Pivotal

PRODUCT DOCUMENTATION

Pivotal Extension Framework

Version 2.2

Installation and User Guide

Rev: A02 – April 30, 2014

Copyright

Use of Open Source

This product may be distributed with open source code, licensed to you in accordance with the applicable open source license. If you would like a copy of any such source code, Pivotal will provide a copy of the source code that is required to be made available in accordance with the applicable open source license. Pivotal may charge reasonable shipping and handling charges for such distribution.

About Pivotal Software, Inc.

Greenplum transitioned to a new corporate identity (Pivotal, Inc.) in 2013. As a result of this transition, there will be some legacy instances of our former corporate identity (Greenplum) appearing in our products and documentation. If you have any questions or concerns, please do not hesitate to contact us through our web site: http://gopivotal.com/about-pivotal/support.

Contents

Chapter 1 PXF Installation and Administration

PXF is an extensible framework that allows HAWQ to query external system data. PXF includes built-in connectors for accessing data that exists inside HDFS files, Hive tables, and HBase tables. Users can also create their own connectors to other parallel data stores or processing engines. To create these connectors using JAVA plugins, see PXF External Table and API Reference.

Topics:

Prerequisites

Check that the following systems are installed and running before you install PXF:

- HAWQ
- Pivotal Hadoop (PHD)
- Hadoop File System (HDFS) with REST service enabled on the Namenode and all the Datanodes.
- Hive: Check that the Hive Metastore service is available and running. This is especially important on clusters where you have set the property *hive.matastore.uri* in the *hive-site.xml* file on the Namenode to point to that cluster.
- HBase
- When configuring Secure (Kerberized) HDFS, NameNode port must be 8020 for PXF to function. This limitation will be removed in the next release.

Upgrading from GPXF

If you have a previous version of HAWQ, or GPXF installed, see the Pivotal ADS 1.2.0.0 Release Notes, for instructions about how to upgrade to HAWQ 1.2.x.

Please note the following:

1. DROP the tables created using GPXF and CREATE them again using PXF.
2. When you CREATE tables for PXF, remember to perform the following:
    a. Change the protocol name in the LOCATION clause from gpxf to pxf.
    b. Ensure that a Profile or Fragmenter, Accessor, Resolver are *always* specified for the table.
    c. If not using a Profile, Check that you have the new names for the Fragmenter, Accessor, Resolver classes. See PXF External Table and API Reference for more information about the API.
3. Check that you are using the correct gucs for PXF:

```
gpxf_enable_filter_pushdown -> pxf_enable_filter_pushdown
gpxf_enable_stat_collection -> pxf_enable_stat_collection
gpxf_enable_locality_optimizations -> pxf_enable_locality_optimizations
```

Installing PXF

Note

This topic describes how you can install PXF as a separate component, if you did not install it using the Pivotal Hadoop (HD) Enterprise Command Line Interface.

This topic contains the following information:

- Installing the PXF JARs File
- Setting up the Java Classpath
- Enabling REST service
- Restarting the Hadoop Cluster
- Testing PXF with HDFS Files
- Installing PXF to Work with Hive
- Testing Hive and PXF Integration

The following steps are required to install and configure PXF on a PHD 2.0.0 cluster.

Notes

- HBase steps are only required for PXF using HBase.
- Hive steps are only required for PXF for Hive.
- All steps must be performed on all nodes, unless noted differently.

Installing the PXF Framework JARs Files

Install the PXF Framework JARs files on all nodes in the cluster:

```
sudo rpm -i pxf-core-2.2.0-x.rpm
sudo rpm -i pxf-api-2.2.0-x.rpm
```

- PXF RPMs reside in the Pivotal ADS/HAWQ stack file.

- The script installs the JARs files at the default location at */usr/lib/gphd/pxf-2.2.0*. Two S*oftlinks:* pxf-core.jar and pxf-api.jar will be created in */usr/lib/gphd/pxf*

Setting up the Java Classpath

Append the following lines to the */etc/gphd/hadoop/conf/hadoop-env.sh* on all nodes in the cluster:

```
export GPHD_ROOT=/usr/lib/gphd
export HADOOP_CLASSPATH=\
$GPHD_ROOT/pxf/pxf-core.jar:\
$GPHD_ROOT/pxf/pxf-api.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/zookeeper/zookeeper-3.4.5-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-common-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-protocol-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-client-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-thrift-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/htrace-core-2.01.jar:\
/etc/gphd/hbase/conf:\
$GPHD_ROOT/hive/lib/hive-service-0.12.0-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hive/lib/hive-metastore-0.12.0-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hive/lib/hive-common-0.12.0-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hive/lib/hive-exec-0.12.0-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hive/lib/libfb303-0.9.0.jar:\
$GPHD_ROOT/hive/lib/libthrift-0.9.0.jar:\

/etc/gphd/hive/conf:\
```

This adds the following to the PHD 2.0.0 Java classpath:

- PXF JARs, and staging dir */usr/lib/gphd/publicstage* (see *"About the Public Directory"* below).
- HBase: hbase-common, hbase-protocol, hbase-client, hbase-thrift, htrace-core, configuration dir
- Zookeeper jar.
- Hive: hive-service, libthrift, hive-metastore, libfb303, hive-common, hive-exec.

Enabling the REST Service

To enable REST service, edit the */etc/gphd/hadoop/conf/hdfs-site.xml* files on all the nodes:

```
<property>
    <name>dfs.webhdfs.enabled</name>
    <value>true</value>
</property>
```

Restarting the Cluster

1. From the Admin node, use the Pivotal Hadoop (HD) Manager Command Line Interface to start the cluster.
2. Optionally, you can examine the classpath with the following command:

```
massh /tmp/clientnodes verbose "hadoop classpath"
```

Secure PXF

PXF can be used on a secure HDFS cluster. Read, write and analyze of PXF tables on HDFS files are enabled. No changes are required to PXF tables that pre-exist from a previous version.

Requirements

- Both HDFS and YARN principals are created and are properly configured.
- HDFS uses port 8020 (see limitations section).
- Hawq is configured to work in secure mode properly, according to the instructions in the Hawq guide.

Limitations

- The HDFS Namenode port must be 8020. This is a limitation that will be fixed in the next PXF version.
- HBase/Hive setups that require authentication are currently not supported.

Common Errors

Please refer to the troubleshooting section for common errors related to PXF security, and their meaning.

Reading and Writing Data with PXF

PXF comes with a number of built-in connectors for reading data that exists inside HDFS files, Hive tables, and HBase tables and for writing data into HDFS files. These built-in connectors use the PXF extensible API. You can also use the extensible API to create your own connectors to any other type of parallel data store or processing engine. See PXF External Table and API Reference for more information about the API.

This topic contains the following information:

- Accessing HDFS File Data with PXF (Read + Write)
- Accessing HIVE Data with PXF (Read only)
- Accessing HBase Data with PXF (Read only)
- Accessing GemFireXD Data with PXF (Read only)

Built-in Profiles

A profile is a collection of common metadata attributes. Use the convenient and simplified PXF syntax.

PXF comes with a number of built-in profiles which group together a collection of metadata attributes to achieve a common goal:

Profile	Description	Fragmenter/Accessor/Resolver
HdfsTextSimple	Read or write delimited single line records from or to plain text files on HDFS.	• com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter • com.pivotal.pxf.plugins.hdfs.LineBreakAccessor • com.pivotal.pxf.plugins.hdfs.StringPassResolver
HdfsTextMulti	Read delimited single or multi line records (with quoted linefeeds) from plain text files on HDFS. It is not splittable (non parallel) and therefore reading is slower than reading with HdfsTextSimple.	• com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter • com.pivotal.pxf.plugins.hdfs.QuotedLineBreakAccessor • com.pivotal.pxf.plugins.hdfs.StringPassResolver
Hive	Use this when connected to Hive.	• com.pivotal.pxf.plugins.hive.HiveDataFragmenter • com.pivotal.pxf.plugins.hive.HiveAccessor • com.pivotal.pxf.plugins.hive.HiveResolver

Profile	Description	Fragmenter/Accessor/Resolver
HBase	Use this when connected to an HBase data store engine.	• com.pivotal.pxf.plugins.hbase.HBaseDataFragmenter • com.pivotal.pxf.plugins.hbase.HBaseAccessor • com.pivotal.pxf.plugins.hbase.HBaseResolver
Avro	Reading Avro files (i.e fileName.avro).	• com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter • com.pivotal.pxf.plugins.hdfs.AvroFileAccessor • com.pivotal.pxf.plugins.hdfs.AvroResolver
GemFireXD	Use this when connected to GemFireXD	• com.pivotal.pxf.plugins.gemfirexd.GemFireXDFragmenter • com.pivotal.pxf.plugins.gemfirexd.GemFireXDAccessor

Adding and Updating Profiles

Administrators can add new profiles or edit the built-in profiles inside *pxf-profiles.xml* (and apply with the Pivotal Hadoop (HD) Enterprise Command Line Interface). You can use the all the profiles in *pxf-profiles.xml*.

Each profile has a mandatory unique name and an optional description.

In addition, each profile contains a set of plugins that are an extensible set of metadata attributes.

Custom Profile Example

```
<profile>
 <name>MyCustomProfile</name>
 <description>A Custom Profile Example</description>
 <plugins>
     <fragmenter>package.name.CustomProfileFragmenter</fragmenter>
     <accessor>package.name.CustomProfileAccessor</accessor>
     <customPlugin1>package.name.MyCustomPluginValue1</customPlugin1>
     <customPlugin2>package.name.MyCustomPluginValue2</customPlugin2>
 </plugins>
</profile>
```

Deprecated Classnames

Before PXF 2.2.0, you could use connector class names without their package names. e.g. *HdfsDataFragmenter* instead of *com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter*. PXF 2.2.0 does *not* support old names and you will see an error message:

```
WARNING:  Use of HdfsDataFragmenter is deprecated and it will be removed on the next major version
DETAIL:  Please use the appropriate PXF profile for forward compatibility (e.g.
profile=HdfsTextSimple)
```

Pivotal recommends that you use PXF Profiles as best practice.

Recommended Built-in PXF Profiles

Old name	PXF Profile
HdfsDataFragmenter, TextFileAccessor, TextResolver	HdfsTextSimple
HdfsDataFragmenter, QuotedLineBreakAccessor, TextResolver	HdfsTextMulti
HdfsDataFragmenter, AvroFileAccessor, AvroResolver	Avro
HdfsDataFragmenter, SequenceFileAccessor, CustomWritable	SequenceWritable
HBaseDataFragmenter, HBaseAccessor, HBaseResolver	HBase
HiveDataFragmenter, HiveAccessor, HiveResolver	Hive

Here is a list of class names old and new:

Old Name	New Name
TextFileAccessor, LineBreakAccessor, LineReaderAccessor	com.pivotal.pxf.plugins.hdfs.LineBreakAccessor
QuotedLineBreakAccessor	com.pivotal.pxf.plugins.hdfs.QuotedLineBreakAccessor
AvroFileAccessor	com.pivotal.pxf.plugins.hdfs.AvroFileAccessor
SequenceFileAccessor	com.pivotal.pxf.plugins.hdfs.SequenceFileAccessor
TextResolver, StringPassResolver	com.pivotal.pxf.plugins.hdfs.StringPassResolver
AvroResolver	com.pivotal.pxf.plugins.hdfs.AvroResolver
WritableResolver	com.pivotal.pxf.plugins.hdfs.WritableResolver
HdfsDataFragmenter	com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter

Accessing HDFS File Data with PXF

Installing the PXF HDFS plugin

Install the PXF HDFS plugin jar file on all nodes in the cluster:

```
sudo rpm -i pxf-hdfs-2.2.0-x.rpm
```

- PXF RPMs reside in the Pivotal ADS/HAWQ stack file.
- The script installs the JAR file at the default location at */usr/lib/gphd/pxf-2.2.0*. The S*oftlink pxf-hdfs.jar* will be created in */usr/lib/gphd/pxf*

Notes

- Pivotal recommends that you test PXF on HDFS before connecting to Hive or HBase.
- PXF on secure HDFS clusters requires NameNode to be configured on port 8020
- HBase/Hive configurations requiring user authentication are not supported

The syntax for accessing an HDFS file is as follows:

Syntax

```
CREATE [READABLE|WRITABLE] EXTERNAL TABLE <tbl name> (<attr list>)
LOCATION ('pxf://<name node hostname:50070>/<path to file or directory>?Profile=<chosen
profile>[&<additional options>=<value>]')
FORMAT '[TEXT | CSV | CUSTOM]' (<formatting properties>)
[ [LOG ERRORS INTO <error_table>] SEGMENT REJECT LIMIT <count> [ROWS | PERCENT] ];

SELECT ... FROM <tbl name>; --to read from hdfs with READABLE table.
INSERT INTO <tbl name> ...; --to write to hdfs with WRITABLE table.
```

To read the data in the files or to write based on the existing format, you need to select the FORMAT, Profile, or one of the classes.

This topic describes the following:

- FORMAT clause
- Fragmenter
- Accessor

- Resolver

Note

For more details about the API and classes, see the PXF External Table and API Reference.

FORMAT clause

To read data, use the following formats with any PXF connector:

- **FORMAT 'TEXT'**: Use with plain delimited text files on HDFS.
- **FORMAT 'CSV'**: Use with comma-separated value files on HDFS.
- **FORMAT 'CUSTOM'**: Use with other files, such as binary formats. Must always be used with built-in formatter '*pxfwritable_import*' (for read) or '*pxfwritable_export*' (for write).

Fragmenter

Always use either [*HdfsTextSimple | HdfsTextMulti]* Profile or an com.pivotal.pxf.plugins.hdfs. *HdfsDataFragmenter* for HDFS file data*.*

Note

For read tables, you must include a Profile or a Fragmenter in the table definition.

Accessor

The choice of an Accessor depends on the HDFS data file type.

Note: You must include a Profile or an Accessor in the table definition.

File Type	Accessor	FORMAT clause	Comments
Plain Text delimited	com.pivotal.pxf.plugins.hdfs.LineBreakAccessor	FORMAT 'TEXT' (<format param list>)	Read + Write
Plain Text CSV	com.pivotal.pxf.plugins.hdfs.LineBreakAccessor	FORMAT 'CSV' (<format param list>)	LineBreakAccess parallel and faster Use if each logica is a physical data Read + Write

File Type	Accessor	FORMAT clause	Comments
	com.pivotal.pxf.plugins.hdfs.QuotedLineBreakAccessor		QuotedLineBreak slower and non pa Use if the data inc embedded (quote characters. Read Only
SequenceFile	com.pivotal.pxf.plugins.hdfs.SequenceFileAccessor	FORMAT 'CUSTOM' (formatter='pxfwritable_import')	Read + Write (use formatter='pxfwrita for write)
AvroFile	com.pivotal.pxf.plugins.hdfs.AvroFileAccessor	FORMAT 'CUSTOM' (formatter='pxfwritable_import')	Read Only

Resolver

Choose the Resolver format if data records are serialized in the HDFS file.

Note: You must include a Profile or a Resolver in the table definition.

Record Serialization	Resolver	Comments
Avro	com.pivotal.pxf.plugins.hdfs.AvroResolver	• Avro files include the record schema, Avro serialization can be used in other file types (e.g, Sequence File). • For Avro serialized records outside an Avro file, include a schema file name (.avsc) in the url under the optional *Schema-Data* option. • The schema file name must exist in the public stage directory. • Deserialize Only (Read)
Java Writable	com.pivotal.pxf.plugins.hdfs.WritableResolver	• Include the name of the Java class that uses Writable serialization in the url under the optional *Schema-Data.* • The class file must exist in the public stage directory (or in Hadoop's class path). • Deserialize and Serialize (Read + Write) • See Customized Writable Schema File Guidelines

Record Serialization	Resolver	Comments
None (plain text)	com.pivotal.pxf.plugins.hdfs.StringPassResolver	• Does not serialize plain text records. The database parses plain records. Passes records as they are. • Deserialize and Serialize (Read + Write)

Additional Options

Option Name	Description
COMPRESSION_CODEC	• Useful for WRITABLE PXF tables. • Specifies the c ompression codec class name for compressing the written data. The class must implement the org.apache.hadoop.io.compress.CompressionCodec interface. • Some valid values are org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec. • Note: org.apache.hadoop.io.compress.BZip2Codec is runs in a single thread and can be slow. • This option has no default value. • When the option is not defined, no compression will be done.

COMPRESSION_TYPE	• Useful WRITABLE PXF tables with SequenceFileAccessor.
	• Ignored when COMPRESSION_CODEC is not defined.
	• Specifies the compression type for sequence file.
	• Valid options are:
	  ◦ RECORD - only the value part of each row is compressed.
	  ◦ BLOCK - both keys and values are collected in 'blocks' separately and compressed.
	• Default value: RECORD.

SCHEMA-DATA	The data schema file used to create and read the HDFS file. For example, you could create an avsc (for Avro), or a java class (for Writable Serialization) file. Check that the file exists on the public directory (see About the Public Directory). This option has no default value.

Option Name	Description
THREAD-SAFE	Determines if the table query can run in multithread mode or not. When set to FALSE requests will be handled in a single thread. Should be set when a plugin or other element that are not thread safe are used (e.g. compression codec). Allowed values: TRUE, FALSE. Default value is TRUE - requests can run in multithread mode.
<custom>	Any option that is desired to add to the pxf URI string will be accepted and passed along with its value to the Fragmenter, Accessor, Analyzer and Resolver implementations

Accessing data on a High Availability HDFS cluster

To access data on a High Availability HDFS cluster, you need to change the authority in the URI in the LOCATION.
Use <**HA nameservice**> instead of <**name node host:50070**>.

```
CREATE [READABLE|WRITABLE] EXTERNAL TABLE <tbl name> (<attr list>)
LOCATION ('pxf://<HA nameservice>/<path to file or directory>?Profile=<chosen profile>[&<additional options>=<value>]')
FORMAT '[TEXT | CSV | CUSTOM]' (<formatting properties>);
```

You can access the data using SQL queries.

About the Public Directory

PXF provides a space to store your customized serializers and schema files on the filesystem. You must add schema files on all the datanodes and restart the cluster. The RPM creates the directory at the default location: */usr/lib/gphd/publicstage*.

Ensure that all HDFS users have read permissions to HDFS services and limit write permissions to specific users.

Record key in key-value file formats

For sequence file and other file formats that store rows in a key-value format, the key value can be accessed through HAWQ using the saved keyword '*recordkey*' as a field name.

The field type must correspond to the key type, the same as the other fields must match the HDFS data.

WritableResolver supports read and write of recordkey, which can be of the following Writable hadoop types: BooleanWritable, ByteWritable, IntWritable, DoubleWritable, FloatWritable, LongWritable, Text.

If the *recordkey* field is not defined, the key is ignored in read, and a default value (segment id as LongWritable) is written in write.

Example

Let's say we have a data schema Babies.class containing 3 fields: (name text, birthday text, weight float).

An external table must include these three fields, and can either include or ignore the recordkey.

```
-- writable table with recordkey
CREATE WRITABLE EXTERNAL TABLE babies_registry (recordkey int, name text, birthday text, weight
float)
LOCATION
('pxf://namenode_host:50070/babies_1940s?ACCESSOR=com.pivotal.pxf.plugins.hdfs.SequenceFileAccessor&R
'CUSTOM' (formatter='pxfwritable_export');
INSERT INTO babies_registry VALUES (123456, "James Paul McCartney", "June 18, 1942", 3.800);
-- writable table without recordkey
CREATE WRITABLE EXTERNAL TABLE babies_registry2 (name text, birthday text, weight float)
LOCATION
('pxf://namenode_host:50070/babies_1940s?ACCESSOR=com.pivotal.pxf.plugins.SequenceFileAccessor&RESOLV
'CUSTOM' (formatter='pxfwritable_export');
INSERT INTO babies_registry VALUES ("Richard Starkey", "July 7, 1940", 4.0); -- this record's key
will have some default value
```

The same goes for reading data from an existing file of a key-value format, e.g. a Sequence file.

```
-- readable table with recordkey
CREATE EXTERNAL TABLE babies_1940 (recordkey int, name text, birthday text, weight float)
LOCATION
('pxf://namenode_host:50070/babies_1940s?ACCESSOR=com.pivotal.pxf.plugins.hdfs.SequenceFileAccessor&R
'CUSTOM' (formatter='pxfwritable_import');
SELECT * FROM babies_1940; -- retrieves each record's key
-- readable table without recordkey
CREATE EXTERNAL TABLE babies_1940_2 (name text, birthday text, weight float)
LOCATION
('pxf://namenode_host:50070/babies_1940s?ACCESSOR=com.pivotal.pxf.plugins.hdfs.SequenceFileAccessor&R
'CUSTOM' (formatter='pxfwritable_import');
SELECT * FROM babies_1940_2; -- ignores the records' key
```

Customized Writable Schema File Guidelines

When using a WritableResolver, a schema file needs to be defined. The file needs to be a java class file and must be on the class path of PXF.

The class file must follow the following requirements:

1. Must implement org.apache.hadoop.io.Writable interface.
2. WritableResolver uses reflection to recreate the schema and populate its fields (for both read and write). Then it uses the Writable interface functions to read/write.
   Therefore fields must be public to enable access to them. Private fields will be ignored.
3. Fields are accessed and populated by the order in which they are declared in the class file.
4. Supported field types:
   String, int, double, float, long, short, boolean, byte array.
   Array of any of the above types is supported, but the constructor must define the array size so the reflection will work.

Accessing Hive Data with PXF

Installing the PXF HIVE plugin

Install the PXF HIVE plugin on all nodes in the cluster:

```
sudo rpm -i pxf-hive-2.2.0-x.rpm
```

- PXF RPMs reside in the Pivotal ADS/HAWQ stack file.
- The script installs the JAR file at the default location at */usr/lib/gphd/pxf-2.2.0*. The S*oftlink pxf-hive.jar* will be created in */usr/lib/gphd/pxf*

PXF HIVE Prerequisites

Check the following before adding PXF support on Hive:

- PXF HDFS plugin is installed on the cluster nodes.
- You are running the Hive Metastore service on a machine in your cluster.
- Check that you have set the *hive.metastore.uris property* in the *hive-site.xml* on the Namenode.
- The Hive JAR files are installed on the cluster nodes.

See Setting up the Java Classpath.

Syntax

Hive tables are always defined in a specific way in PXF, regardless of the underlying file storage format. The PXF Hive plugins automatically detect source tables:

- Text based
- SequenceFile
- RCFile
- ORCFile

The source table can also be a combination of these types. The PXF Hive plugin uses this information to query the data in runtime. The following PXF table definition is valid for any file storage type.

```
CREATE EXTERNAL TABLE hivetest(id int, newid int)
LOCATION ('pxf://<NN host>:50070/<hive table name>?PROFILE=Hive')
FORMAT 'custom' (formatter='pxfwritable_import');

SELECT * FROM hivetest;
```

Note : 50070 as noted in the example above is the REST server port on the HDFS NameNode. If a different port is assigned in your installation, use that port.

Hive Command Line

To start the Hive command line and work directly on a Hive table:

```
/>${HIVE_HOME}/bin/hive
```

Here's an example of how to create and load data into a sample Hive table from an existing file.

```
Hive> CREATE TABLE test (name string, type string, supplier_key int, full_price double) row format
delimited fields terminated by ',';
Hive> LOAD DATA local inpath '/local/path/data.txt' into table test;
```

Mapping Hive Collection Types

PXF supports Hive data types that are not primitive types. For example :

```
CREATE TABLE sales_collections (
  item STRING,
  price FLOAT,
  properties ARRAY<STRING>,
  hash MAP<STRING,FLOAT>,
  delivery_address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
ROW FORMAT DELIMITED  FIELDS
TERMINATED BY '\001' COLLECTION ITEMS TERMINATED BY '\002' MAP KEYS TERMINATED BY '\003' LINES
TERMINATED BY '\n'  STORED AS TEXTFILE;
LOAD DATA LOCAL INPATH '/local/path/<some data file>' INTO TABLE sales_collection;
```

To query a Hive table schema similar to the one in the example you need to define the PXF external table with attributes corresponding to members in the Hive table array and map fields. For example:

```
CREATE EXTERNAL TABLE gp_sales_collections(
  item_name TEXT,
  item_price REAL,
  property_type TEXT,
  property_color TEXT,
  property_material TEXT,
  hash_key1 TEXT,
  hash_val1 REAL,
  hash_key2 TEXT,
  hash_val3 REAL,
  delivery_street TEXT,
  delivery_city TEXT,
  delivery_state TEXT,
  delivery_zip INTEGER
)
LOCATION ('pxf://<namenode_host>:50070/sales_collections?PROFILE=Hive')
FORMAT 'custom' (FORMATTER='pxfwritable_import');
```

Partition Filtering

The PXF Hive plugin uses the Hive partitioning feature and directory structure. This enables partition exclusion on HDFS files that contain the Hive table. To use the Partition Filtering feature to reduce network traffic and I/O, run a PXF query using a WHERE clause that refers to a specific partition in the partitioned Hive table.

To take advantage of PXF Partition filtering push down, name the partition fields in the external table. These names must be the same as those stored in the Hive table. Otherwise, PXF ignores Partition filtering and the filtering is performed on the HAWQ side, impacting performance.

NOTE

The Hive plugin only filters on partition columns but not on other table attributes.

Example

Create a Hive tabl*e sales_part* with 2 partition columns - *delivery_state* and *delivery_city:*

```
CREATE TABLE sales_part (name STRING, type STRING, supplier_key INT, price DOUBLE)
PARTITIONED BY (delivery_state STRING, delivery_city STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

Load data into this Hive table and add some partitions:

```
LOAD DATA LOCAL INPATH '/local/path/data1.txt' INTO TABLE sales_part PARTITION(delivery_state =
'CALIFORNIA', delivery_city = 'San Francisco');
LOAD DATA LOCAL INPATH '/local/path/data2.txt' INTO TABLE sales_part PARTITION(delivery_state =
'CALIFORNIA', delivery_city = 'Sacramento');
LOAD DATA LOCAL INPATH '/local/path/data3.txt' INTO TABLE sales_part PARTITION(delivery_state =
'NEVADA'    , delivery_city = 'Reno');
LOAD DATA LOCAL INPATH '/local/path/data4.txt' INTO TABLE sales_part PARTITION(delivery_state =
'NEVADA'    , delivery_city = 'Las Vegas');
```

The Hive storage directory should appears as follows:

```
/hive/warehouse/sales_part/delivery_state=CALIFORNIA/delivery_city=’San Francisco’/data1.txt
/hive/warehouse/sales_part/delivery_state=CALIFORNIA/delivery_city=Sacramento/data2.txt
/hive/warehouse/sales_part/delivery_state=NEVADA/delivery_city=Reno/data3.txt
/hive/warehouse/sales_part/delivery_state=NEVADA/delivery_city=’Las Vegas’/data4.txt
```

To define a PXF table to read this Hive table and take advantage of partition filter push down, define the fields corresponding to the Hive partition fields at the end of the attribute list.

When defining an external table, check that the fields corresponding to the Hive partition fields are at the end of the column list. In HiveQL, issuing a *select** statement on a partitioned table shows the partition fields at the end of the record.

```
CREATE EXTERNAL TABLE pxf_sales_part(
  item_name TEXT,
  item_type TEXT,
  supplier_key INTEGER,
  item_price DOUBLE PRECISION,
  delivery_state TEXT,
  delivery_city TEXT
)
LOCATION ('pxf://namenode_host:50070/sales_part?Profile=Hive')
FORMAT 'custom' (FORMATTER='pxfwritable_import');

SELECT * FROM pxf_sales_part;
```

Example

In the following example the HAWQ query filters the *delivery_city* partition *Sacramento*. The filter on *item_name* is not pushed down since it is not a partition column. It is performed on the HAWQ side after all the data on *Sacramento* is transferred for processing.

```
SELECT * FROM pxf_sales_part WHERE delivery_city = 'Sacramento' AND item_name = 'shirt';
```

Example

The following HAWQ query reads all the data under *delivery_city* partition *CALIFORNIA*, regardless of the city partition.

```
SELECT * FROM pxf_sales_part WHERE delivery_state = 'CALIFORNIA'
```

Accessing HBase Data with PXF

Installing the PXF HBase plugin

Install the PXF HBase plugin on all nodes in the cluster:

```
sudo rpm -i pxf-hbase-2.2.0-x.rpm
```

- PXF RPMs reside in the Pivotal ADS/HAWQ stack file.
- The script installs the JAR file at the default location at */usr/lib/gphd/pxf-2.2.0*. The S*oftlink pxf-hbase.jar* will be created in */usr/lib/gphd/pxf*

PXF HBase Prerequisites

Before using PXF HBase plugin, verify the following:

- PXF HDFS plugin is installed on the cluster nodes.
- That you have set the *Hadoop-env.sh* on the Namenode.
- All Datanodes have the *hbase.jar*, *zookeeper.jar* and the *HBase conf* directory set in the *HADOOP_CLASSPATH*.

See *Installing PXF* for more information.

Syntax

To query an *HBase* table use the following syntax:

```
CREATE EXTERNAL TABLE <pxf tblname> (<col list - see details below>)
LOCATION ('pxf://<NN REST host>:<NN REST port>/<HBase table name>?PROFILE=HBase')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');

SELECT * FROM <pxf tblname>;
```

Column Mapping

Most HAWQ external tables (PXF or others) require that the HAWQ table attributes match the source data record layout, and include all the available attributes. However, use the PXF HBase plugin to specify the subset of HBase qualifiers to define the HAWQ PXF table. To set up a clear mapping between each attribute in the PXF table and a specific qualifier in the HBase table, you can use either:

- Directmapping
- Indirect mapping

In addition, the HBase row key is handled in a special way.

Row Key

You can use the HBase table row key in several ways. For example, you can see them using query results, or you can run a WHERE clause filter on a range of row key values. To use the row key in the HAWQ query, define the HAWQ table with the reserved PXF attribute *recordkey.* This attribute name tells PXF to return the record key in any key-value based system and in HBase.

NOTE

Since HBase is byte and not character-based Pivotal recommends that you define the *recordkey* as type *bytea*. This may result in better ability to filter data and increase performance.

```
CREATE EXTERNAL TABLE <tname> (recordkey bytea, ... ) LOCATION ('pxf:// ...')
```

Direct Mapping

Use Direct Mapping to map HAWQ table attributes to HBase qualifiers. You can specify the HBase qualifier names of interest, with column family names included, as quoted values.

For example, you have defined an HBase table called *hbase_sales* with multiple column families and many qualifiers. To see the following in the resulting attribute section of the CREATE EXTERNAL TABLE:

- *rowkey*
- qualifier *saleid* in the column family *cf1*
- qualifier *comments* in the column family *cf8*

```
CREATE EXTERNAL TABLE hbase_sales (
  recordkey bytea,
  “cf1:saleid” int,
  “cf8:comments” varchar
) ...
```

The PXF HBase plugin uses these attribute names as-is and returns the values of these HBase qualifiers.

Indirect Mapping (via Lookup Table)

Direct mapping method is fast and intuitive, but using indirect mapping helps to reconcile HBase qualifier names with HAWQ behavior:

- HBase qualifier names that are longer than 32 characters. HAWQ has a 32 character limit on attribute name size.

- HBase qualifier names can be binary or non-printable. HAWQ attribute names are character based.

In either case, Indirect Mapping uses a lookup table on HBase. You can create the lookup table to store all necessary lookup information. This works as a template for any future queries. The name of the lookup table must be *pxflookup* and must include the column family named *mapping.*

Using the sales example in Direct Mapping, if our *rowkey* represents the HBase table name and the *mapping* column family includes the actual attribute mapping in the key value form of *<hawq attr name>=<hbase cf:qualifier>.*

Example

(row key)	mapping
sales	id=cf1:saleid
sales	cmts=cf8:comments

Note: The mapping assigned new names for each qualifier. You can use these names in your HAWQ table definition:

```
CREATE EXTERNAL TABLE hbase_sales (
  recordkey bytea
  id int,
  cmts varchar
) ...
```

PXF automatically matches HAWQ to HBase column names when a *pxflookup* table exists in HBase.

Accessing GemFire Data with PXF

NOTE

Before using PXF GemFire plugin, verify the following:

- That you have installed the *gfxd rpm* on the Namenode and on the Datanodes.

- The namenode and all Datanodes have the *sqlfire.jar* set in the *HADOOP_CLASSPATH*.

See *Installing PXF* for more information.

Syntax

To query an *GemFire* table use the following syntax:

```
CREATE EXTERNAL TABLE <pxf tblname> (<col list>)
LOCATION ('pxf://<NN REST host>:<NN REST port>/<GemFire table name>?Profile=GemFireXD&<GemFire
specific connector options>')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');

SELECT * FROM <pxf tblname>;
```

The GemfireXD connector has quite a few connector options that can be used in the LOCATION URI, and are well documented in the GemFire document itself. It is highly recommended to learn them carefully in order to get the expected behavior when querying GemFireXD data through PXF.

Troubleshooting

The following table describes some common errors while using PXF:

Table: PXF Errors and Explanation

Error
ERROR: invalid URI pxf://localhost:50070/demo/file1: missing options section
ERROR: protocol "pxf" does not exist
ERROR: remote component error: 0 DETAIL: There is no pxf servlet listening on the host and port specified in the external table url.
ERROR: Missing FRAGMENTER option in the pxf uri: pxf://localhost:50070/demo/file1?a=a
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason:*org.apache.hadoop.mapred.InvalidInputExce*
ERROR: *remote component error: 500* DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *org.apache.hadoop.mapred.InvalidInputExce* files
ERROR: remote component error: 500 PXF not correctly installed in CLASSPATH
ERROR: GPHD component not found

`ERROR:  remote component error (500)`
 	
`Problem accessing /gpdb/v<X>/Fragmenter/getFragments. Reason: java.io.IOException: Can't g`

`ERROR:  fail to get filesystem credential for uri hdfs://<namenode>:8020/`

`ERROR: remote component error (413) from <x>: HTTP status code is 413 but HTTP response st`
HBase Specific Errors
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *org.apache.hadoop.hbase.client.NoServerFo t1,,99999999999999* after 10 tries.
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *org.apache.hadoop.hbase.TableNotFoundEx*
ERROR: remote component error: 500 (/HTTP/1.1 500 Internal Server Error
ERROR: remote component error: 500 (/HTTP/1.1 500 org/apache/zookeeper/KeeperException
ERROR: remote component error: 500 (/HTTP/1.1 500 Illegal HBase column name name
Hive Specific Errors
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *java.net.ConnectException:* Connection refus
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *NoSuchObjectException(message:default.pl*
GemfireXD Specific Errors
No data or wrong data comes back, comes back with very poor performance,

Chapter 2 PXF External Table and API Reference

You can extend PXF functionality and add new services and formats using the Java API without changing HAWQ. The API includes the four classes Fragmenter, Accessor, Resolver, Analyzer. The Fragmenter, Accessor, and Resolver classes must be implemented to add a new service. The Analyzer class is optional.

Topics:

Creating an External Table

Syntax for an *EXTERNAL TABLE* that uses the PXF protocol is as follows:.

```
CREATE EXTERNAL TABLE ext_table <attr list, ...>
LOCATION('pxf://<namenode>:<port>/path/to/data?FRAGMENTER=package.name.FragmenterForX&ACCESSOR=packag
custom user options>=<value>')FORMAT 'custom'(formatter='pxfwritable_import');
```

Where:

Table: Parameter values and description

Parameter	Value and description
namenode	The current host of the PXF service is HDFS Namenode port.
REST	Port for Namenode, 50070 by default.
path/to/data	A directory, file name, wildcard pattern, table name, etc
FRAGMENTER	The plugin (java class) to use for fragmenting data. Used in READABLE external tables only.
ACCESSOR	The plugin (java class) to use for accessing the data. Used in READABLE and WRITABLE tables.
RESOLVER	The plugin (java class) to use for serializing and deserializing the data. Used in READABLE and WRITABLE tables.
Custom Options	Anything else that is desired to add. Will be passed in runtime to the plugins indicated above.

For more information about this example, see *"About the Java Class Services and Formats" and the Pivotal Extension Installation and Administrator Guide.*

About the Java Class Services and Formats

The Java class names you must include in the PXF URI are Fragmenter, Accessor, and Resolver. The Fragmenter class is mandatory for READABLE tables, and not supported for WRITABLE tables. Pivotal recommends that you reuse a previously-defined Accessor or Resolver data format.

All the attributes are passed from HAWQ as headers to the PXF Java service. The Java service retrieves the source data and converts it to a HAWQ-readable format. You can pass any additional information to the user-implemented services.

The example in *"Creating an External Table"* shows the available keys and associated values. The example also contains attributes that are passed in from the HAWQ side. The available keys and associated values are as follows:

```
FRAGMENTER: ‘pkg.name.FragmenterForX’
ACCESSOR: ‘pkg.name.AccessorForX’
RESOLVER: ‘pkg.name.ResolverForX’
```

These three Java plugins and the optional plugin, Analyzer, extend the *com.pivotal.pxf.api.utilities.Plugin* class.

The Java classes can be described as follows:

```
package com.pivotal.pxf.api.utilities;
/*
 * Base class for all plugin types (Accessor, Resolver, Fragmenter, Analyzer)
 * Holds InputData as well (the meta data information used by all plugin types)
 */
public class Plugin
{
    protected InputData inputData;

    /*
     * C'tor
     */
    public Plugin(InputData input);

    /**
     * Checks if the plugin is thread safe or not, based on inputData.
     *
     * @return if plugin is thread safe or not
     */
    public boolean isThreadSafe() {
        return true;
    }
}
```

Attributes are available through the *com.pivotal.pxf.api.utilities.InputData* class. The following example shows how *inputData.getProperty(‘USERINFO1’)* returns *optional_info*.

```
package com.pivotal.pxf.api.utilities;
/*
 * Common configuration of all MetaData classes
 * Provides read-only access to common parameters supplied using system properties
 */
public class InputData
{
    /*
     * Constructor of InputData
     * Parses X-GP-* configuration variables
     *
     * @param paramsMap contains all query-specific parameters from Hawq
     * @param servletContext Servlet context contains attributes required by SecuredHDFS
     */
    public InputData(Map<String, String> paramsMap);

    /*
     * Expose the parameters map
     */
    public Map<String, String> getParametersMap();

    /* Copy contructor of InputData
     * Used to create from an extending class
     */
    public InputData(InputData copy);

    /*
     * Returns a property as a string type
     */
    public String getProperty(String property);

    /*
     * returns the number of segments in GP
     */
    public int totalSegments();

    /*
     * returns the current segment ID
     */
    public int segmentId();

    /* returns the current outputFormat
     * currently either text or gpdbwritable
     */
    public OutputFormat outputFormat();

    /*
     * returns the server name providing the service
     */
    public String serverName();

    /*
     * returns the server port providing the service
     */
    public int serverPort();
```

```
    /*
     * Returns true if there is a filter string to parse
     */
    public boolean hasFilter();

    /*
     * The filter string
     */
    public String filterString();

    /*
     * returns the number of columns in Tuple Description
     */
    public int columns();

    /*
     * returns column index from Tuple Description
     */
    public ColumnDescriptor getColumn(int index);

    /*
     * returns fragment serialized metadata
     */
    public byte[] getFragmentMetadata();

    /*
     * Set fragment serialized metadata
     */
    public void setFragmentMetadata(byte[] location);
    /*
     * returns fragment user data
     */
    public byte[] getFragmentUserData();
    /*
     * returns a data fragment index
     */
    public int getDataFragment();
    /*
     * returns the column descriptor of the recordkey column.
     * If the recordkey column was not specified by the user in the create table statement,
     * then getRecordkeyColumn will return null.
     */
    public ColumnDescriptor getRecordkeyColumn();

    /*
     * Returns the data source of the required resource (i.e a file path or a table name).
     */
    public String dataSource();

     /*
     * Sets the data source of the required resource (i.e a file path or a table name).
     */
    public void setDataSource(String dataSource);
    /* returns the path of the schema used for various deserializers
     * e.g, Avro file name, Java object file name.
     */
```

```
    public String srlzSchemaName() throws FileNotFoundException, IllegalArgumentException;

    /*
     * returns the ClassName for the java class that handles the file access
     */
    public String accessor();
    /*
     * returns the ClassName for the java class that handles the record deserialization
     */
    public String resolver();

    /*
     * The avroSchema fetched by the AvroResolver and used in case of Avro File
     * In case of avro records inside a sequence file this variable will be null
     * and the AvroResolver will not use it.
     */
    public Object getSchema();

    /*
     * The avroSchema is set from the outside by the AvroFileAccessor
     */
    public void setSchema(Object schema);

    /*
     * Returns the compression codec (can be null - means no compression)
     */
    public String compressCodec();
}
```

Fragmenter

Note

The Fragmenter Plugin reads data into HAWQ. Such tables are called READABLE PXF tables. The Fragmenter Plugin cannot write data out of HAWQ. Such tables are called WRITABLE tables.

The Fragmenter is responsible for passing datasource metadata back to HAWQ. It also returns a list of data fragments to the Accessor or Resolver. Each data fragment describes some part of the requested data set. It contains the datasource name, such as the file or table name, including the hostname where it is located. For example, if the source is a HDFS file, the Fragmenter returns a list of data fragments containing a HDFS file block. Each fragment includes the location of the block. If the source data is an HBase table, the Fragmenter returns information about table regions, including their locations.

The following implementations are shipped with PXF 2.2 and higher:

```
com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter
com.pivotal.pxf.plugins.hbase.HBaseDataFragmenter
com.pivotal.pxf.plugins.hive.HiveDataFragmenter
```

The Fragmenter.getFragments() methods returns a List<Fragment>:

```
package com.pivotal.pxf.api;
/*
 * Fragment holds a data fragment' information.
 * Fragmenter.getFragments() returns a list of fragments.
 */
public class Fragment
{
     private String sourceName;    // File path+name, table name, etc.
    private int index;        // Fragment index (incremented per sourceName)
    private String[] replicas;    // Fragment replicas (1 or more)
    private byte[] metadata;    // Fragment metadata information (starting point + length, region
location, etc.)
    private byte[] userData;    // ThirdParty data added to a fragment. Ignored if null
    ...
}
```

Any Fragmenter class needs to extend *com.pivotal.pxf.api.Fragmenter*:

com.pivotal.pxf.api.Fragmenter

```
package com.pivotal.pxf.api;
/*
 * Interface that defines the splitting of a data resource into fragments that can be processed in
parallel
 * GetFragments returns the fragments information of a given path (source name and location of each
fragment).
 * Used to get fragments of data that could be read in parallel from the different segments.
 */
public abstract class Fragmenter extends Plugin {
        protected List<Fragment> fragments;

    public Fragmenter(InputData metaData) {
        super(metaData);
        fragments = new LinkedList<Fragment>();
    }

    /*
     * path is a data source URI that can appear as a file name, a directory name  or a wildcard
     * returns the data fragments
     */
    public abstract List<Fragment> getFragments() throws Exception;
}
```

Class Description

getFragments() returns a string in a JSON format of the retrieved fragment. For example, if the input path is a HDFS directory, the source name for each fragment should include the file name including the path for the fragment.

Accessor

The Accessor retrieves specific fragments and passes records back to the Resolver. For example, the Accessor creates a *FileInputFormat* and a Record Reader for an HDFS file and sends this to the Resolver. In the case of HBase or Hive files, the Accessor returns single rows from an HBase or Hive table. PXF 1.x or higher contains the following implementations:

Table: Accessor base classes

Accessor class	Description
com.pivotal.pxf.plugins.hdfs.HdfsAtomicDataAccessor	Base class for accessing datasources which cannot be split. These will be accessed by a single HAWQ segment. `QuotedLineBreakAccessor - Accessor for TEXT files that has records with embedded linebreaks`

com.pivotal.pxf.plugins.hdfs.HdfsSplittableDataAccessor	Base class for accessing HDFS files using *RecordReaders*:
	`LineBreakAccessor - Accessor for TEXT files (replaced the deprecated TextFileAccessor, LineReaderAccessor)`
	`AvroFileAccessor - Accessor for Avro files`

com.pivotal.pxf.plugins.hive.HiveAccessor	Accessor for Hive tables
com.pivotal.pxf.plugins.hbase.HBaseAccessor	Accessor for HBase tables

The class needs to extend the *com.pivotal.pxf.Plugin class,* and implement one or both interface*s:*

- *com.pivotal.pxf.api.ReadAccessor*
- *com.pivotal.pxf.api.WriteAccessor*

```
package com.pivotal.pxf.api;
/*
 * Internal interface that defines the access to data on the source
 * data store (e.g, a file on HDFS, a region of an HBase table, etc).
 * All classes that implement actual access to such data sources must
 * respect this interface
 */
public interface ReadAccessor {
    public boolean openForRead() throws Exception;
    public OneRow readNextObject() throws Exception;
    public void closeForRead() throws Exception;
}
```

```
package com.pivotal.pxf.api;
/*
 * An interface for writing data into a data store
 * (e.g, a sequence file on HDFS).
 * All classes that implement actual access to such data sources must
 * respect this interface
 */
public interface WriteAccessor {
    public boolean openForWrite() throws Exception;
    public OneRow writeNextObject(OneRow onerow) throws Exception;
    public void closeForWrite() throws Exception;
}
```

The Accessor calls o*penForRead() to read existing data*. After reading the data, it calls c*loseForRead()*. *readNextObject()* and returns one of the following:

- a single record, encapsulated in a OneRow object
- null if it reaches *EOF*

The Accessor calls *openForWrite()* to write data out. After writing the data, it writes a *OneRow* object with *writeNextObject()*, and when done calls *closeForWrite()*. *OneRow* represents a key-value item.

com.pivotal.pxf.api.OneRow:

```
package com.pivotal.pxf.api;
/*
 * Represents one row in the external system data store. Supports
 * the general case where one row contains both a record and a
 * separate key like in the HDFS key/value model for MapReduce
 * (Example: HDFS sequence file)
 */
public class OneRow {
    /*
     * Default constructor
     */
    public OneRow()

    /*
     * Constructor sets key and data
     */
    public OneRow(Object inKey, Object inData)

    /*
     * Copy constructor
     */
    public OneRow(OneRow copy)

    /*
     * Setter for key
     */
    public void setKey(Object inKey)

    /*
     * Setter for data
     */
    public void setData(Object inData)
    /*
     * Accessor for key
     */
    public Object getKey()

    /*
     * Accessor for data
     */
    public Object getData()

    /*
     * Show content
     */
    public String toString()
}
```

Resolver

The Resolver deserializes records in the *OneRow* format and serializes them to a list of *OneField* objects. PXF converts a *OneField* object to a HAWQ-readable *GPDBWritable* format. PXF 1.x or higher contains the following implementations:

Table: Resolver base classes

Resolver class	Description
com.pivotal.pxf.plugins.hdfs.StringPassResolver	Supports: `GPBWritable VARCHAR` *StringPassResolver* replaced the deprecated *TextResolver.* It passes whole records (composed of any data types) as strings without parsing them

com.pivotal.pxf.plugins.hdfs.WritableResolver	Resolver for custom Hadoop Writable implementations. Custom class can be specified with the schema *{{,}}* and supports the following:
	`DataType.BOOLEAN`
	`DataType.INTEGER`
	`DataType.BIGINT`
	`DataType.REAL`
	`DataType.FLOAT8`
	`DataType.VARCHAR`
	`DataType.BYTEA`

com.pivotal.pxf.plugins.hdfs.AvroResolver	Supports the same field objects as *WritableResolver*.

com.pivotal.pxf.plugins.hbase.HBaseResolver	Supports the same field objects as *WritableResolver* and also supports the following:
	`DataType.SMALLINT`
	`DataType.NUMERIC`
	`DataType.TEXT`
	`DataType.BPCHAR`
	`DataType.TIMESTAMP`

com.pivotal.pxf.plugins.hive.HiveResolver	Supports the same field objects as *WritableResolver* and also supports the following:
	`DataType.SMALLINT`
	`DataType.TEXT`
	`DataType.TIMESTAMP`

The class needs to extend the *com.pivotal.pxf.resolvers.Plugin class* , and implement one or both interface*s:*

- *com.pivotal.pxf.api.ReadResolver*

- *com.pivotal.pxf.api.WriteResolver*

```
package com.pivotal.pxf.api;
/*
 * Interface that defines the deserialization of one record brought from
 * the data Accessor. Every implementation of a deserialization method
 * (e.g, Writable, Avro, ...) must implement this interface.
 */
public interface ReadResolver {
    public List<OneField> getFields(OneRow row) throws Exception;
}
```

```
package com.pivotal.pxf.api;
/*
* Interface that defines the serialization of data read from the DB
* into a OneRow object.
* Every implementation of a serialization method
* (e.g, Writable, Avro, ...) must implement this interface.
*/
public interface WriteResolver {
    public OneRow setFields(List<OneField> record) throws Exception;
}
```

Notes

- getFields should return a List<OneField>, each OneField representing a single field.
- *setFields* should return a single *OneRow* object, given a List<OneField>.

com.pivotal.pxf.api.OneField

```
package com.pivotal.pxf.api;
/*
 * Defines one field on a deserialized record.
 * 'type' is in OID values recognized by GPDBWritable
 * 'val' is the actual field value
 */
public class OneField {
    public OneField() {}
    public OneField(int type, Object val)    {
        this.type = type;
        this.val = val;
    }

    public int type;
    public Object val;
}
```

The value of t*ype* should follow the com.pivotal.pxf.api.io.DataType *enums*. v*al* is the appropriate Java class. Supported types are as follows:

Table: Resolver supported types

DataType recognized OID	Field value
DataType.SMALLINT	*Short*
DataType.INTEGER	*Integer*
DataType.BIGINT	*Long*
DataType.REAL	*Float*
DataType.FLOAT8	*Double*
DataType.NUMERIC	*String ("651687465135468432168421")*
DataType.BOOLEAN	Boolean
DataType.VARCHAR	
DataType.BPCHAR	
DataType.TEXT	*String*
DataType.BYTEA	*byte []*
DataType.TIMESTAMP	*Timestamp*

Analyzer

The Analyzer provides PXF statistical data for the HAWQ query optimizer. For a detailed explanation about HAWQ statistical data gathering, see *ANALYZE* in the *Pivotal ADS Administrator Guide*. Implement the PXF Analyzer for the HDFS text, sequence, and AVRO files. For HBase tables and Hive tables, the Analyzer returns default values.

Notes:

- The new *boolean guc pxf_enable_stat_collection* requests statistics. The default value is *on*. When you turn it off, the statistics collected reflect default values.
- Pivotal recommends that you implement the Analyzer to return an estimated result as fast as possible.

The class needs to extend *com.pivotal.pxf.api.Analyzer*.

com.pivotal.pxf.analyzers.Analyzer

```
package com.pivotal.pxf.api;
import com.pivotal.pxf.api.utilities.InputData;
import com.pivotal.pxf.api.utilities.Plugin;

/*
 * Abstract class that defines getting statistics for ANALYZE.
 * getEstimatedStats returns statistics for a given path
 * (block size, number of blocks, number of tuples).
 * Used when calling ANALYZE on a PXF external table, to get
 * table's statistics that are used by the optimizer to plan queries.
 */
public abstract class Analyzer extends Plugin
{
    public Analyzer(InputData inputData)
    {
        super(inputData);
    }

    /*
     * 'path' is the data source name (e.g, file, dir, wildcard, table name).
     * returns the data statistics in json format.
     *
     * NOTE: It is highly recommended to implement an extremely fast logic
     * that returns *estimated* statistics. Scanning all the data for exact
     * statistics is considered bad practice.
     */
    public AnalyzerStats getEstimatedStats(String data) throws Exception
    {
        /* Return default values */
        return new AnalyzerStats();
    }
}
```

getEstimatedStats creates a*n AnalyzerStats* , and returns the result *AnalyzerStats.dataToJSON*.

com.pivotal.pxf.api. AnalyzerStats

```
package com.pivotal.pxf.api;
import java.io.IOException;
import org.codehaus.jackson.map.ObjectMapper;

/*
 * AnalyzerStats is a public class that represents the size
 * information of given path.
 */
public class AnalyzerStats {
    private static final long DEFAULT_BLOCK_SIZE = 67108864L; // 64MB (in bytes)
    private static final long DEFAULT_NUMBER_OF_BLOCKS = 1L;
    private static final long DEFAULT_NUMBER_OF_TUPLES = 1000000L;

    private long    blockSize;        // block size (in bytes)
    private long    numberOfBlocks;    // number of blocks
    private long    numberOfTuples; // number of tuples

    public AnalyzerStats(long blockSize,
                         long numberOfBlocks,
                         long numberOfTuples)
    {
        this.setBlockSize(blockSize);
        this.setNumberOfBlocks(numberOfBlocks);
        this.setNumberOfTuples(numberOfTuples);
    }

    /*
     * Default values
     */
    public AnalyzerStats()
    {
        this(DEFAULT_BLOCK_SIZE, DEFAULT_NUMBER_OF_BLOCKS, DEFAULT_NUMBER_OF_TUPLES);
    }

    /*
     * Given a AnalyzerStats, serialize it in JSON to be used as
     * the result string for HAWQ. An example result is as follows:
     *
     * {"PXFDataSourceStats":{"blockSize":67108864,"numberOfBlocks":1,"numberOfTuples":5}}
     */
    public static String dataToJSON(AnalyzerStats stats) throws IOException
    {
        ObjectMapper    mapper    = new ObjectMapper();
        // mapper serializes all members of the class by default
        return "{\"PXFDataSourceStats\":" + mapper.writeValueAsString(stats) + "}";
    }
    /*
     * Given a stats structure, convert it to be readable. Intended
     * for debugging purposes only. 'datapath' is the data path part of
     * the original URI (e.g., table name, *.csv, etc).
```

```
     */
    public static String dataToString(AnalyzerStats stats, String datapath) {
        return "Statistics information for \"" + datapath + "\" " +
                " Block Size: " + stats.blockSize +
                ", Number of blocks: " + stats.numberOfBlocks +
                ", Number of tuples: " + stats.numberOfTuples;
    }
    public long getBlockSize() {
        return blockSize;
    }
    private void setBlockSize(long blockSize) {
        this.blockSize = blockSize;
    }
    public long getNumberOfBlocks() {
        return numberOfBlocks;
    }
    private void setNumberOfBlocks(long numberOfBlocks) {
        this.numberOfBlocks = numberOfBlocks;
    }
    public long getNumberOfTuples() {
        return numberOfTuples;
    }
    private void setNumberOfTuples(long numberOfTuples) {
        this.numberOfTuples = numberOfTuples;
    }
}
```

About Custom Profiles

Administrators can add new profiles or edit the built-in profiles in *pxf-profiles.xml* file. You need to apply the changes using ICM reconfigure. All PXF users can use the profiles in *pxf-profiles.xml*.

Each profile mandatory unique name, and an optional description.

In addition, each profile contains a set of plugins that are an extensible set of metadata attributes.

Custom Profile Example

```
<profile>
    <name>MyCustomProfile</name>
    <description>A Custom Profile Example</description>
    <plugins>
        <fragmenter>package.name.CustomProfileFragmenter</fragmenter>
        <accessor>package.name.CustomProfileAccessor</accessor>
        <customPlugin1>package.name.MyCustomPluginValue1</customPlugin1>
        <customPlugin2>package.name.MyCustomPluginValue2</customPlugin2>
    </plugins>
</profile>
```

About Query Filter Push-Down

If a query includes a number of WHERE clause filters, HAWQ may push all or some queries to PXF. If pushed to PXF, the Accessor can use the filtering information when accessing the data source to fetch tuples. These filters only return records that pass filter evaluation conditions. This reduces data processing and reduces network traffic from the SQL engine.

This topic includes the following information:

- Filter Availability and Ordering
- Creating a Filter Builder class
- Filter Operations
- Sample Implementation
- Using Filters

Filter Availability and Ordering

PXF allows push-down filtering if the following rules are met:

- Only single expressions or a group of AND'ed expressions - no OR'ed expressions.
- Only expressions of supported data types and operators. See the *Pivotal Extension Framework Installation and Administration Guide* for more information.

FilterParser scans the pushed down filter list and uses the user's build() implementation to build the filter.

- For simple expressions (e.g, a >= 5) FilterParser places column objects on the left of the expression and therefore constants on the right.
- For compound expressions (e.g <expression> AND <expression>) it handles three cases in the build() function:

1. Simple Expression: <Column Index> <Operation> <Constant>
2. Compound Expression: <Filter Object> AND <Filter Object>
3. Compound Expression: <List of Filter Objects> AND <Filter Object>

Creating a Filter Builder Class

To check if a filter queried PXF, call the *InputData hasFilter()* function:

```
/*
  * Returns true if there is a filter string to parse
  */
 public boolean hasFilter()
 {
     return filterStringValid;
 }
```

If *hasFilter()* returns *false*, there is no filter information. If it returns *true*, PXF parses the serialized filter string into a meaningful filter object to use later. To do so, create a filter builder class that implements the *FilterParser.FilterBuilder* interface:

```
/*
  * Interface a user of FilterParser should implement
  * This is used to let the user build filter expressions in the manner she
  * sees fit
  *
  * When an operator is parsed, this function is called to let the user decide
  * what to do with it operands.
  */
 interface FilterBuilder {
     public Object build(Operation operation, Object left, Object right) throws Exception;
 }
```

While PXF parses the serialized filter string from the incoming HAWQ query, it calls the *build() interface* function. PXF calls this function for each condition or filter pushed down to PXF. Implementing this function returns some Filter object or representation that the Accessor or Resolver uses in runtime to filter out records. The *build()* function accepts an Operation as input, and left and right operands.

Filter Operations

```
/*
  * Operations supported by the parser
  */
 public enum Operation
 {
     HDOP_LT, //less than
     HDOP_GT, //greater than
     HDOP_LE, //less than or equal
     HDOP_GE, //greater than or equal
     HDOP_EQ, //equal
     HDOP_NE, //not equal
     HDOP_AND //AND'ed conditions
 };
```

Filter Operands

There are three types of operands:

- Column Index
- Constant
- Filter Object

Column Index

```
/*
  * The class represents a column index
  * It used to know the type of an operand in the stack
  */
 public class ColumnIndex
 {
     private int index;

     public ColumnIndex(int idx)
     {
         index = idx;
     }

     public int index()
     {
         return index;
     }
 }
```

Constant

```
/*
  * The class represents a constant object (String, Long, ...)
  * It used to know the type of an operand in the stack
  */
 public class Constant
 {
      private Object constant;

      public Constant(Object obj)
      {
          constant = obj;
      }

      public Object constant()
      {
          return constant;
      }
 }
```

Filter Object

Filter Objects can be internal, such as those you define; or external, those that the remote system uses. For example, for HBase, you define the HBase *Filter* class (*org.apache.hadoop.hbase.filter.Filter*), while for Hive, you use an internal default representation created by the PXF framework, called *BasicFilter* . You can decide the filter object to use, including writing a new one. *BasicFilter* is the most common:

```
/*
  * Basic filter provided for cases where the target storage system does not provide it's own
filter
  * For example: Hbase storage provides it's own filter but for a Writable based record in a
SequenceFile
  * there is no filter provided and so we need to have a default
  */
 static public class BasicFilter
 {
     private Operation oper;
     private ColumnIndex column;
     private Constant constant;

     /*
      * C'tor
      */
     public BasicFilter(Operation inOper, ColumnIndex inColumn, Constant inConstant)
     {
         oper = inOper;
         column = inColumn;
         constant = inConstant;
     }

     /*
      * returns oper field
      */
     public Operation getOperation()
     {
         return oper;
     }

     /*
      * returns column field
      */
     public ColumnIndex getColumn()
     {
         return column;
     }

     /*
      * returns constant field
      */
     public Constant getConstant()
     {
         return constant;
     }
 }
```

Sample Implementation

Let's look at the following sample implementation of the filter builder class and its *build()* function that handles all 3 cases. Let's assume that BasicFilter was used to hold our filter operations

```
public class MyDemoFilterBuilder implements FilterParser.FilterBuilder
{
    private InputData inputData;

    public MyDataFilterBuilder(InputData input)
    {
        inputData = input;
    }

    /*
     * Translates a filterString into a FilterParser.BasicFilter or a list of such filters
     */
    public Object getFilterObject(String filterString) throws Exception
    {
        FilterParser parser = new FilterParser(this);
        Object result = parser.parse(filterString);

        if (!(result instanceof FilterParser.BasicFilter) && !(result instanceof List))
            throw new Exception("String " + filterString + " resolved to no filter");

        return result;
    }

    public Object build(FilterParser.Operation opId,
                        Object leftOperand,
                        Object rightOperand) throws Exception
    {
        if (leftOperand instanceof FilterParser.BasicFilter)
        {
            //sanity check
            if (opId != FilterParser.Operation.HDOP_AND || !(rightOperand instanceof
FilterParser.BasicFilter))
                throw new Exception("Only AND is allowed between compound expressions");

            //case 3
            if (leftOperand instanceof List)
                return handleCompoundOperations((List<FilterParser.BasicFilter>)leftOperand,
(FilterParser.BasicFilter)rightOperand);
            //case 2
            else
                return handleCompoundOperations((FilterParser.BasicFilter)leftOperand,
(FilterParser.BasicFilter)rightOperand);
        }

        //sanity check
        if (!(rightOperand instanceof FilterParser.Constant))
            throw new Exception("expressions of column-op-column are not supported");

        //case 1 (assume column is on the left)
        return handleSimpleOperations(opId, (FilterParser.ColumnIndex)leftOperand,
(FilterParser.Constant)rightOperand);
    }

    private FilterParser.BasicFilter handleSimpleOperations(FilterParser.Operation opId,
                                                            FilterParser.ColumnIndex column,
```

```
                                                              FilterParser.Constant constant)
    {
        return new FilterParser.BasicFilter(opId, column, constant);
    }

    private  List handleCompoundOperations(List<FilterParser.BasicFilter> left,
                                        FilterParser.BasicFilter right)
    {
        left.add(right);
        return left;
    }

    private List handleCompoundOperations(FilterParser.BasicFilter left,
                                           FilterParser.BasicFilter right)
    {
        List<FilterParser.BasicFilter> result = new LinkedList<FilterParser.BasicFilter>();

         result.add(left);
        result.add(right);
        return result;
    }
}
```

Here is an example of creating a filter builder class to implement the Filter interface, implement the *build()* function, and generate the Filter object. To do this, use either the Accessor, Resolver, or both to call the *getFilterObject* function:

```
if (inputData.hasFilter())
{
    String filterStr = inputData.filterString();
    DemoFilterBuilder demobuilder = new DemoFilterBuilder(inputData);
    Object filter = demobuilder.getFilterObject(filterStr);
    ...
}
```

Using Filters

Once you have built the FIlter object(s), you can use them to read data and filter out records that do not meet the filter conditions:

1. Check whether you have a single or multiple filters.
2. Evaluate each filter and iterate over each filter in the list. Disqualify the record if a filter conditions fail.

```
if (filter instanceof List)
{
    for (Object f : (List)filter)
        <evaluate f>; //may want to break if evaluation results in negative answer for any filter.
}
else
{
    <evaluate filter>;
}
```

Example of evaluating a single filter:

```
//Get our BasicFilter Object
FilterParser.BasicFilter bFilter = (FilterParser.BasicFilter)filter;


//Get operation and operator values
FilterParser.Operation op = bFilter.getOperation();
int colIdx = bFilter.getColumn().index();
String val = bFilter.getConstant().constant().toString();

//Get more info about the column if desired
ColumnDescriptor col = input.getColumn(colIdx);
String colName = filterColumn.columnName();

//Now evaluate it against the actual column value in the record...
```

Reference

This section contains the following information:

- External Table Samples
- Plugin Examples
- Configuration Files

External Table Examples

Example 1

Shows an external table that can analyze all *Sequencefiles* that are populated *Writable* serialized records and exist inside the hdfs directory *sales/2012/01*. *SaleItem.class* is a java class that implements the *Writable* interface and describes a java record that includes three class members.

Note: In this example the class member names do not necessarily match the database attribute names, but the types match. *SaleItem.class* must exist in the classpath of every Datanode.

```
CREATE EXTERNAL TABLE jan_2012_sales (id int, total int, comments varchar)
LOCATION
('pxf://10.76.72.26:50070/sales/2012/01/*.seq?FRAGMENTER=com.pivotal.pxf.plugins.hdfs.HdfsDataFragmen
'custom' (formatter='pxfwritable_import');
```

Example 2

Shows an external table that can analyze an HBase table called *sales*. It has 10 column families *(cf1 – cf10)* and many qualifier names in each family. This example focuses on the *rowkey*, the qualifier *saleid* inside column family *cf1*, and the qualifier *comments* inside column family *cf8* and uses Direct Mapping:

```
CREATE EXTERNAL TABLE hbase_sales (hbaserowkey text, "cf1:saleid" int, "cf8:comments" varchar)
LOCATION
('pxf://10.76.72.26:50070/sales?PROFILE=HBase')
FORMAT 'custom' (formatter='pxfwritable_import');
```

Example 3

This example uses Indirect Mapping. Note how the attribute name changes and how they correspond to the HBase lookup table. Executing a *SELECT from my_hbase_sales*, the attribute names automatically convert to their HBase correspondents.

```
CREATE EXTERNAL TABLE my_hbase_sales (hbaserowkey text, id int, cmts varchar)
LOCATION
('pxf://10.76.72.26:8080/sales?PROFILE=HBase')
FORMAT 'custom' (formatter='pxfwritable_import');
```

Example 4

Shows an example for writable table of compressed data.

```
CREATE WRITABLE EXTERNAL TABLE sales_aggregated_2012 (id int, total int, comments varchar)
LOCATION
('pxf://10.76.72.26:8080/sales/2012/aggregated?PROFILE=HdfsTextSimple&COMPRESSION_CODEC=org.apache.ha
'TEXT';
```

Example 5

Shows an example for writable table into sequence file, using schema file. Note that for write, the formatter is *pxfwritable_export.*

```
CREATE WRITABLE EXTERNAL TABLE sales_max_2012 (id int, total int, comments varchar)
LOCATION
('pxf://10.76.72.26:8080/sales/2012/max?FRAGMENTER=com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter&AC
'custom' (formatter='pxfwritable_export');
```

Plugin Examples

This section contains sample dummy implantations of all four plug-ins. It also contains a usage example.

Dummy Fragmenter

```
import com.pivotal.pxf.api.Fragmenter;
import com.pivotal.pxf.api.Fragment;
import com.pivotal.pxf.api.utilities.InputData;
import java.util.List;

/*
 * Class that defines the splitting of a data resource into fragments that can
 * be processed in parallel
 * getFragments() returns the fragments information of a given path (source name and location of
each fragment).
 * Used to get fragments of data that could be read in parallel from the different segments.
 * Dummy implementation, for documentation
 */
public class DummyFragmenter extends Fragmenter {
    public DummyFragmenter(InputData metaData) {
        super(metaData);
    }
    /*
     * path is a data source URI that can appear as a file name, a directory name  or a wildcard
     * returns the data fragments - identifiers of data and a list of available hosts
     */
    @Override
    public List<Fragment> getFragments() throws Exception {
        String localhostname = java.net.InetAddress.getLocalHost().getHostName();
        String[] localHosts = new String[]{localhostname, localhostname};
        fragments.add(new Fragment(inputData.dataSource() + ".1" /* source name */,
                localHosts /* available hosts list */,
                "fragment1".getBytes()));
        fragments.add(new Fragment(inputData.dataSource() + ".2" /* source name */,
                localHosts /* available hosts list */,
                "fragment2".getBytes()));
        fragments.add(new Fragment(inputData.dataSource() + ".3" /* source name */,
                localHosts /* available hosts list */,
                "fragment3".getBytes()));
        return fragments;
    }
}
```

Dummy Accessor

```
import com.pivotal.pxf.api.ReadAccessor;
import com.pivotal.pxf.api.WriteAccessor;
import com.pivotal.pxf.api.OneRow;
import com.pivotal.pxf.api.utilities.InputData;
import com.pivotal.pxf.api.utilities.Plugin;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/*
```

```
 * Internal interface that defines the access to a file on HDFS.  All classes
 * that implement actual access to an HDFS file (sequence file, avro file,...)
 * must respect this interface
 * Dummy implementation, for documentation
 */
public class DummyAccessor extends Plugin implements ReadAccessor, WriteAccessor {
    private static final Log LOG = LogFactory.getLog(DummyAccessor.class);
    private int rowNumber;
    private int fragmentNumber;
    public DummyAccessor(InputData metaData) {
        super(metaData);
    }
    @Override
    public boolean openForRead() throws Exception {
        /* fopen or similar */
        return true;
    }
    @Override
    public OneRow readNextObject() throws Exception {
        /* return next row , <key=fragmentNo.rowNo, val=rowNo,text,fragmentNo>*/
        /* check for EOF */
        if (fragmentNumber > 0)
            return null; /* signal EOF, close will be called */
        int fragment = inputData.getDataFragment();
        String fragmentMetadata = new String(inputData.getFragmentMetadata());
        /* generate row */
        OneRow row = new OneRow(fragment + "." + rowNumber, /* key */
                rowNumber + "," + fragmentMetadata + "," + fragment /* value */);
        /* advance */
        rowNumber += 1;
        if (rowNumber == 2) {
            rowNumber = 0;
            fragmentNumber += 1;
        }
        /* return data */
        return row;
    }
    @Override
    public void closeForRead() throws Exception {
        /* fclose or similar */
    }
    @Override
    public boolean openForWrite() throws Exception {
        /* fopen or similar */
        return true;
    }
    @Override
    public boolean writeNextObject(OneRow onerow) throws Exception {
        LOG.info(onerow.getData());
        return true;
    }
    @Override
    public void closeForWrite() throws Exception {
        /* fclose or similar */
    }
}
```

Dummy Resolver

```
import com.pivotal.pxf.api.OneField;
import com.pivotal.pxf.api.OneRow;
import com.pivotal.pxf.api.ReadResolver;
import com.pivotal.pxf.api.WriteResolver;
import com.pivotal.pxf.api.utilities.InputData;
import com.pivotal.pxf.api.utilities.Plugin;
import java.util.LinkedList;
import java.util.List;
import static com.pivotal.pxf.api.io.DataType.INTEGER;
import static com.pivotal.pxf.api.io.DataType.VARCHAR;

/*
 * Class that defines the deserializtion of one record brought from the external input data.
 * Every implementation of a deserialization method (Writable, Avro, BP, Thrift, ...)
 * must inherit this abstract class
 * Dummy implementation, for documentation
 */
public class DummyResolver extends Plugin implements ReadResolver, WriteResolver {
    private int rowNumber;
    public DummyResolver(InputData metaData) {
        super(metaData);
        rowNumber = 0;
    }
    @Override
    public List<OneField> getFields(OneRow row) throws Exception {
        /* break up the row into fields */
        List<OneField> output = new LinkedList<OneField>();
        String[] fields = ((String) row.getData()).split(",");
        output.add(new OneField(INTEGER.getOID() /* type */, Integer.parseInt(fields[0]) /* value
*/));
        output.add(new OneField(VARCHAR.getOID(), fields[1]));
        output.add(new OneField(INTEGER.getOID(), Integer.parseInt(fields[2])));
        return output;
    }
    @Override
    public OneRow setFields(List<OneField> record) throws Exception {
        /* should read inputStream row by row */
        return rowNumber > 5
                ? null
                : new OneRow(null, "row number " + rowNumber++);
    }
}
```

Dummy Analyzer

```
import com.pivotal.pxf.api.AnalyzerStats;
import com.pivotal.pxf.api.ReadAccessor;
import com.pivotal.pxf.api.Analyzer;
import com.pivotal.pxf.api.utilities.InputData;

/*
 * Class that defines getting statistics for ANALYZE.
 * getEstimatedStats returns statistics for a given path
 * (block size, number of blocks, number of tuples).
 * Used when calling ANALYZE on a GPXF external table,
 * to get table's statistics that are used by the optimizer to plan queries.
 * Dummy implementation, for documentation
 */
public class DummyAnalyzer extends Analyzer {
    public DummyAnalyzer(InputData metaData) {
        super(metaData);
    }

    /*
     * path is a data source URI that can appear as a file name, a directory name or a wildcard
     * returns the data statistics in json format
     */
    @Override
    public AnalyzerStats getEstimatedStats(String data) throws Exception {
        return new AnalyzerStats(160000 /* disk block size in bytes */,
                3 /* number of disk blocks */,
                6 /* total number of rows */);
    }
}
```

Usage Example

```
psql=# CREATE EXTERNAL TABLE dummy_tbl (int1 integer, word text, int2 integer)
location
('pxf://localhost:50070/dummy_location?FRAGMENTER=DummyFragmenter&ACCESSOR=DummyAccessor&RESOLVER=Dum
format'custom' (formatter = 'pxfwritable_import');

CREATE EXTERNAL TABLE
psql=# SELECT * FROM dummy_tbl;
int1 | word | int2
------+------+------
0 | fragment1 | 0
1 | fragment1 | 0
0 | fragment2 | 0
1 | fragment2 | 0
0 | fragment3 | 0
1 | fragment3 | 0
(6 rows)
psql=# CREATE WRITABLE EXTERNAL TABLE dummy_tbl_write (int1 integer, word text, int2 integer)
location
('pxf://localhost:50070/dummy_location?ACCESSOR=DummyAccessor&RESOLVER=DummyResolver')
format'custom' (formatter = 'pxfwritable_import');

CREATE EXTERNAL TABLE
psql=# INSERT INTO dummy_tbl_write VALUES (1, 'a', 11), (2, 'b', 22);
INSERT 0 2
```

Configuration Files

This section contains sample environment variable files for HDFS, HIVE, and HBase:

hadoop-env.sh

You can use this file to configure the following types of configurations:

- HDFS only
- HDFS and HBase
- HDFS, HBase, and Hive

HDFS only

```
export GPHD_ROOT=/usr/lib/gphd
export HADOOP_CLASSPATH=\
$GPHD_ROOT/pxf/pxf-core.jar:\
$GPHD_ROOT/pxf/pxf-api.jar:\
$GPHD_ROOT/publicstage:\
```

HDFS and HBase

```
export GPHD_ROOT=/usr/lib/gphd
export HADOOP_CLASSPATH=\
$GPHD_ROOT/pxf/pxf-core.jar:\
$GPHD_ROOT/pxf/pxf-api.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/zookeeper/zookeeper-3.4.5-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-common-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-protocol-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-client-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-thrift-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/htrace-core-2.01.jar:\
/etc/gphd/hbase/conf:\
```

HDFS, HBase, and Hive

```
export GPHD_ROOT=/usr/lib/gphd
export HADOOP_CLASSPATH=\
$GPHD_ROOT/pxf/pxf-core.jar:\
$GPHD_ROOT/pxf/pxf-api.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/zookeeper/zookeeper-3.4.5-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-common-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-protocol-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-client-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/hbase-thrift-0.96.0-hadoop2-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hbase/lib/htrace-core-2.01.jar:\
/etc/gphd/hbase/conf:\
$GPHD_ROOT/hive/lib/hive-service-0.12.0-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hive/lib/hive-metastore-0.12.0-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hive/lib/hive-common-0.12.0-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hive/lib/hive-exec-0.12.0-gphd-3.0.0.0.jar:\
$GPHD_ROOT/hive/lib/libfb303-0.9.0.jar:\
$GPHD_ROOT/hive/lib/libthrift-0.9.0.jar:\
```

hbase-site.xml

You can use this file to configure the following types of configurations:

- HBase
- HDFS, HBase and Hive

HBase

The Java Class path requires the PXF JAR filse. *hbase-site.xml* needs to be configured to match the hbase settings on all nodes (Namenode and Datanodes).

Credentials for Remote Services

Credentials for remote services is allowing a PXF plugin to access a remote service that requires credentials.

In Hawq

For this we implemented two GUCs in hawq:

1. pxf_remote_service_login - a string of characters detailing information regarding login (i.e. user name)
2. pxf_remote_service_secret - a string of characters detailing information that is considered secret (i.e. password)

Currently, we store the contents of the two in memory without any security for the whole session. Leaving the session will insecurely drop the GUCs' contents.

These GUCs are temporary and could be marked deprecated soon in favor of a complete solution for managing credentials for remote services in PXF.

In a PXF Plugin

As a PXF plugin, the content of the two GUCs is available through the following InputData API functions:

1. string getLogin()
2. string getSecret()

Both functions will return 'null' if the corresponding Hawq GUC was set to an empty string or wasn't set at all.