11 10:41:42 INFO mapred.JobClient:  map 100% reduce 0%
12/01/11 10:41:45 INFO mapred.JobClient:  map 100% reduce 22%
12/01/11 10:41:51 INFO mapred.JobClient:  map 100% reduce 100%
12/01/11 10:41:56 INFO mapred.JobClient: Job complete: job_201201101146_0005
12/01/11 10:41:56 INFO mapred.JobClient: Counters: 23
12/01/11 10:41:56 INFO mapred.JobClient:   Job Counters
12/01/11 10:41:56 INFO mapred.JobClient:     Launched reduce tasks=1
12/01/11 10:41:56 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=21545
12/01/11 10:41:56 INFO mapred.JobClient:     Total time spent by all reduces waiting
after reserving slots (ms)=0
12/01/11 10:41:56 INFO mapred.JobClient:     Total time spent by all maps waiting after
reserving slots (ms)=0
12/01/11 10:41:56 INFO mapred.JobClient:     Launched map tasks=3
12/01/11 10:41:56 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=13851
12/01/11 10:41:56 INFO mapred.JobClient:   File Output Format Counters
12/01/11 10:41:56 INFO mapred.JobClient:     Bytes Written=0
12/01/11 10:41:56 INFO mapred.JobClient:   FileSystemCounters
12/01/11 10:41:56 INFO mapred.JobClient:     FILE_BYTES_READ=69
12/01/11 10:41:56 INFO mapred.JobClient:     HDFS_BYTES_READ=318
12/01/11 10:41:56 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=89367
12/01/11 10:41:56 INFO mapred.JobClient:   File Input Format Counters
12/01/11 10:41:56 INFO mapred.JobClient:     Bytes Read=0
12/01/11 10:41:56 INFO mapred.JobClient:   Map-Reduce Framework
12/01/11 10:41:56 INFO mapred.JobClient:     Reduce input groups=1
12/01/11 10:41:56 INFO mapred.JobClient:     Map output materialized bytes=81
12/01/11 10:41:56 INFO mapred.JobClient:     Combine output records=0
12/01/11 10:41:56 INFO mapred.JobClient:     Map input records=3
12/01/11 10:41:56 INFO mapred.JobClient:     Reduce shuffle bytes=54
12/01/11 10:41:56 INFO mapred.JobClient:     Reduce output records=1
12/01/11 10:41:56 INFO mapred.JobClient:     Spilled Records=6
12/01/11 10:41:56 INFO mapred.JobClient:     Map output bytes=57
12/01/11 10:41:56 INFO mapred.JobClient:     Combine input records=0
12/01/11 10:41:56 INFO mapred.JobClient:     Map output records=3
12/01/11 10:41:56 INFO mapred.JobClient:     SPLIT_RAW_BYTES=318
12/01/11 10:41:56 INFO mapred.JobClient:     Reduce input records=3
```

Note: The version of the example supplied in the Hadoop Connector download package will produce more output, since it runs several input queries.

Verifying the Results

Once your Hadoop application finishes, you can verify it ran correctly by looking at the mrtarget table in your Vertica database:

Connect to your Vertica database using vsql and run the following query:

```
=> SELECT * FROM mrtarget;
```

The results should look like this:

```
  a  | b | c |     d      |    f    |           t            |       v        |
          z
-----+---+---+------------+---------+------------------------+----------------+---------
--------------------------------
 125 | t | c | 2012-01-11 | 234.567 | 2012-01-11 10:41:48.837 | example string |
\000\000\000\000\000\000\000\000\000\000
(1 row)
```

Using Hadoop Streaming with the Vertica Connector for Hadoop Map Reduce

Hadoop streaming allows you to create an ad-hoc Hadoop job that uses standard commands (such as UNIX command-line utilities) for its map and reduce processing. When using streaming, Hadoop executes the command you pass to it a mapper and breaks each line from its standard output into key and value pairs. By default, the key and value are separated by the first tab character in the line. These values are then passed to the standard input to the command that you specified as the reducer. See the Hadoop wiki's topic on streaming for more information. You can have a streaming job retrieve data from an Vertica database, store data into an Vertica database, or both.

Reading Data From Vertica in a Streaming Hadoop Job

To have a streaming Hadoop job read data from an Vertica database, you set the `inputformat` argument of the Hadoop command line to `com.vertica.deprecated.VerticaStreamingInput`. You also need to supply parameters that tell the Hadoop job how to connect to your Vertica database. See Passing Parameters to the Vertica Connector for Hadoop Map Reduce At Run Time for an explanation of these command-line parameters.

Note: The `VerticaStreamingInput` class is within the deprecated namespace because the current version of Hadoop (as of 0.20.1) has not defined a current API for streaming. Instead, the streaming classes conform to the Hadoop version 0.18 API.

In addition to the standard command-line parameters that tell Hadoop how to access your database, there are additional streaming-specific parameters you need to use that supply Hadoop with the query it should use to extract data from Vertica and other query-related options.

Parameter	Description	Required	Default
`mapred.vertica.input.query`	The query to use to retrieve data from the Vertica database. See Setting the Query to Retrieve Data from Vertica for more information.	Yes	none
`mapred.vertica.input.paramquery`	A query to execute to retrieve parameters for the query given in the `.input.query` parameter.	If query has parameter and no discrete parameters supplied	
`mapred.vertica.query.params`	Discrete list of parameters for the query.	If query has parameter and no parameter query supplied	
`mapred.vertica.input.delimiter`	The character to use for separating column values. The command you use as a mapper needs to split individual column values apart using this delimiter.	No	0xa
`mapred.vertica.input.terminator`	The character used to signal the end of a row of data from the query result.	No	0xb

The following command demonstrates reading data from a table named allTypes. This command uses the UNIX cat command as a mapper and reducer so it will just pass the contents through.

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-streaming-*.jar \
   -libjars $HADOOP_HOME/lib/hadoop-vertica.jar \
   -Dmapred.vertica.hostnames=VerticaHost01,VerticaHost02,... \
   -Dmapred.vertica.database=ExampleDB \
   -Dmapred.vertica.username=ExampleUser \
   -Dmapred.vertica.password=password123 \
   -Dmapred.vertica.port=5433 \
   -Dmapred.vertica.input.query="SELECT key, intcol, floatcol, varcharcol FROM allTypes
ORDER BY key" \
   -Dmapred.vertica.input.delimiter=, \
   -Dmapred.map.tasks=1 \
   -inputformat com.vertica.hadoop.deprecated.VerticaStreamingInput \
   -input /tmp/input -output /tmp/output -reducer /bin/cat -mapper /bin/cat
```

The results of this command are saved in the `/tmp/output` directory on your HDFS filesystem. On a four-node Hadoop cluster, the results would be:

```
# $HADOOP_HOME/bin/hadoop dfs -ls /tmp/output
Found 5 items

drwxr-xr-x   - release supergroup          0 2012-01-19 11:47 /tmp/output/_logs

-rw-r--r--   3 release supergroup         88 2012-01-19 11:47 /tmp/output/part-00000
-rw-r--r--   3 release supergroup         58 2012-01-19 11:47 /tmp/output/part-00001
-rw-r--r--   3 release supergroup         58 2012-01-19 11:47 /tmp/output/part-00002
-rw-r--r--   3 release supergroup         87 2012-01-19 11:47 /tmp/output/part-00003
# $HADOOP_HOME/bin/hadoop dfs -tail /tmp/output/part-00000
1       2,1,3165.75558015273,ONE,
5       6,5,1765.76024139635,ONE,
9       10,9,4142.54176256463,ONE,
# $HADOOP_HOME/bin/hadoop dfs -tail /tmp/output/part-00001
2       3,2,8257.77313710329,ONE,
6       7,6,7267.69718012601,ONE,
# $HADOOP_HOME/bin/hadoop dfs -tail /tmp/output/part-00002
3       4,3,443.188765520475,ONE,
7       8,7,4729.27825566408,ONE,
# $HADOOP_HOME/bin/hadoop dfs -tail /tmp/output/part-00003
0       1,0,2456.83076632307,ONE,
4       5,4,9692.61214265391,ONE,
8       9,8,3327.25019418294,ONE,13      1015,15,15.1515,FIFTEEN
2       1003,3,333.0,THREE
3       1004,4,0.0,FOUR
4       1005,5,0.0,FIVE
5       1007,7,0.0,SEVEN
6       1008,8,1.0E36,EIGHT
7       1009,9,-1.0E36,NINE
8       1010,10,0.0,TEN
9       1011,11,11.11,ELEVEN
```

Notes

- Even though the input is coming from Vertica, you need to supply the `-input` parameter to Hadoop for it to process the streaming job.
- The `-Dmapred.map.tasks=1` parameter prevents multiple Hadoop nodes from reading the same data from the database, which would result in Hadoop processing multiple copies of the data.

Writing Data to Vertica in a Streaming Hadoop Job

Similar to reading from a streaming Hadoop job, you write data to Vertica by setting the `outputformat` parameter of your Hadoop command to `com.vertica.deprecated.VerticaStreamingOutput`. This class requires key/value pairs, but the keys are ignored. The values passed to VerticaStreamingOutput are broken into rows and inserted into a target table. Since keys are ignored, you can use the keys to partition the data for the reduce phase without affecting Vertica's data transfer.

As with reading from Vertica, you need to supply parameters that tell the streaming Hadoop job how to connect to the database. See Passing Parameters to the Vertica Connector for Hadoop Map Reduce At Run Time for an explanation of these command-line parameters. If you are reading data from one Vertica database and writing to another, you need to use the output parameters, similarly if you were reading and writing to separate databases using a Hadoop application. There are also additional parameters that configure the output of the streaming Hadoop job, listed in the following table.

Parameter	Description	Required	Default
`mapred.vertica.output.table.name`	The name of the table where Hadoop should store its data.	Yes	none
`mapred.vertica.output.table.def`	The definition of the table. The format is the same as used for defining the output table for a Hadoop application. See Defining the Output Table for details.	If the table does not already exist in the database	
`mapred.vertica.output.table.drop`	Whether to truncate the table before adding data to it.	No	false
`mapred.vertica.output.delimiter`	The character to use for separating column values.	No	0x7 (ASCII bell character)
`mapred.vertica.output.terminator`	The character used to signal the end of a row of data..	No	0x8 (ASCII backspace)

The following example demonstrates reading two columns of data data from an Vertica database table named allTypes and writing it back to the same database in a table named hadoopout. The command provides the definition for the table, so you do not have to manually create the table beforehand.

```
hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-streaming-*.jar \
 -libjars $HADOOP_HOME/lib/hadoop-vertica.jar \
 -Dmapred.vertica.output.table.name=hadoopout \
 -Dmapred.vertica.output.table.def="intcol integer, varcharcol varchar" \
 -Dmapred.vertica.output.table.drop=true \
 -Dmapred.vertica.hostnames=VerticaHost01,VerticaHost02,VerticaHost03 \
 -Dmapred.vertica.database=ExampleDB \
 -Dmapred.vertica.username=ExampleUser \
 -Dmapred.vertica.password=password123 \
 -Dmapred.vertica.port=5433 \
 -Dmapred.vertica.input.query="SELECT intcol, varcharcol FROM allTypes ORDER BY key" \
 -Dmapred.vertica.input.delimiter=, \
 -Dmapred.vertica.output.delimiter=, \
 -Dmapred.vertica.input.terminator=0x0a \
 -Dmapred.vertica.output.terminator=0x0a \
 -inputformat com.vertica.hadoop.deprecated.VerticaStreamingInput \
 -outputformat com.vertica.hadoop.deprecated.VerticaStreamingOutput \
 -input /tmp/input \
 -output /tmp/output \
 -reducer /bin/cat \
 -mapper /bin/cat
```

After running this command, you can view the result by querying your database:

```
=> SELECT * FROM hadoopout;
 intcol | varcharcol
--------+------------
      1 | ONE
      5 | ONE
      9 | ONE
      2 | ONE
      6 | ONE
      0 | ONE
      4 | ONE
      8 | ONE
      3 | ONE
      7 | ONE
(10 rows)
```

Loading a Text File From HDFS into Vertica

One common task when working with Hadoop and Vertica is loading text files from the Hadoop Distributed File System (HDFS) into an Vertica table. You can load these files using Hadoop streaming, saving yourself the trouble of having to write custom map and reduce classes.

Note: Hadoop streaming is less efficient than a Java map/reduce Hadoop job, since it passes data through several different interfaces. Streaming is best used for smaller, one-time loads. If you need to load large amounts of data on a regular basis, you should create a standard Hadoop map/reduce job in Java or a script in Pig.

For example, suppose you have a text file in the HDFS you want to load contains values delimited by pipe characters (|), with each line of the file is terminated by a carriage return:

```
# $HADOOP_HOME/bin/hadoop dfs -cat /tmp/textdata.txt
1|1.0|ONE
2|2.0|TWO
3|3.0|THREE
```

In this case, the line delimiter poses a problem. You can easily include the column delimiter in the Hadoop command line arguments. However, it is hard to specify a carriage return in the Hadoop command line. To get around this issue, you can write a mapper script to strip the carriage return and replace it with some other character that is easy to enter in the command line and also does not occur in the data.

Below is an example of a mapper script written in Python. It performs two tasks:

- Strips the carriage returns from the input text and terminates each line with a tilde (~).
- Adds a key value (the string "streaming") followed by a tab character at the start of each line of the text file. The mapper script needs to do this because the streaming job to read text files skips the reducer stage. The reducer isn't necessary, since the all of the data being read in text file should be stored in the Vertica tables. However, `VerticaStreamingOutput` class requires key and values pairs, so the mapper script adds the key.

```
#!/usr/bin/python
import sys
for line in sys.stdin.readlines():
    # Get rid of carriage returns.
    # CR is used as the record terminator by Streaming.jar
    line = line.strip();
    # Add a key. The key value can be anything.
    # The convention is to use the name of the
    # target table, as shown here.
    sys.stdout.write("streaming\t%s~\n" % line)
```

The Hadoop command to stream text files from the HDFS into Vertica using the above mapper script appears below.

```
hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-streaming-*.jar \
    -libjars $HADOOP_HOME/lib/hadoop-vertica.jar \
    -Dmapred.reduce.tasks=0 \
    -Dmapred.vertica.output.table.name=streaming \
    -Dmapred.vertica.output.table.def="intcol integer, floatcol float, varcharcol
varchar" \
    -Dmapred.vertica.hostnames=VerticaHost01,VerticaHost02,VerticaHost03 \
    -Dmapred.vertica.port=5433 \
    -Dmapred.vertica.username=ExampleUser \
    -Dmapred.vertica.password=password123 \
    -Dmapred.vertica.database=ExampleDB \
    -Dmapred.vertica.output.delimiter="|" \
    -Dmapred.vertica.output.terminator="~" \
    -input /tmp/textdata.txt \
    -output output \
    -mapper "python path-to-script/mapper.py" \
    -outputformat com.vertica.hadoop.deprecated.VerticaStreamingOutput
```

Notes

- The `-Dmapred.reduce-tasks=0` parameter disables the streaming job's reducer stage. It does not need a reducer since the mapper script processes the data into the format that the `VerticaStreamingOutput` class expects.
- Even though the `VerticaStreamingOutput` class is handling the output from the mapper, you need to supply a valid output directory to the Hadoop command.

The result of running the command is a new table in the Vertica database:

```
=> SELECT * FROM streaming;
 intcol | floatcol | varcharcol
--------+----------+------------
      3 |        3 | THREE
      1 |        1 | ONE
      2 |        2 | TWO
(3 rows)
```

Accessing Vertica From Pig

The Vertica Connector for Hadoop Map Reduce includes a Java package that lets you access an Vertica database using Pig. You must copy this `.jar` to somewhere in your Pig installation's CLASSPATH (see Installing the Connector for details).

Registering the Vertica .jar Files

Before it can access Vertica, your Pig Latin script must register the Vertica-related `.jar` files. All of your Pig scripts should start with the following commands:

```
REGISTER 'path-to-pig-home/lib/vertica_7.0.x_jdk_5.jar';
REGISTER 'path-to-pig-home/lib/pig-vertica.jar';
```

These commands ensure that Pig can locate the Vertica JDBC classes, as well as the interface for the connector.

Reading Data From Vertica

To read data from an Vertica database, you tell Pig Latin's LOAD statement to use a SQL query and to use the `VerticaLoader` class as the load function. Your query can be hard coded, or contain a parameter. See Setting the Query to Retrieve Data from Vertica for details.

Note: You can only use a discrete parameter list or supply a query to retrieve parameter values—you cannot use a collection to supply parameter values as you can from within a Hadoop application.

The format for calling the `VerticaLoader` is:

```
com.vertica.pig.VerticaLoader('hosts','database','port','username','password');
```

hosts	A comma-separated list of the hosts in the Vertica cluster.
database	The name of the database to be queried.
port	The port number for the database.
username	The username to use when connecting to the database.
password	The password to use when connecting to the database. This is the only optional parameter. If not present, the connector assumes the password is empty.

The following Pig Latin command extracts all of the data from the table named allTypes using a simple query:

```
A = LOAD 'sql://{SELECT * FROM allTypes ORDER BY key}' USING
   com.vertica.pig.VerticaLoader('Vertica01,Vertica02,Vertica03',
  'ExampleDB','5433','ExampleUser','password123');
```

This example uses a parameter and supplies a discrete list of parameter values:

```
A = LOAD 'sql://{SELECT * FROM allTypes WHERE key = ?};{1,2,3}' USING
   com.vertica.pig.VerticaLoader('Vertica01,Vertica02,Vertica03',
   'ExampleDB','5433','ExampleUser','password123');
```

This final example demonstrates using a second query to retrieve parameters from the Vertica database.

```
A = LOAD 'sql://{SELECT * FROM allTypes WHERE key = ?};sql://{SELECT DISTINCT key FROM
allTypes}'
     USING com.vertica.pig.VerticaLoader('Vertica01,Vertica02,Vertica03','ExampleDB',
    '5433','ExampleUser','password123');
```

Writing Data to Vertica

To write data to an Vertica database, you tell Pig Latin's STORE statement to save data to a database table (optionally giving the definition of the table) and to use the `VerticaStorer` class as the save function. If the table you specify as the destination does not exist, and you supplied the table definition, the table is automatically created in your Vertica database and the data from the relation is loaded into it.

The syntax for calling the `VerticaStorer` is the same as calling `VerticaLoader`:

```
com.vertica.pig.VerticaStorer('hosts','database','port','username','password');
```

The following example demonstrates saving a relation into a table named hadoopOut which must already exist in the database:

```
STORE A INTO '{hadoopOut}' USING
    com.vertica.pig.VerticaStorer('Vertica01,Vertica02,Vertica03','ExampleDB','5433',
    'ExampleUser','password123');
```

This example shows how you can add a table definition to the table name, so that the table is created in Vertica if it does not already exist:

```
STORE A INTO '{outTable(a int, b int, c float, d char(10), e varchar, f boolean, g date,
   h timestamp, i timestamptz, j time, k timetz, l varbinary, m binary,
   n numeric(38,0), o interval)}' USING
   com.vertica.pig.VerticaStorer('Vertica01,Vertica02,Vertica03','ExampleDB','5433',
   'ExampleUser','password123');
```

Note: If the table already exists in the database, and the definition that you supply differs from the table's definition, the table is not dropped and recreated. This may cause data type errors when data is being loaded.

Using the Vertica Connector for HDFS

The Hadoop Distributed File System (HDFS) is where Hadoop usually stores its input and output files. It stores files across the Hadoop cluster redundantly, ensuring files remain available even if some nodes are down. It also makes Hadoop more efficient, by spreading file access tasks across the cluster to help limit I/O bottlenecks.

The Vertica Connector for HDFS lets you load files from HDFS into Vertica using the COPY statement. You can also create external tables that access data stored on HDFS as if it were a native Vertica table. The connector is useful if your Hadoop job does not directly store its data in Vertica using the Vertica Connector for Hadoop Map Reduce (see Using the Vertica Connector for Hadoop Map Reduce) or if you use HDFS to store files and want to process them using Vertica.

Note: The files you load from HDFS using the Vertica Connector for HDFS is usually in a delimited format, where column values are separated by a character such as a comma or a pipe character (|).

Like the Vertica Connector for Hadoop Map Reduce, the Vertica Connector for HDFS takes advantage of the distributed nature of both Vertica and Hadoop. Individual nodes in the Vertica cluster connect directly to nodes in the Hadoop cluster when loading multiple files from the HDFS. This parallel operation decreases load times.

The connector is read-only—it cannot write data to HDFS. Use it when you want to import data from HDFS into Vertica. If you want Hadoop to be able to read data from and write data to Vertica, use the Vertica Connector for Hadoop Map Reduce.

The Vertica Connector for HDFS can connect to a Hadoop cluster through unauthenticated and Kerberos-authenticated connections.

Vertica Connector for HDFS Requirements

The Vertica Connector for HDFS connects to the Hadoop file system using WebHDFS, a built-in component of HDFS that provides access to HDFS files to applications outside of Hadoop. WebHDFS was added to Hadoop in version 1.0, so your Hadoop installation must be version 1.0 or later. The the connector has been tested with Apache Hadoop 1.0.0, Hortonworks 1.1, and Cloudera CDH4.

In addition, the WebHDFS system must be enabled. See your Hadoop distribution's documentation for instructions on configuring and enabling WebHDFS.

Note: HTTPfs (also known as HOOP) is another method of accessing files stored in an HDFS. It relies on a separate server process that receives requests for files and retrieves them from the HDFS. Since it uses a REST API that is compatible with WebHDFS, it could theoretically work with the connector. However, the connector has not been tested with HTTPfs and Micro Focus does not support using the Vertica Connector for HDFS with HTTPfs. In addition, since all of the files retrieved from HDFS must pass through the HTTPfs server, it is less efficient than WebHDFS, which lets Vertica nodes directly connect to the Hadoop nodes storing the file blocks.

Kerberos Authentication Requirements

The Vertica Connector for HDFS can connect to the Hadoop file system using Kerberos authentication. To use Kerberos, your connector must meet these additional requirements:

- Your Vertica installation must be Kerberos-enabled.
- Your Hadoop cluster must be configured to use Kerberos authentication.
- Your connector must be able to connect to the Kerberos-enabled Hadoop Cluster.
- The Kerberos server must be running version 5.
- The Kerberos server must be accessible from every node in your Vertica cluster.
- You must have Kerberos principals (users) that map to Hadoop users. You use these principals to authenticate your Vertica users with the Hadoop cluster.

Before you can use the Vertica Connector for HDFS with Kerberos you must Install the Kerberos client and libraries on your Vertica cluster.

Testing Your Hadoop WebHDFS Configuration

To ensure that your Hadoop installation's WebHDFS system is configured and running, follow these steps:

1. Log into your Hadoop cluster and locate a small text file on the Hadoop filesystem. If you do not have a suitable file, you can create a file named `test.txt` in the `/tmp` directory using the following command:

   ```
   echo -e "A|1|2|3\nB|4|5|6" | hadoop fs -put - /tmp/test.txt
   ```

2. Log into a host in your Vertica database using the database administrator account.

3. If you are using Kerberos authentication, authenticate with the Kerberos server using the keytab file for a user who is authorized to access the file. For example, to authenticate as an user named exampleuser@MYCOMPANY.COM, use the command:

   ```
   $ kinit exampleuser@MYCOMPANY.COM -k -t /path/exampleuser.keytab
   ```

   Where *path* is the path to the keytab file you copied over to the node. You do not receive any message if you authenticate successfully. You can verify that you are authenticated by using the klist command:

```
$ klistTicket cache: FILE:/tmp/krb5cc_500
Default principal: exampleuser@MYCOMPANY.COM
Valid starting     Expires            Service principal
07/24/13 14:30:19  07/25/13 14:30:19  krbtgt/MYCOMPANY.COM@MYCOMPANY.COM
        renew until 07/24/13 14:30:19
```

4. Test retrieving the file:

   - If you are not using Kerberos authentication, run the following command from the Linux command line:

     ```
     curl -i -L \
     "http://
     hadoopNameNode:50070/webhdfs/v1/tmp/test.txt?op=OPEN&user.name=hadoopUserName"
     ```

     Replacing *hadoopNameNode* with the hostname or IP address of the name node in your Hadoop cluster, */tmp/test.txt* with the path to the file in the Hadoop filesystem you located in step 1, and *hadoopUserName* with the user name of a Hadoop user that has read access to the file.

     If successful, the command you will see output similar to the following:

     ```
     HTTP/1.1 200 OKServer: Apache-Coyote/1.1
     Set-Cookie:
     hadoop.auth="u=hadoopUser&p=password&t=simple&e=1344383263490&s=n8YB/CHFg56qNmRQRT
     qO0IdRMvE="; Version=1; Path=/
     Content-Type: application/octet-stream
     Content-Length: 16
     Date: Tue, 07 Aug 2012 13:47:44 GMT
     A|1|2|3
     B|4|5|6
     ```

   - If you are using Kerberos authentication, run the following command from the Linux command line:

     ```
     curl --negotiate -i -L -u:anyUser
     http://hadoopNameNode:50070/webhdfs/v1/tmp/test.txt?op=OPEN
     ```

     Replace *hadoopNameNode* with the hostname or IP address of the name node in your Hadoop cluster, and */tmp/test.txt* with the path to the file in the Hadoop filesystem you located in step 1.

     If successful, you will see output similar to the following:

     ```
     HTTP/1.1 401 UnauthorizedContent-Type: text/html; charset=utf-8
     WWW-Authenticate: Negotiate
     Content-Length: 0
     ```

```
Server: Jetty(6.1.26)
HTTP/1.1 307 TEMPORARY_REDIRECT
Content-Type: application/octet-stream
Expires: Thu, 01-Jan-1970 00:00:00 GMT
Set-Cookie: hadoop.auth="u=exampleuser&p=exampleuser@MYCOMPANY.COM&t=kerberos&
e=1375144834763&s=iY52iRvjuuoZ5iYG8G5g12O2Vwo=";Path=/
Location:
http://hadoopnamenode.mycompany.com:1006/webhdfs/v1/user/release/docexample/test.t
xt?
op=OPEN&delegation=JAAHcmVsZWFzZQdyZWxlYXNlAIoBQCrfpdGKAUBO7CnRju3TbBSlID_
osB658jfGf
RpEt8-u9WHymRJXRUJIREZTIGRlbGVnYXRpb24SMTAuMjAuMTAwLjkxOjUwMDcw&offset=0
Content-Length: 0
Server: Jetty(6.1.26)
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: 16
Server: Jetty(6.1.26)
A|1|2|3
B|4|5|6
```

If the curl command fails, you must review the error messages and resolve any issues before using the Vertica Connector for HDFS with your Hadoop cluster. Some debugging steps include:

- Verify the HDFS service's port number.
- Verify that the Hadoop user you specified exists and has read access to the file you are attempting to retrieve.

Installing the Vertica Connector for HDFS

The Vertica Connector for HDFS is not included as part of the Vertica Server installation. You must download it from my.vertica.com and install it on all nodes in your Vertica database.

The connector installation packages contains several support libraries in addition to the library for the connector. Unlike some other packages supplied by Micro Focus, you need to install these package on **all of the hosts** in your Vertica database so each host has a copy of the support libraries.

Downloading and Installing the Vertica Connector for HDFS Package

Following these steps to install the Vertica Connector for HDFS:

1. Use a Web browser to log into the myVertica portal.
2. Click the Download tab.

3. Locate the section for the Vertica Connector for HDFS that you want and download the installation package that matches the Linux distribution on your Vertica cluster.

4. Copy the installation package to **each host** in your database.

5. Log into each host as root and run the following command to install the connector package.

   - On Red Hat-based Linux distributions, use the command:

     ```
     rpm -Uvh /path/installation-package.rpm
     ```

     For example, if you downloaded the Red Hat installation package to the dbadmin home directory, the command to install the package is:

     ```
     rpm -Uvh /home/dbadmin/vertica-hdfs-connectors-7.0.x86_64.RHEL5.rpm
     ```

   - On Debian-based systems, use the command:

     ```
     dpkg -i /path/installation-package.deb
     ```

Once you have installed the connector package on each host, you need to load the connector library into Vertica. See Loading the HDFS User Defined Source for instructions.

Loading the HDFS User Defined Source

Once you have installed the Vertica Connector for HDFS package on each host in your database, you need to load the connector's library into Vertica and define the User Defined Source (UDS) in the Vertica catalog. The UDS is what you use to access data from the HDFS. The connector install package contains a SQL script named `install.sql` that performs these steps for you. To run it:

1. Log into to the Administration Host using the database administrator account.

2. Execute the installation script:

   ```
   vsql -f /opt/vertica/packages/hdfs_connectors/install.sql
   ```

3. Enter the database administrator password if prompted.

Note: You only need to run an installation script once in order to create the User Defined Source in the Vertica catalog. You do not need to run the install script on each node.

The SQL install script loads the Vertica Connector for HDFS library and defines the HDFS User Defined Source named HDFS. The output of running the installation script successfully looks like this:

```
                  version
-------------------------------------------
 Vertica Analytic Database v7.0.x
(1 row)
CREATE LIBRARY
CREATE SOURCE FUNCTION
```

Once the install script finishes running, the connector is ready to use.

Loading Data Using the Vertica Connector for HDFS

Once you have installed the Vertica Connector for HDFS, you can use the HDFS User Defined Source (UDS) in a COPY statement to load data from HDFS files.

The basic syntax for using the HDFS UDS in a COPY statement is:

```
COPY tableName SOURCE Hdfs(url='HDFSFileURL', username='username');
```

tableName	The name of the table to receive the copied data.
HDFSFileURL	A string containing one or more comma-separated URLs that identify the file or files to be read. See below for details. **Note:** If the URL contains commas, they must be escaped.
username	The username of a Hadoop user that has permissions to access the files you want to copy.

The HDFSFileURL parameter is a string containing one or more comma-separated HTTP URLs that identify the files in the HDFS that you want to load. The format for each URL in this string is:

```
http://NameNode:port/webhdfs/v1/HDFSFilePath
```

NameNode	The host name or IP address of the Hadoop cluster's name node.
Port	The port number on which the WebHDFS service is running. This is usually 50070 or 14000, but may be different in your Hadoop installation.
`webhdfs/v1/`	The protocol being used to retrieve the file. This portion of the URL is always the same, to tell Hadoop to use version 1 of the WebHDFS API.

HDFSFilePath	The path from the root of the HDFS filesystem to the file or files you want to load. This path can contain standard Linux wildcards. **Note:** Any wildcards you use to specify multiple input files must resolve to files only, and not include any directories. For example, if you specify the path `/user/HadoopUser/output/*`, and the `output` directory contains a subdirectory, the connector returns an error message.

For example, the following command loads a single file named `/tmp/test.txt` from the Hadoop cluster whose name node's host name is hadoop using the Vertica Connector for HDFS:

```
=> COPY testTable SOURCE Hdfs(url='http://hadoop:50070/webhdfs/v1/tmp/test.txt',->
username='hadoopUser');
 Rows Loaded
-------------
           2
(1 row)
```

Copying Files in Parallel

The basic COPY command shown above copies a single file using just a single host in the Vertica cluster. This is not efficient, since just a single Vertica host performs the data load.

To make the load process more efficient, the data you want to load should be in multiple files (which it usually is if you are loading the results of a Hadoop job). You can then load them all by using wildcards in the URL, by supplying multiple comma-separated URLs in the url parameter of the Hdfs user-defined source function call, or both. Loading multiple files through the Vertica Connector for HDFS results in a very efficient load since the Vertica hosts connect directly to individual nodes in the Hadoop cluster to retrieve files. It is less likely that a single node in either cluster will become a bottleneck that slows down the operation.

For example, the following statement loads all of the files whose filenames start with "part-" located in the `/user/hadoopUser/output` directory on the HDFS. Assuming there are at least as many files in this directory as there are nodes in your Vertica cluster, all of the nodes in your cluster will fetch and load the data from the HDFS.

```
=> COPY Customers SOURCE-> Hdfs
(url='http://hadoop:50070/webhdfs/v1/user/hadoopUser/output/part-*',
-> username='hadoopUser');
 Rows Loaded
-------------
       40008
(1 row)
```

Using multiple comma-separated URLs in the URL string you supply in the COPY statement lets you load data from multiple directories on the HDFS at once (or even data from different Hadoop clusters):

```
=> COPY Customers SOURCE-> Hdfs
(url='http://hadoop:50070/webhdfs/v1/user/HadoopUser/output/part-*,
-> http://hadoop:50070/webhdfs/v1/user/AnotherUser/part-*',
-> username='HadoopUser');
 Rows Loaded
-------------
       80016
(1 row)
```

Note: Vertica statements have a 65,000 character limit. If you supply too many long URLs in a single statement, you could go over this limit. Normally, you would only approach this limit if you have automated the generation of the COPY statement.

Viewing Rejected Rows and Exceptions

COPY statements that use the Vertica Connector for HDFS use the same method for recording rejections and exceptions as other COPY statements. Rejected rows and exceptions are saved to log files stored by default in the CopyErrorLogs subdirectory in the database's catalog directory. Due to the distributed nature of the Vertica Connector for HDFS, you cannot use the ON option to force all of the exception and rejected row information to be written to log files on a single Vertica host. You need to collect all of the log files from across the hosts in order to review all of the exception and rejection information.

Creating an External Table Based on HDFS Files

You can use the Vertica Connector for HDFS as a source for an external table that lets you directly perform queries on the contents of files on the Hadoop Distributed File System (HDFS). See Using External Tables in the Administrator's Guide for more information on external tables.

Using an external table to access data stored on an HDFS is useful when you need to extract data from files that are periodically updated, or have additional files added on the HDFS. It saves you from having to drop previously loaded data and then reload the data using a COPY statement. The external table always accesses the current version of the files on the HDFS.

Note: An external table performs a bulk load each time it is queried. Its performance is significantly slower than querying an internal Vertica table. You should only use external tables for infrequently-run queries (such as daily reports). If you need to frequently query the content of the HDFS files, you should either use COPY to load the entire content of the files into Vertica or save the results of a query run on an external table to an internal table which you then use for repeated queries.

To create an external table that reads data from HDFS, you use the Hdfs User Defined Source (UDS) in a CREATE EXTERNAL TABLE AS COPY statement. The COPY portion of this statement has the same format as the COPY statement used to load data from HDFS. See Loading Data Using the Vertica Connector for HDFS for more information.

The following simple example shows how to create an external table that extracts data from every file in the `/user/hadoopUser/example/output` directory using the Vertica Connector for HDFS.

```
=> CREATE EXTERNAL TABLE hadoopExample (A VARCHAR(10), B INTEGER, C INTEGER, D INTEGER)
-> AS COPY SOURCE Hdfs(url=
-> 'http://hadoop01:50070/webhdfs/v1/user/hadoopUser/example/output/*',
-> username='hadoopUser');
CREATE TABLE
=> SELECT * FROM hadoopExample;
   A    | B | C | D
-------+---+---+---
 test1 | 1 | 2 | 3
 test1 | 3 | 4 | 5
(2 rows)
```

Later, after another Hadoop job adds contents to the output directory, querying the table produces different results:

```
=> SELECT * FROM hadoopExample;
   A    | B  | C  | D
-------+----+----+----
 test3 | 10 | 11 | 12
 test3 | 13 | 14 | 15
 test2 |  6 |  7 |  8
 test2 |  9 |  0 | 10
 test1 |  1 |  2 |  3
 test1 |  3 |  4 |  5
(6 rows)
```

Load Errors in External Tables

Normally, querying an external table on HDFS does not produce any errors if rows rejected by the underlying COPY statement (for example, rows containing columns whose contents are incompatible with the data types in the table). Rejected rows are handled the same way they are in a standard COPY statement: they are written to a rejected data file, and are noted in the exceptions file. For more information on how COPY handles rejected rows and exceptions, see Capturing Load Rejections and Exceptions in the Administrator's Guide.

Rejections and exception files are created on all of the nodes that load data from the HDFS. You cannot specify a single node to receive all of the rejected row and exception information. These files are created on each Vertica node as they process files loaded through the Vertica Connector for HDFS.

Note: Since the the connector is read-only, there is no way to store rejection and exception information on the HDFS.

Fatal errors during the transfer of data (for example, specifying files that do not exist on the HDFS) do not occur until you query the external table. The following example shows what happens if you recreate the table based on a file that does not exist on HDFS.

```
=> DROP TABLE hadoopExample;
```

```
DROP TABLE
=> CREATE EXTERNAL TABLE hadoopExample (A INTEGER, B INTEGER, C INTEGER, D INTEGER)
-> AS COPY SOURCE HDFS(url='http://hadoop01:50070/webhdfs/v1/tmp/nofile.txt',
-> username='hadoopUser');
CREATE TABLE
=> SELECT * FROM hadoopExample;
ERROR 0:  Error calling plan() in User Function HdfsFactory at
[src/Hdfs.cpp:222], error code: 0, message: No files match
[http://hadoop01:50070/webhdfs/v1/tmp/nofile.txt]
```

Note that it is not until you actually query the table that the the connector attempts to read the file. Only then does it return an error.

Vertica Connector for HDFS Troubleshooting Tips

Here are a few things to check if you have issue using the Vertica Connector for HDFS:

- If you find a user is suddenly unable to connect to Hadoop through the connector in a Kerberos-enabled environment, it could be that someone exported a new keytab file for the user which invalidates existing keytab files. You can determine if this is the problem by comparing the key version number associated with the user's principal in Kerberos with the key version number stored in the keytab file.

  a. To find the key version number for a user in Kerberos, start the kadmin utility (kadmin.local if you are logged into the Kerberos Key Distribution Center) from the Linux command line and run the getprinc command for the user:

  ```
  $ sudo kadmin[sudo] password for dbadmin:
  Authenticating as principal root/admin@MYCOMPANY.COM with password.
  Password for root/admin@MYCOMPANY.COM:
  kadmin:  getprinc exampleuser@MYCOMPANY.COM
  Principal: exampleuser@MYCOMPANY.COM
  Expiration date: [never]
  Last password change: Fri Jul 26 09:40:44 EDT 2013
  Password expiration date: [none]
  Maximum ticket life: 1 day 00:00:00
  Maximum renewable life: 0 days 00:00:00
  Last modified: Fri Jul 26 09:40:44 EDT 2013 (root/admin@MYCOMPANY.COM)
  Last successful authentication: [never]
  Last failed authentication: [never]
  Failed password attempts: 0
  Number of keys: 2
  Key: vno 3, des3-cbc-sha1, no salt
  Key: vno 3, des-cbc-crc, no salt
  MKey: vno 0
  Attributes:
  Policy: [none]
  ```

In the above example, there are two keys stored for the user, both of which are at version number (vno) 3.

b. Use the klist command to get the version numbers of the keys stored in the keytab file:

```
$ sudo klist -ek exampleuser.keytabKeytab name: FILE:exampleuser.keytab
KVNO Principal
---- ----------------------------------------------------------------------
   2 exampleuser@MYCOMPANY.COM (des3-cbc-sha1)
   2 exampleuser@MYCOMPANY.COM (des-cbc-crc)
   3 exampleuser@MYCOMPANY.COM (des3-cbc-sha1)
   3 exampleuser@MYCOMPANY.COM (des-cbc-crc)
```

The first column in the output lists the key version number. In the above example, the keytab includes both key versions 2 and 3, so the keytab file can be used to authenticate the user with Kerberos.

- You may receive an error message similar to this:

```
ERROR 5118: UDL specified no execution nodes; at least one execution node must be
specified
```

To fix this error, ensure that all of the nodes in your cluster have the correct Vertica Connector for HDFS package installed. This error can occur if one or more of the nodes do not have the supporting libraries installed on it. This may occur if you missed one of the nodes when initially installing the the connector, or if a node has been recovered or added since the connector was installed.

Installing and Configuring Kerberos on Your Vertica Cluster

You must perform several steps to configure your Vertica cluster before the Vertica Connector for HDFS can authenticate an Vertica user using Kerberos.

Note: You only need to perform these configuration steps of you are using the connector with Kerberos. In a non-Kerberos environment, the connector does not require your Vertica cluster to have any special configuration.

Perform the following steps on **each node** in your Vertica cluster:

1. Install the Kerberos libraries and client. To learn how to install these packages, see the documentation your Linux distribution. On some Red Hat and CentOS version, you can install these packages by executing the following commands as root:

```
yum install krb5-libs krb5-workstation
```

On some versions of Debian, you would use the command:

```
apt-get install krb5-config krb5-user krb5-clients
```

2. Update the Kerberos configuration file (`/etc/krb5.conf`) to reflect your site's Kerberos configuration. The easiest method of doing this is to copy the `/etc/krb5.conf` file from your Kerberos Key Distribution Center (KDC) server.

3. Copy the keytab files for the users to a directory on the node (see Preparing Keytab Files for the Vertica Connector for HDFS). The absolute path to these files must be the same on every node in your Vertica cluster.

4. Ensure the keytab files are readable by the database administrator's Linux account (usually dbadmin). The easiest way to do this is to change ownership of the files to dbadmin:

```
sudo chown dbadmin *.keytab
```

Installing and Configuring Kerberos on Your Vertica Cluster

You must perform several steps to configure your Vertica cluster before the Vertica Connector for HDFS can authenticate an Vertica user using Kerberos.

Note: You only need to perform these configuration steps of you are using the connector with Kerberos. In a non-Kerberos environment, the connector does not require your Vertica cluster to have any special configuration.

Perform the following steps on **each node** in your Vertica cluster:

1. Install the Kerberos libraries and client. To learn how to install these packages, see the documentation your Linux distribution. On some Red Hat and CentOS version, you can install these packages by executing the following commands as root:

```
yum install krb5-libs krb5-workstation
```

On some versions of Debian, you would use the command:

```
apt-get install krb5-config krb5-user krb5-clients
```

2. Update the Kerberos configuration file (`/etc/krb5.conf`) to reflect your site's Kerberos configuration. The easiest method of doing this is to copy the `/etc/krb5.conf` file from your Kerberos Key Distribution Center (KDC) server.

3. Copy the keytab files for the users to a directory on the node (see Preparing Keytab Files for the Vertica Connector for HDFS). The absolute path to these files must be the same on every node in your Vertica cluster.

4. Ensure the keytab files are readable by the database administrator's Linux account (usually dbadmin). The easiest way to do this is to change ownership of the files to dbadmin:

   ```
   sudo chown dbadmin *.keytab
   ```

Preparing Keytab Files for the Vertica Connector for HDFS

The Vertica Connector for HDFS uses keytab files to authenticate Vertica users with Kerberos, so they can access files on the Hadoop HDFS filesystem. These files take the place of entering a password at a Kerberos login prompt.

You must have a keytab file for each Vertica user that needs to use the connector. The keytab file must match the Kerberos credentials of a Hadoop user that has access to the HDFS.

Caution: Exporting a keytab file for a user changes the version number associated with the user's Kerberos account. This change **invalidates any previously exported keytab file** for the user. If a keytab file has already been created for a user and is currently in use, you should use that keytab file rather than exporting a new keytab file. Otherwise, exporting a new keytab file will cause any processes using an existing keytab file to no longer be able to authenticate.

To export a keytab file for a user:

1. Start the kadmin utility:

   - If you have access to the root account on the Kerberos Key Distribution Center (KDC) system, log into it and use the `kadmin.local` command. (If this command is not in the system search path, try `/usr/kerberos/sbin/kadmin.local`.)

   - If you do not have access to the root account on the Kerberos KDC, then you can use the `kadmin` command from a Kerberos client system as long as you have the password for the Kerberos administrator account. When you start kadmin, it will prompt you for the Kerberos administrator's password. You may need to have root privileges on the client system in order to run kadmin.

2. Use kadmin's xst (export) command to export the user's credentials to a keytab file:

   ```
   xst -k username.keytab username@YOURDOMAIN.COM
   ```

where *username* is the name of Kerberos principal you want to export, and *YOURDOMAIN.COM* is your site's Kerberos realm. This command creates a keytab file named *username*.keytab in the current directory.

The following example demonstrates exporting a keytab file for a user named exampleuser@MYCOMPANY.COM using the kadmin command on a Kerberos client system:

```
$ sudo kadmin
[sudo] password for dbadmin:
Authenticating as principal root/admin@MYCOMPANY.COM with password.
Password for root/admin@MYCOMPANY.COM:
kadmin:  xst -k exampleuser.keytab exampleuser@MYCOMPANY.COM
Entry for principal exampleuser@MYCOMPANY.COM with kvno 2, encryption
type des3-cbc-sha1 added to keytab WRFILE:exampleuser.keytab.
Entry for principal exampleuser@VERTICACORP.COM with kvno 2, encryption
type des-cbc-crc added to keytab WRFILE:exampleuser.keytab.
```

After exporting the keyfile, you can use the klist command to list the keys stored in the file:

```
$ sudo klist -k exampleuser.keytab
[sudo] password for dbadmin:
Keytab name: FILE:exampleuser.keytab
KVNO Principal
---- ----------------------------------------------------------------
   2 exampleuser@MYCOMPANY.COM
   2 exampleuser@MYCOMPANY.COM
```

Using the HCatalog Connector

The Vertica HCatalog Connector lets you access data stored in Apache's Hive data warehouse software the same way you access it within a native Vertica table.

Hive, HCatalog, and WebHCat Overview

There are several Hadoop components that you need to understand in order to use the HCatalog connector:

- Apache's Hive lets you query data stored in a Hadoop Distributed File System (HDFS) the same way you query data stored in a relational database. Behind the scenes, Hive uses a set of serializer and deserializer (SerDe) classes to extract data from files stored on the HDFS and break it into columns and rows. Each SerDe handles data files in a specific format. For example, one SerDe extracts data from comma-separated data files while another interprets data stored in JSON format.

- Apache HCatalog is a component of the Hadoop ecosystem that makes Hive's metadata available to other Hadoop components (such as Pig).

- WebHCat (formerly known as Templeton) makes HCatalog and Hive data available via a REST web API. Through it, you can make an HTTP request to retrieve data stored in Hive, as well as information about the Hive schema.

Vertica's HCatalog Connector lets you transparently access data that is available through WebHCat. You use the connector to define a schema in Vertica that corresponds to a Hive database or schema. When you query data within this schema, the HCatalog Connector transparently extracts and formats the data from Hadoop into tabular data. The data within this HCatalog schema appears as if it is native to Vertica. You can even perform operations such as joins between Vertica-native tables and HCatalog tables. For more details, see How the HCatalog Connector Works.

HCatalog Connection Features

The HCatalog Connector lets you query data stored in Hive using the Vertica native SQL syntax. Some of its main features are:

- The HCatalog Connector always reflects the current state of data stored in Hive.

- The HCatalog Connector uses the parallel nature of both Vertica and Hadoop to process Hive data. The result is that querying data through the HCatalog Connector is often faster than querying the data directly through Hive.

- Since Vertica performs the extraction and parsing of data, the HCatalog Connector does not signficantly increase the load on your Hadoop cluster.

- The data you query through the HCatalog Connector can be used as if it were native Vertica data. For example, you can execute a query that joins data from a table in an HCatalog schema with a native table.

HCatalog Connection Considerations

There are a few things to keep in mind when using the HCatalog Connector:

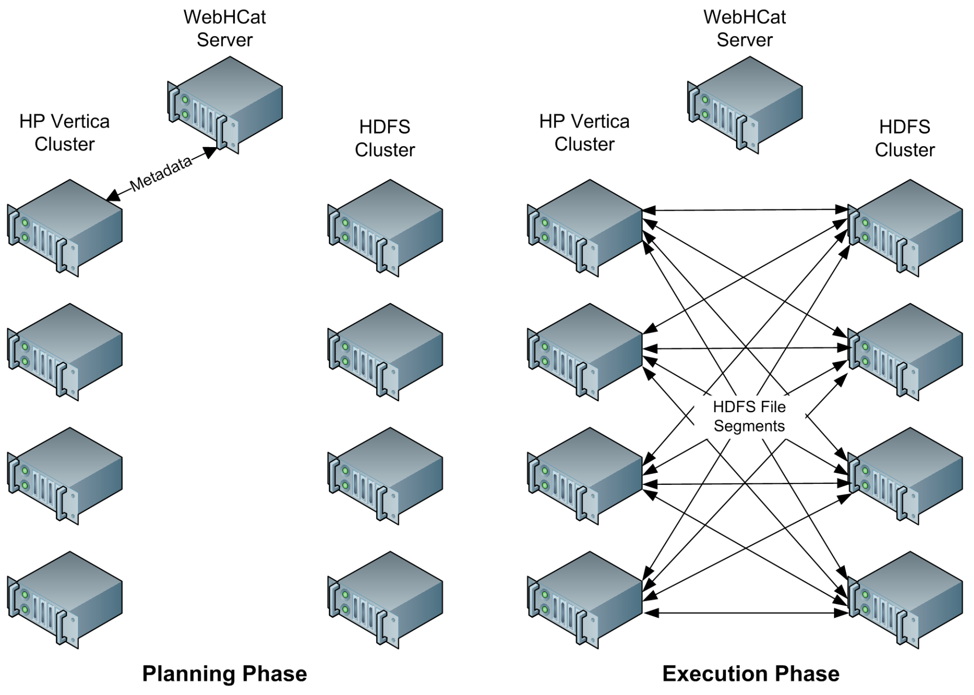
- Hive's data is stored in flat files in a distributed filesystem, requiring it to be read and deserialized each time it is queried. This deserialization causes Hive's performance to be much slower than Vertica. The HCatalog Connector has to perform the same process as Hive to read the data. Therefore, querying data stored in Hive using the HCatalog Connector is much slower than querying a native Vertica table. If you need to perform extensive analysis on data stored in Hive, you should consider loading it into Vertica through the HCatalog Connector or the WebHDFS connector. Vertica optimization often makes querying data through the HCatalog Connector faster than directly querying it through Hive.

- Hive supports complex data types such as lists, maps, and structs that Vertica does not support. Columns containing these data types are converted to a JSON representation of the data type and stored as a VARCHAR. See Data Type Conversions from Hive to Vertica.

Note: The HCatalog Connector is read only. It cannot insert data into Hive.

How the HCatalog Connector Works

When planning a query that accesses data from a Hive table, the Vertica HCatalog Connector on the initiator node contacts the WebHCat server in your Hadoop cluster to determine if the table exists. If it does, the connector retrieves the table's metadata from the metastore database so the query planning can continue. When the query executes, all nodes in the Vertica cluster directly retrieve the data necessary for completing the query from the Hadoop HDFS. They then use the Hive SerDe classes to extract the data so the query can execute.



This approach takes advantage of the parallel nature of both Vertica and Hadoop. In addition, by performing the retrieval and extraction of data directly, the HCatalog Connector reduces the impact of the query on the Hadoop cluster.

HCatalog Connector Requirements

Both your Vertica and Hadoop installations must meet some requirements before you can use the HCatalog Connector.

Vertica Requirements

All of the nodes in your cluster must have a Java Virtual Machine (JVM) installed. See Installing the Java Runtime on Your Vertica Cluster.

Hadoop Requirements

Your Hadoop cluster must meet several requirements in order for it to work with the HCatalog connector:

- It must have Hive and HCatalog installed and running. See Apache's HCatalog page for more information.
- It must have WebHCat (formerly known as Templeton) installed and running. See Apache' s WebHCat page for details.

- The WebHCat server and all of the HDFS nodes that store HCatalog data must be directly accessible from all of the hosts in your Vertica database. You need to ensure that any firewall separating the Hadoop cluster and the Vertica cluster will let WebHCat, metastore database, and HDFS traffic through.
- The data that you want to query must be in an internal or external Hive table.
- If a table you want to query uses a non-standard SerDe, you must install the SerDe's classes on your Vertica cluster before you can query the data. See Using Non-Standard SerDes.

The Vertica Connector for HCatalog has been tested with:

- HCatalog 0.5
- Hive 0.10 and 0.11
- Hadoop 2.0

Testing Connectivity

To test the connection between your database cluster and WebHcat, log into a node in your Vertica cluster and run the following command to execute an HCatalog query:

```
$ curl http://webHCatServer:port/templeton/v1/status?user.name=hcatUsername
```

Where:

- *webHCatServer* is the IP address or hostname of the WebHCat server
- *port* is the port number assigned to the WebHCat service (usually 50111)
- *hcatUsername* is a valid username authorized to use HCatalog

You will usually want to append `;echo` to the command to add a linefeed after the curl command's output. Otherwise, the command prompt is appended to the command's output, making it harder to read.

For example:

```
$ curl http://hcathost:50111/templeton/v1/status?user.name=hive; echo
```

If there are no errors, this command returns a status message in JSON format, similar to the following:

```
{"status":"ok","version":"v1"}
```

This result indicates that WebHCat is running and that the Vertica host can connect to it and retrieve a result. If you do not receive this result, troubleshoot your Hadoop installation and the connectivity between your Hadoop and Vertica clusters.

You can also run some queries to verify that WebHCat is correctly configured to work with Hive. The following example demonstrates listing the databases defined in Hive and the tables defined within a database:

```
$ curl http://hcathost:50111/templeton/v1/ddl/database?user.name=hive; echo
{"databases":["default","production"]}
$ curl http://hcathost:50111/templeton/v1/ddl/database/default/table?user.name=hive; echo
{"tables":["messages","weblogs","tweets","transactions"],"database":"default"}
```

See Apache's WebHCat reference for details about querying Hive using WebHCat.

Installing the Java Runtime on Your Vertica Cluster

The HCatalog Connector requires a 64-bit Java Virtual Machine (JVM). The JVM must support Java 6 or later.

Note: If your Vertica cluster is configured to execute User Defined Extensions (UDxs) written in Java, it already has a correctly-configured JVM installed. See Developing User Defined Functions in Java in the Programmer's Guide for more information.

Installing Java on your Vertica cluster is a two-step process:

1. Install a Java runtime on all of the hosts in your cluster.
2. Set the JavaBinaryForUDx configuration parameter to tell Vertica the location of the Java executable.

Installing a Java Runtime

For Java-based features, Vertica requires a 64-bit Java 6 (Java version 1.6) or later Java runtime. Vertica supports runtimes from either Oracle or OpenJDK. You can choose to install either the Java Runtime Environment (JRE) or Java Development Kit (JDK), since the JDK also includes the JRE.

Many Linux distributions include a package for the OpenJDK runtime. See your Linux distribution's documentation for information about installing and configuring OpenJDK.

To install the Oracle Java runtime, see the Java Standard Edition (SE) Download Page. You usually run the installation package as root in order to install it. See the download page for instructions.

Once you have installed a JVM on each host, ensure that the `java` command is in the search path and calls the correct JVM by running the command:

```
$ java -version
```

This command should print something similar to:

```
java version "1.6.0_37"Java(TM) SE Runtime Environment (build 1.6.0_37-b06)
Java HotSpot(TM) 64-Bit Server VM (build 20.12-b01, mixed mode)
```

Note: Any previously installed Java VM on your hosts may interfere with a newly installed Java runtime. See your Linux distribution's documentation for instructions on configuring which JVM is the default. Unless absolutely required, you should uninstall any incompatible version of Java before installing the Java 6 or Java 7 runtime.

Setting the JavaBinaryForUDx Configuration Parameter

The JavaBinaryForUDx configuration parameter tells Vertica where to look for the JRE to execute Java UDxs. After you have installed the JRE on all of the nodes in your cluster, set this parameter to the absolute path of the Java executable. You can use the symbolic link that some Java installers create (for example `/usr/bin/java`). If the Java executable is in your shell search path, you can get the path of the Java executable by running the following command from the Linux command line shell:

```
$ which java
/usr/bin/java
```

If the `java` command is not in the shell search path, use the path to the Java executable in the directory where you installed the JRE. Suppose you installed the JRE in `/usr/java/default` (which is where the installation package supplied by Oracle installs the Java 1.6 JRE). In this case the Java executable is `/usr/java/default/bin/java`.

You set the configuration parameter by executing the following statement as a database superuser:

```
=> SELECT SET_CONFIG_PARAMETER('JavaBinaryForUDx','/usr/bin/java');
```

See SET_CONFIG_PARAMETER in the SQL Reference Manual for more information on setting configuration parameters.

To view the current setting of the configuration parameter, query the CONFIGURATION_PARAMETERS system table:

```
=> \x
Expanded display is on.
=> SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'JavaBinaryForUDx';
-[ RECORD 1 ]-----------------+----------------------------------------------------------
node_name                     | ALL
parameter_name                | JavaBinaryForUDx
current_value                 | /usr/bin/java
default_value                 |
change_under_support_guidance | f
change_requires_restart       | f
description                   | Path to the java binary for executing UDx written in Java
```

Once you have set the configuration parameter, Vertica can find the Java executable on each node in your cluster.

Note: Since the location of the Java executable is set by a single configuration parameter for the entire cluster, you must ensure that the Java executable is installed in the same path on all of the hosts in the cluster.

Defining a Schema Using the HCatalog Connector

After you set up the HCatalog Connector, you can use it to define a schema in your Vertica database to access the tables in a Hive database. You define the schema using the CREATE HCATALOG SCHEMA statement. See CREATE HCATALOG SCHEMA in the SQL Reference Manual for a full description.

When creating the schema, you must supply at least two pieces of information:

- the name of the schema to define in Vertica
- the host name or IP address of Hive's metastore database (the database server that contains metadata about Hive's data, such as the schema and table definitions)

Other parameters are optional. If you do not supply a value, Vertica uses default values.

After you define the schema, you can query the data in the Hive data warehouse in the same way you query a native Vertica table. The following example demonstrates creating an HCatalog schema and then querying several system tables to examine the contents of the new schema. See Viewing Hive Schema and Table Metadata for more information about these tables.

```
=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost' HCATALOG_SCHEMA='default'
-> HCATALOG_USER='hcatuser';
CREATE SCHEMA
=> -- Show list of all HCatalog schemas
=> \x
Expanded display is on.
=> SELECT * FROM v_catalog.hcatalog_schemata;
-[ RECORD 1 ]--------+------------------------------
schema_id            | 45035996273748980
schema_name          | hcat
schema_owner_id      | 45035996273704962
schema_owner         | dbadmin
create_time          | 2013-11-04 15:09:03.504094-05
hostname             | hcathost
port                 | 9933
webservice_hostname  | hcathost
webservice_port      | 50111
hcatalog_schema_name | default
hcatalog_user_name   | hcatuser
metastore_db_name    | hivemetastoredb
```

```
=> -- List the tables in all HCatalog schemas
=> SELECT * FROM v_catalog.hcatalog_table_list;
-[ RECORD 1 ]------+------------------
table_schema_id    | 45035996273748980
table_schema       | hcat
hcatalog_schema    | default
table_name         | messages
hcatalog_user_name | hcatuser
-[ RECORD 2 ]------+------------------
table_schema_id    | 45035996273748980
table_schema       | hcat
hcatalog_schema    | default
table_name         | weblog
hcatalog_user_name | hcatuser
-[ RECORD 3 ]------+------------------
table_schema_id    | 45035996273748980
table_schema       | hcat
hcatalog_schema    | default
table_name         | tweets
hcatalog_user_name | hcatuser
```

Querying Hive Tables Using HCatalog Connector

Once you have defined the HCatalog schema, you can query data from the Hive database by using the schema name in your query.

```
=> SELECT * from hcat.messages limit 10;
 messageid |   userid   |        time         |             message
-----------+------------+---------------------+----------------------------------
         1 | nPfQ1ayhi  | 2013-10-29 00:10:43 | hymenaeos cursus lorem Suspendis
         2 | N7svORIoZ  | 2013-10-29 00:21:27 | Fusce ad sem vehicula morbi
         3 | 4VvzN3d    | 2013-10-29 00:32:11 | porta Vivamus condimentum
         4 | heojkmTmc  | 2013-10-29 00:42:55 | lectus quis imperdiet
         5 | coROws3OF  | 2013-10-29 00:53:39 | sit eleifend tempus a aliquam mauri
         6 | oDRP1i     | 2013-10-29 01:04:23 | risus facilisis sollicitudin sceler
         7 | AU7a9Kp    | 2013-10-29 01:15:07 | turpis vehicula tortor
         8 | ZJWg185DkZ | 2013-10-29 01:25:51 | sapien adipiscing eget Aliquam tor
         9 | E7ipAsYC3  | 2013-10-29 01:36:35 | varius Cum iaculis metus
        10 | kStCv      | 2013-10-29 01:47:19 | aliquam libero nascetur Cum mal
(10 rows)
```

Since the tables you access through the HCatalog Connector act like Vertica tables, you can perform operations that use both Hive data and native Vertica data, such as a join:

```
=> SELECT u.FirstName, u.LastName, d.time, d.Message from UserData u
-> JOIN hcat.messages d ON u.UserID = d.UserID LIMIT 10;
FirstName | LastName |        time         |                 Message
----------+----------+---------------------+-----------------------------------
Whitney   | Kerr     | 2013-10-29 00:10:43 | hymenaeos cursus lorem Suspendis
Troy      | Oneal    | 2013-10-29 00:32:11 | porta Vivamus condimentum
```

```
Renee     | Coleman  | 2013-10-29 00:42:55 | lectus quis imperdiet
Fay       | Moss     | 2013-10-29 00:53:39 | sit eleifend tempus a aliquam mauri
Dominique | Cabrera  | 2013-10-29 01:15:07 | turpis vehicula tortor
Mohammad  | Eaton    | 2013-10-29 00:21:27 | Fusce ad sem vehicula morbi
Cade      | Barr     | 2013-10-29 01:25:51 | sapien adipiscing eget Aliquam tor
Oprah     | Mcmillan | 2013-10-29 01:36:35 | varius Cum iaculis metus
Astra     | Sherman  | 2013-10-29 01:58:03 | dignissim odio Pellentesque primis
Chelsea   | Malone   | 2013-10-29 02:08:47 | pede tempor dignissim Sed luctus
(10 rows)
```

Viewing Hive Schema and Table Metadata

When using Hive, you access metadata about schemata and tables by executing statements written in HiveQL (Hive's version of SQL) such as `SHOW TABLES`. When using the HCatalog Connector, you can get metadata about the tables in the Hive database through several Vertica system tables.

There are four system tables that contain metadata about the tables accessible through the HCatalog Connector:

- HCATALOG_SCHEMATA lists all of the schemata (plural of schema) that have been defined using the HCatalog Connector. See HCATALOG_SCHEMATA in the SQL Reference Manual for detailed information.

- HCATALOG_TABLE_LIST contains an overview of all of the tables available from all schemata defined using the HCatalog Connector. This table only shows the tables which the user querying the table can access. The information in this table is retrieved using a single call to WebHCat for each schema defined using the HCatalog Connector, which means there is a little overhead when querying this table. See HCATALOG_TABLE_LIST in the SQL Reference Manual for detailed information.

- HCATALOG_TABLES contains more in-depth information than HCATALOG_TABLE_LIST. However, querying this table results in Vertica making a REST web service call to WebHCat for each table available through the HCatalog Connector. If there are many tables in the HCatalog schemata, this query could take a while to complete. See HCATALOG_TABLES in the SQL Reference Manual for more information.

- HCATALOG_COLUMNS lists metadata about all of the columns in all of the tables available through the HCatalog Connector. Similarly to HCATALOG_TABLES, querying this table results in one call to WebHCat per table, and therefore can take a while to complete. See HCATALOG_COLUMNS in the SQL Reference Manual for more information.

The following example demonstrates querying the system tables containing metadata for the tables available through the HCatalog Connector.

```
=> CREATE HCATALOG SCHEMA hcat WITH hostname='hcathost'
-> HCATALOG_SCHEMA='default' HCATALOG_DB='default' HCATALOG_USER='hcatuser';
CREATE SCHEMA
=> SELECT * FROM HCATALOG_SCHEMATA;
```

```
-[ RECORD 1 ]--------+-----------------------------
schema_id            | 45035996273864536
schema_name          | hcat
schema_owner_id      | 45035996273704962
schema_owner         | dbadmin
create_time          | 2013-11-05 10:19:54.70965-05
hostname             | hcathost
port                 | 9083
webservice_hostname  | hcathost
webservice_port      | 50111
hcatalog_schema_name | default
hcatalog_user_name   | hcatuser
metastore_db_name    | hivemetastoredb

=> SELECT * FROM HCATALOG_TABLE_LIST;
-[ RECORD 1 ]------+------------------
table_schema_id    | 45035996273864536
table_schema       | hcat
hcatalog_schema    | default
table_name         | hcatalogtypes
hcatalog_user_name | hcatuser
-[ RECORD 2 ]------+------------------
table_schema_id    | 45035996273864536
table_schema       | hcat
hcatalog_schema    | default
table_name         | tweets
hcatalog_user_name | hcatuser
-[ RECORD 3 ]------+------------------
table_schema_id    | 45035996273864536
table_schema       | hcat
hcatalog_schema    | default
table_name         | messages
hcatalog_user_name | hcatuser
-[ RECORD 4 ]------+------------------
table_schema_id    | 45035996273864536
table_schema       | hcat
hcatalog_schema    | default
table_name         | msgjson
hcatalog_user_name | hcatuser

=> -- Get detailed description of a specific table
=> SELECT * FROM HCATALOG_TABLES WHERE table_name = 'msgjson';
-[ RECORD 1 ]---------+---------------------------------------------------------
table_schema_id       | 45035996273864536
table_schema          | hcat
hcatalog_schema       | default
table_name            | msgjson
hcatalog_user_name    | hcatuser
min_file_size_bytes   | 13524
total_number_files    | 10
location              | hdfs://hive.example.com:8020/user/exampleuser/msgjson
last_update_time      | 2013-11-05 14:18:07.625-05
output_format         | org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
last_access_time      | 2013-11-11 13:21:33.741-05
max_file_size_bytes   | 45762
is_partitioned        | f
partition_expression  |
table_owner           | hcatuser
```

```
input_format          | org.apache.hadoop.mapred.TextInputFormat
total_file_size_bytes | 453534
hcatalog_group        | supergroup
permission            | rwxr-xr-x

=> -- Get list of columns in a specific table
=> SELECT * FROM HCATALOG_COLUMNS WHERE table_name = 'hcatalogtypes'
-> ORDER BY ordinal_position;
-[ RECORD 1 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | intcol
hcatalog_data_type       | int
data_type                | int
data_type_id             | 6
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 1
-[ RECORD 2 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | floatcol
hcatalog_data_type       | float
data_type                | float
data_type_id             | 7
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 2
-[ RECORD 3 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | doublecol
hcatalog_data_type       | double
data_type                | float
data_type_id             | 7
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 3
-[ RECORD 4 ]------------+-----------------
table_schema             | hcat
```

```
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | charcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 4
-[ RECORD 5 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | varcharcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 5
-[ RECORD 6 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | boolcol
hcatalog_data_type       | boolean
data_type                | boolean
data_type_id             | 5
data_type_length         | 1
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 6
-[ RECORD 7 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | timestampcol
hcatalog_data_type       | string
data_type                | varchar(65000)
data_type_id             | 9
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 7
-[ RECORD 8 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | varbincol
hcatalog_data_type       | binary
data_type                | varbinary(65000)
data_type_id             | 17
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 8
-[ RECORD 9 ]------------+-----------------
table_schema             | hcat
hcatalog_schema          | default
table_name               | hcatalogtypes
is_partition_column      | f
column_name              | bincol
hcatalog_data_type       | binary
data_type                | varbinary(65000)
data_type_id             | 17
data_type_length         | 65000
character_maximum_length | 65000
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 9
```

Synching an HCatalog Schema With a Local Schema

Querying data from an HCatalog schema can be slow due to Hive and WebHCat performance issues. This slow performance can be especially annoying when you use the HCatalog Connector to query the HCatalog schema's metadata to examine the structure of the tables in the Hive database.

To avoid this problem you can use the use the SYNC_WITH_HCATALOG_SCHEMA function to create snapshot of the HCatalog schema's metadata within an Vertica schema. You supply this function with the name of a pre-existing Vertica schema and an HCatalog schema available through the HCatalog Connector. It creates a set of external tables within the Vertica schema that you can then use to examine the structure of the tables in the Hive database. Since the data in the Vertica schema is local, queries are much faster. You can also use standard Vertica statements and system tables queries to examine the structure of Hive tables in the HCatalog schema.

Caution: The SYNC_WITH_HCATALOG_SCHEMA overwrites tables in the Vertica schema whose names match a table in the HCatalog schema. To avoid losing data, always create an empty Vertica schema to sync with an HCatalog schema.

The Vertica schema is just a snapshot of the HCatalog schema's metadata. Vertica does not synchronize later changes to the HCatalog schema with the local schema after you call SYNC_WITH_HCATALOG_SCHEMA. You can call the function again to re-synchronize the local schema to the HCatalog schema.

Note: By default, the function does not drop tables that appear in the local schema that do not appear in the HCatalog schema. Thus after the function call the local schema does not reflect tables that have been dropped in the Hive database. You can change this behavior by supplying the optional third Boolean argument that tells the function to drop any table in the local schema that does not correspond to a table in the HCatalog schema.

The following example demonstrates calling SYNC_WITH_HCATALOG_SCHEMA to sync the HCatalog schema named hcat with a local schema.

```
=> CREATE SCHEMA hcat_local;
CREATE SCHEMA
=> SELECT sync_with_hcatalog_schema('hcat_local', 'hcat');
sync_with_hcatalog_schema
----------------------------------------
Schema hcat_local synchronized with hcat
tables in hcat = 56
tables altered in hcat_local = 0
tables created in hcat_local = 56
stale tables in hcat_local = 0
table changes erred in hcat_local = 0
(1 row)

=> -- Use vsql's \d command to describe a table in the synced schema

=> \d hcat_local.messages
List of Fields by Tables
  Schema   |   Table  | Column  |      Type      | Size  | Default | Not Null | Primary
Key | Foreign Key
-----------+----------+---------+----------------+-------+---------+----------+----------
---+-------------
hcat_local | messages | id      | int            |     8 |         | f        | f
  |
hcat_local | messages | userid  | varchar(65000) | 65000 |         | f        | f
  |
hcat_local | messages | "time"  | varchar(65000) | 65000 |         | f        | f
  |
hcat_local | messages | message | varchar(65000) | 65000 |         | f        | f
  |
(4 rows)
```

Note: You can query tables in the local schema you synched with an HCatalog schema. Querying tables in a synched schema isn't much faster than directly querying the HCatalog schema because SYNC_WITH_HCATALOG_SCHEMA only duplicates the HCatalog schema's metadata. The data in the table is still retrieved using the HCatalog Connector,

Data Type Conversions from Hive to Vertica

The data types recognized by Hive differ from the data types recognize by Vertica. The following table lists how the HCatalog Connector converts Hive data types into data types compatible with Vertica.

Hive Data Type	Vertica Data Type
TINYINT (1-byte)	TINYINT (8-bytes)
SMALLINT (2-bytes)	SMALLINT (8-bytes)
INT (4-bytes)	INT (8-bytes)
BIGINT (8-bytes)	BIGINT (8-bytes)
BOOLEAN	BOOLEAN
FLOAT (4-bytes)	FLOAT (8-bytes)
DOUBLE (8-bytes)	DOUBLE PRECISION (8-bytes)
STRING (2 GB max)	VARCHAR (65000)
BINARY (2 GB max)	VARBINARY (65000)
LIST/ARRAY	VARCHAR (65000) containing a JSON-format representation of the list.
MAP	VARCHAR (65000) containing a JSON-format representation of the map.
STRUCT	VARCHAR (65000) containing a JSON-format representation of the struct.

Data-Width Handling Differences Between Hive and Vertica

The HCatalog Connector relies on Hive SerDe classes to extract data from files on HDFS. Therefore, the data read from these files are subject to Hive's data width restrictions. For example, suppose the SerDe parses a value for an INT column into a value that is greater than 2^{32}-1 (the maximum value for a 32-bit integer). In this case, the value is rejected even if it would fit into an Vertica's 64-bit INTEGER column because it cannot fit into Hive's 32-bit INT.

Once the value has been parsed and converted to an Vertica data type, it is treated as an native data. This treatment can result in some confusion when comparing the results of an identical query run in Hive and in Vertica. For example, if your query adds two INT values that result in a value that is larger than 2^{32}-1, the value overflows its 32-bit INT data type, causing Hive to return an error. When running the same query with the same data in Vertica using the HCatalog Connector, the value will probably still fit within Vertica's 64-int value. Thus the addition is successful and returns a value.

Using Non-Standard SerDes

Hive stores its data in unstructured flat files located in the Hadoop Distributed File System (HDFS). When you execute a Hive query, it uses a set of serializer and deserializer (SerDe) classes to extract data from these flat files and organize it into a relational database table. For Hive to be able to extract data from a file, it must have a SerDe that can parse the data the file contains. When you create a table in Hive, you can select the SerDe to be used for the table's data.

Hive has a set of standard SerDes that handle data in several formats such as delimited data and data extracted using regular expressions. You can also use third-party or custom-defined SerDes that allow Hive to process data stored in other file formats. For example, some commonly-used third-party SerDes handle data stored in JSON format.

The HCatalog Connector directly fetches file segments from HDFS and uses Hive's SerDes classes to extract data from them. The Connector includes all Hive's standard SerDes classes, so it can process data stored in any file that Hive natively supports. If you want to query data from a Hive table that uses a custom SerDe, you must first install the SerDe classes on the Vertica cluster.

Determining Which SerDe You Need

If you have access to the Hive command line, you can determine which SerDe a table uses by using Hive's SHOW CREATE TABLE statement. This statement shows the HiveQL statement needed to recreate the table. For example:

```
hive> SHOW CREATE TABLE msgjson;
OK
CREATE EXTERNAL TABLE msgjson(
messageid int COMMENT 'from deserializer',
userid string COMMENT 'from deserializer',
time string COMMENT 'from deserializer',
message string COMMENT 'from deserializer')
ROW FORMAT SERDE
'org.apache.hadoop.hive.contrib.serde2.JsonSerde'
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
'hdfs://hivehost.example.com:8020/user/exampleuser/msgjson'
TBLPROPERTIES (
'transient_lastDdlTime'='1384194521')
Time taken: 0.167 seconds
```

In the example, `ROW FORMAT SERDE` indicates that a special SerDe is used to parse the data files. The next row shows that the class for the SerDe is named `org.apache.hadoop.hive.contrib.serde2.JsonSerde`.You must provide the HCatalog Connector with a copy of this SerDe class so that it can read the data from this table.

You can also find out which SerDe class you need by querying the table that uses the custom SerDe. The query will fail with an error message that contains the class name of the SerDe needed to parse the data in the table. In the following example, the portion of the error message that names the missing SerDe class is in bold.

```
=> SELECT * FROM hcat.jsontable;
ERROR 3399:  Failure in UDx RPC call InvokePlanUDL(): Error in User Defined
Object [VHCatSource], error code: 0
com.vertica.sdk.UdfException: Error message is [
org.apache.hcatalog.common.HCatException : 2004 : HCatOutputFormat not
initialized, setOutput has to be called. Cause : java.io.IOException:
java.lang.RuntimeException:
MetaException(message:org.apache.hadoop.hive.serde2.SerDeException
SerDe com.cloudera.hive.serde.JSONSerDe does not exist) ] HINT If error
message is not descriptive or local, may be we cannot read metadata from hive
metastore service thrift://hcathost:9083 or HDFS namenode (check
UDxLogs/UDxFencedProcessesJava.log in the catalog directory for more information)
at com.vertica.hcatalogudl.HCatalogSplitsNoOpSourceFactory
.plan(HCatalogSplitsNoOpSourceFactory.java:98)
at com.vertica.udxfence.UDxExecContext.planUDSource(UDxExecContext.java:898)
. . .
```

Installing the SerDe on the Vertica Cluster

You usually have two options to getting the SerDe class file the HCatalog Connector needs:

- Find the installation files for the SerDe, then copy those over to your Vertica cluster. For example, there are several third-party JSON SerDes available from sites like Google Code and GitHub. You may find the one that matches the file installed on your Hive cluster. If so, then download the package and copy it to your Vertica cluster.

- Directly copy the JAR files from a Hive server onto your Vertica cluster. The location for the SerDe JAR files depends on your Hive installation. On some systems, they may be located in `/usr/lib/hive/lib`.

Wherever you get the files, copy them into the `/opt/vertica/packages/hcat/lib` directory on every node in your Vertica cluster.

Important: If you add a new host to your Vertica cluster, remember to copy every custom SerDer JAR file to it.

Troubleshooting HCatalog Connector Problems

You may encounter the following issues when using the HCatalog Connector.

Connection Errors

When you use CREATE HCATALOG SCHEMA to create a new schema, the HCatalog Connector does not actually attempt to connect to the WebHCat or metastore servers. When you execute a query using the schema or HCatalog-related system tables, the connector attempts to connect to and retrieve data from your Hadoop cluster.

The types of errors you get will depend on which parameters are incorrect. Suppose you have incorrect parameters for the metastore database, but correct parameters for WebHCat. In this case, HCatalog-related system table queries succeed, while queries on the HCatalog schema fail. The following example demonstrates creating an HCatalog schema with the correct default WebHCat information but the incorrect port number for the metastore database.

```
=> CREATE HCATALOG SCHEMA hcat2 WITH hostname='hcathost'
-> HCATALOG_SCHEMA='default' HCATALOG_USER='hive' PORT=1234;
CREATE SCHEMA
=> SELECT * FROM HCATALOG_TABLE_LIST;
-[ RECORD 1 ]------+---------------------
table_schema_id    | 45035996273864536
table_schema       | hcat2
hcatalog_schema    | default
table_name         | test
hcatalog_user_name | hive

=> SELECT * FROM hcat2.test;
ERROR 3399:  Failure in UDx RPC call InvokePlanUDL(): Error in User Defined
 Object [VHCatSource], error code: 0
com.vertica.sdk.UdfException: Error message is [
org.apache.hcatalog.common.HCatException : 2004 : HCatOutputFormat not
initialized, setOutput has to be called. Cause : java.io.IOException:
MetaException(message:Could not connect to meta store using any of the URIs
provided. Most recent failure: org.apache.thrift.transport.TTransportException:
java.net.ConnectException:
Connection refused
at org.apache.thrift.transport.TSocket.open(TSocket.java:185)
at org.apache.hadoop.hive.metastore.HiveMetaStoreClient.open(
HiveMetaStoreClient.java:277)
. . .
```

To resolve these issues, you must drop the schema and recreate it with the correct parameters. If you still have issues, determine whether there are connectivity issues between your Vertica cluster and your Hadoop cluster. Such issues can include a firewall that prevents one or more Vertica hosts from contacting the WebHCat, metastore, or HDFS hosts.

SerDe Errors

If you attempt to query a table Hive table that uses a non-standard SerDe, and you have not installed the SerDe JAR files on your Vertica cluster, you will receive an error similar to the following example:

```
=> SELECT * FROM hcat.jsontable;
ERROR 3399:  Failure in UDx RPC call InvokePlanUDL(): Error in User Defined
Object [VHCatSource], error code: 0
com.vertica.sdk.UdfException: Error message is [
org.apache.hcatalog.common.HCatException : 2004 : HCatOutputFormat not
initialized, setOutput has to be called. Cause : java.io.IOException:
java.lang.RuntimeException:
MetaException(message:org.apache.hadoop.hive.serde2.SerDeException
SerDe com.cloudera.hive.serde.JSONSerDe does not exist) ] HINT If error
message is not descriptive or local, may be we cannot read metadata from hive
metastore service thrift://hcathost:9083 or HDFS namenode (check
UDxLogs/UDxFencedProcessesJava.log in the catalog directory for more information)
at com.vertica.hcatalogudl.HCatalogSplitsNoOpSourceFactory
.plan(HCatalogSplitsNoOpSourceFactory.java:98)
at com.vertica.udxfence.UDxExecContext.planUDSource(UDxExecContext.java:898)
. . .
```

In the error message, you can see that the root cause is a missing SerDe class (the section of the error message shown in bold). To resolve this issue, you need to install the SerDe class on your Vertica cluster. See Using Non-Standard SerDes for more information.

This error may occur intermittently if just one or a few hosts in your cluster do not have the SerDe class.

Differing Results Between Hive and Vertica Queries

You may find that running the same query on Hive and on Vertica through the HCatalog Connector can return different results. This discrepancy is often caused by the differences between the data types supported by Hive and Vertica. See Data Type Conversions from Hive to Vertica for more information.

Preventing Excessive Query Delays

Network issues or high system loads on the WebHCat server can cause long delays while querying a Hive database using the HCatalog Connector. While Vertica cannot resolve these issues, you can set parameters that limit how long it waits before canceling a query on an HCatalog schema. These parameters can be set globally via Vertica configuration parameters. You can also set them for specific HCatalog schemas in the CREATE HCATALOG SCHEMA statement. These specific settings override the settings in the configuration parameters.

The HCatConnectionTimeout configuration parameter and the CREATE HCATALOG SCHEMA statement's HCATALOG_CONNECTION_TIMEOUT parameter control how many seconds the HCatalog Connector waits for a connection to the WebHCat server. A value of 0 (the default setting for the configuration parameter) means to wait indefinitely. If the WebHCat server does not respond by the time this timeout elapses, the HCatalog Connector breaks the connection and cancels the query. If you find that some queries on an HCatalog schema pause execessively, try setting this parameter to a timeout value, so the query does hang indefinitely.

The other two parameters work together to terminate HCatalog queries that are running too slowly. After it successfully connects to the WebHCat server and requests data, the HCatalog Connector waits for the number of seconds set in the HCatSlowTransferTime configuration parameter. This value can also be set using the CREATE HCATALOG SCHEMA statement's HCATALOG_SLOW_TRANSFER_TIME parameter. After this time has elapsed, if the data transfer rate from the WebHCat server is not at least the value set in the HCatSlowTransferLimit configuration parameter (or by the CREATE HCATALOG SCHEMA statement's HCATALOG_SLOW_TRANSFER_LIMIT parameter) the HCatalog Connector terminates the connection and cancels the query. You can set this parameter to cancel queries that run very slowly but do eventually complete.

Note: Query delays are usually caused by a slow connection rather than a problem establishing the connection. Therefore, try adjusting the slow transfer rate settings first. If you find the cause of the issue is connections that never complete, you can adjust the Linux TCP socket timeouts to a suitable value instead of relying on the HCatConnectionTimeout parameter.

Integrating Vertica with the MapR Distribution of Hadoop

MapR is a distribution of Apache Hadoop produced by MapR Technologies that extends the standard Hadoop components with its own features. By adding Vertica to a MapR cluster, you can benefit from the advantages of both Vertica and Hadoop. To learn more about integrating Vertica and MapR, see Configuring HP Vertica Analytics Platform with MapR,which appears on the MapR website.

We appreciate your feedback!

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on Hadoop Integration Guide (Vertica Analytic Database 7.0.x)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to vertica-docfeedback@microfocus.com.