

# Pivotal Extension Framework

Version 2.1

## Installation and User Guide

Rev: 02

# Table of Contents

1	Pivotal Extension Framework Installation and Administration - 2.1.0	5
1.1	About PXF	5
1.1.1	Prerequisites	5
1.1.2	Upgrading from GPXF	5
1.2	Installing PXF	6
1.2.1	Installing the PXF JAR File	6
1.2.2	Setting up the Java Classpath	6
1.2.3	Enabling the REST Service	7
1.2.4	Restarting the Cluster	7
1.2.5	PXF and Secure HDFS	7
1.3	Reading and Writing Data with PXF	8
1.3.1	Built-in Profiles	8
1.4	Accessing HDFS File Data with PXF	10
1.4.1	Syntax	10
1.4.2	FORMAT clause	11
1.4.3	Fragmenter	11
1.4.4	Accessor	11
1.4.5	Resolver	12
1.4.6	Additional Options	13
1.4.7	About the Public Directory	13
1.5	Accessing Hive Data with PXF	14
1.5.1	Syntax	14
1.5.2	Hive Command Line	14
1.5.3	Mapping Hive Collection Types	15
1.5.4	Partition Filtering	16
1.6	Accessing HBase Data with PXF	17
1.6.1	Syntax	18
1.6.2	Column Mapping	18
1.6.3	Row Key	18
1.7	Accessing GemFire Data with PXF	20
1.7.1	Syntax	20
1.8	Troubleshooting	20
2	Pivotal Extension Framework External Table and API Reference - 2.1.0	23
2.1	Creating an External Table	23
2.2	About the Java Class Services and Formats	23
2.2.1	Fragmenter	26
2.2.2	Accessor	28
2.2.3	Resolver	30
2.2.4	Analyzer	33
2.3	About Custom Profiles	35
2.4	About Query Filter Push-Down	36
2.4.1	Filter Availability and Ordering	36

2.4.2	Creating a Filter Builder Class	37
2.4.3	Filter Operations	37
2.4.4	Sample Implementation	40
2.4.5	Using Filters	42
2.5	Reference	43
2.5.1	External Table Examples	43
2.5.2	Plugin Examples	45
2.5.3	Configuration Files	51

Copyright © 2013 GoPivotal, Inc. All rights reserved.

GoPivotal, Inc. believes the information in this publication is accurate as of its publication date. The information is subject to change without notice. THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS." GOPIVOTAL, INC. ("Pivotal") MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any Pivotal software described in this publication requires an applicable software license.

All trademarks used herein are the property of Pivotal or their respective owners.

### **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.

# 1 Pivotal Extension Framework Installation and Administration - 2.1.0

---

## 1.1 About PXF

---

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 the *Pivotal Extension Framework API and Reference Guide*.

### 1.1.1 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 everywhere and that you have set the property `hive.metastore.uri` in the `hive-site.xml` file on the Namenode.
- HBase

### 1.1.2 Upgrading from GPXF

If you have a previous version of HAWQ, or GPXF installed, see the Pivotal ADS 1.1.3 Release Notes, for instructions about how to upgrade to HAWQ 1.1.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:
  1. Change the protocol name in the LOCATION clause from gpxf to pxf.
  2. Ensure that Fragmenter, Accessor, and Resolver are *always* specified for the table.
  3. Check that you have the new names for the Fragmenter, Accessor, and Resolver classes. See the *Pivotal Extension Framework API and Reference Guide*.
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
```

## 1.2 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 JAR 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 1.0.1 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.

### 1.2.1 Installing the PXF JAR File

Install the PXF JAR files on all nodes in the cluster:

```
sudo rpm -i pxf-2.1.0-x.rpm
```

- The *pxf-2.1.0-x.rpm* resides in the Pivotal ADS/HAWQ stack file. For example, *PADS-1.1.1-x/pxf-2.1.0-x.noarch.rpm*
- The script installs the JAR files at the default location at */usr/lib/gphd/pxf-2.1.1*. A *softlink* will be created in */usr/lib/gphd/pxf*

### 1.2.2 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.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/pxf/avro-1.5.4.jar:\
$GPHD_ROOT/pxf/avro-mapred-1.5.4.jar:\
/usr/lib/gphd/zookeeper/zookeeper.jar:\
/usr/lib/gphd/hbase/hbase.jar:\
/etc/gphd/hbase/conf:\
$GPHD_ROOT/hive/lib/hive-service-0.11.0-gphd-2.0.2.0.jar:\
$GPHD_ROOT/hive/lib/libthrift-0.9.0.jar:\
$GPHD_ROOT/hive/lib/hive-metastore-0.11.0-gphd-2.0.2.0.jar:\
$GPHD_ROOT/hive/lib/libfb303-0.9.0.jar:\
$GPHD_ROOT/hive/lib/hive-common-0.11.0-gphd-2.0.2.0.jar:\
$GPHD_ROOT/hive/lib/hive-exec-0.11.0-gphd-2.0.2.0.jar:\
```

This adds the following to the PHD 1.0.2 Java classpath:

- PXF JAR, and staging dir */usr/lib/gphd/publicstage* (see *"About the Public Directory"* below).
- HBase JAR, configuration dir, zookeeper.
- Hive: hive-service, libthrift, hive-metastore, libfb303, hive-common, hive-exec.
- Avro: avro, avro-mapred JARs for Avro files and serialization support.

## 1.2.3 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>
```

## 1.2.4 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"
```

## 1.2.5 PXF and Secure HDFS

Currently, PXF does not support secure HDFS. If you enable secure HDFS, PXF will stop serving queries. Disabling secure HDFS automatically restores PXF functionality.

You may see the following message if you try to use a PXF table on a secure HDFS cluster

The following error message will appear when trying to use a PXF table on a secured cluster:

```
ERROR: remote component error (403) from '192.168.32.129:50070': PXF is not supported with  
Secured HDFS (libcurl.c:714)
```

## 1.3 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 [Pivotal Extension Framework 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)

### 1.3.1 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.	HdfsDataFragmenter	LineBreakAccessor	StringPassResol



Profile	Description	Fragmenter	Accessor	Resolver
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.	HdfsDataFragmenter	QuotedLineBreakAccessor	StringPassResol
Hive	Use this when connected to Hive.	HiveDataFragmenter	HiveAccessor	HiveResolver
HBase	Use this when connected to an HBase data store engine.	HBaseDataFragmenter	HBaseAccessor	HBaseResolver
Avro	Reading Avro files (i.e fileName.avro).	HdfsDataFragmenter	AvroFileAccessor	AvroResolver
GemFireXD	Use this when connected connected to GemFireXD	GemFireXDFragmenter	GemFireXDAccessor	GemFireXDResc

## 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>CustomProfileFragmenter</fragmenter>
    <accessor>CustomProfileAccessor</accessor>
    <customPlugin1>MyCustomPluginValue1</customPlugin1>
    <customPlugin2>MyCustomPluginValue2</customPlugin2>
  </plugins>
</profile>
```

## 1.4 Accessing HDFS File Data with PXF

### Notes

- Pivotal recommends that you test PXF on HDFS before connecting to Hive or HBase.
- Currently, PXF does not support secure HDFS. If you enable secure HDFS, PXF stops serving queries. Disabling secure HDFS automatically restores PXF functionality.

You may see the following message if you try to use a PXF table on a secure HDFS cluster:

```
ERROR: remote component error (403) from '192.168.32.129:50070': PXF is not supported
with Secured HDFS (libcurl.c:714)
```

The syntax for accessing an HDFS file is as follows:

### 1.4.1 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>);

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 [Pivotal Extension Framework API and Reference Guide](#).

## 1.4.2 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).

## 1.4.3 Fragmenter

Always use either [*HdfsTextSimple* / *HdfsTextMulti*]Profile or an *HdfsDataFragmenter* for HDFS file data.

**Note**

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

## 1.4.4 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	Comment
Plain Text delimited	LineBreakAccessor	FORMAT 'TEXT' (<format param list>)	

File Type	Accessor	FORMAT clause	Comment
Plain Text CSV	LineBreakAccessor	FORMAT 'CSV' (<format param list>)	<ul style="list-style-type: none"> <li>LineBreakAccessor is parallel and faster.</li> <li>Use if each logical data row is a physical data li</li> </ul>
	QuotedLineBreakAccessor		<ul style="list-style-type: none"> <li>QuotedLineBreakAcce</li> <li>is slower and non paral</li> <li>Use if the data includes embedded (quoted) linefeed characters.</li> </ul>
SequenceFile	SequenceFileAccessor	FORMAT 'CUSTOM' (formatter='pxfwritable_import')	
AvroFile	AvroFileAccessor	FORMAT 'CUSTOM' (formatter='pxfwritable_import')	

## 1.4.5 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	Direction
Avro	AvroResolver	<ul style="list-style-type: none"> <li>Avro files include the record schema, Avro serialization can be used in other file types (e.g, Sequence File).</li> <li>For Avro serialized records outside an Avro file, include a schema file name (.avsc) in the url under the optional <i>Schema-Data</i> option.</li> <li>The schema file name must exist in the public stage directory</li> </ul>	Deserialize Only (Read)

Record Serialization	Resolver	Comments	Direction
Java Writable	WritableResolver	<ul style="list-style-type: none"> <li>Include the name of the Java class that uses Writable serialization in the url under the optional <i>Schema-Data</i>.</li> <li>The name must exist in the public stage directory.</li> </ul>	Deserialize and Serialize (Read + Write)
None (plain text)	StringPassResolver	Does not serialize plain text records. The database parses plain records. Passes records as they are.	Deserialize and Serialize (Read + Write)

## 1.4.6 Additional Options

Option Name	Description
COMPRESSION_CODEC	<ul style="list-style-type: none"> <li>Useful for WRITABLE PXF tables.</li> <li>Specifies the c ompression codec class name and implements the <code>org.apache.hadoop.io.compress.CompressionCodec</code> interface. If the class name is not defined, Pivotal recommends that you do not compress.</li> <li>Some valid values are <code>org.apache.hadoop.io.compress.DefaultCodec</code>, <code>org.apache.hadoop.io.compress.GzipCodec</code>.</li> <li>Note: <code>org.apache.hadoop.io.compress.BZip2Codec</code> is not supported.</li> <li>This option has no default value.</li> </ul>
SCHEMA-DATA	<p>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).</p> <p>This option has no default value.</p>
<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

## 1.4.7 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.

## 1.5 Accessing Hive Data with PXF



### NOTE

Check the following before adding PXF support on Hive:

- 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.

### 1.5.1 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.

### 1.5.2 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;
```

## 1.5.3 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');
```

## 1.5.4 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 table *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 appear 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'
```

## 1.6 Accessing HBase Data with PXF



### NOTE

Before using PXF HBase plugin, verify the following:

- 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.

## 1.6.1 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>;
```

## 1.6.2 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.

## 1.6.3 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= <a href="#">cf1:saleid</a>
sales	cmts= <a href="#">cf8:comments</a>

**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.

## 1.7 Accessing GemFire Data with PXF

### NOTE

Before using PXF GemFire plugin, verify the following:

- That you have installed the *gfoxd 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.

### 1.7.1 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')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');

SELECT * FROM <pxf tblname>;
```

## 1.8 Troubleshooting

The following table describes some common errors while using PXF:

**Table: PXF Errors and Explanation**

Error	Common Explanation
<i>ERROR: invalid URI pxf://localhost:50070/demo/file1: missing options section</i>	<i>LOCATION</i> doesn't include any options after the file name: <code>&lt;path&gt;?&lt;key&gt;=&lt;value&gt;&amp;&lt;key&gt;=&lt;value&gt;..</code>
<i>ERROR: protocol "pxf" does not exist</i>	HAWQ is not compiled with PXF protocol, currently this requires a special HAWQ version, GPSQL.
<i>ERROR: remote component error: 0</i>  DETAIL: There is no pxf servlet listening on the host and port specified in the external table url.	Wrong server or port or the service is not started.
<i>ERROR: Missing FRAGMENTER option in the pxf uri: pxf://localhost:50070/demo/file1?a=a</i>	No <i>FRAGMENTER</i> option was specified in <i>LOCATION</i> .

<p><i>ERROR: remote component error: 500</i></p> <p>DETAIL: Problem accessing /gpdb/v&lt;X&gt;/Fragmenter/getFragments. Reason: org.apache.hadoop.mapred.InvalidInputException: Input path does not exist: <a href="hdfs://0.0.0.0:8020/demo/file1">hdfs://0.0.0.0:8020/demo/file1</a></p>	File or pattern given in <i>LOCATION</i> doesn't exist on specified path.
<p><i>ERROR: remote component error: 500</i></p> <p>DETAIL: Problem accessing /gpdb/v&lt;X&gt;/Fragmenter/getFragments. Reason: org.apache.hadoop.mapred.InvalidInputException: Input Pattern <a href="hdfs://0.0.0.0:8020/demo/file*">hdfs://0.0.0.0:8020/demo/file*</a> matches 0 files</p>	File or pattern given in <i>LOCATION</i> doesn't exist on specified path.
<p><i>ERROR: remote component error: 500 PXF not correctly installed in CLASSPATH</i></p>	Cannot find PXF Jar
<p><i>ERROR: GPHD component not found</i></p>	Either the required data node does not exist or REST service on data node is not started or PXF JAR cannot be found on data node
<p><i>ERROR: remote component error (403): PXF is not supported with Secured HDFS</i></p>	PXF cannot be used when secure HDFS is enabled.
<b>HBase Specific Errors</b>	
<p><i>ERROR: remote component error: 500</i></p> <p>DETAIL: Problem accessing /gpdb/v&lt;X&gt;/Fragmenter/getFragments. Reason: org.apache.hadoop.hbase.client.NoServerForRegionException: Unable to find region for t1,,999999999999999 after 10 tries.</p>	HBase service is down, probably HRegionServer.
<p><i>ERROR: remote component error: 500</i></p> <p>DETAIL: Problem accessing /gpdb/v&lt;X&gt;/Fragmenter/getFragments. Reason: org.apache.hadoop.hbase.TableNotFoundException: nosuch</p>	HBase cannot find the requested table
<p><i>ERROR: remote component error: 500 (/HTTP/1.1 500 Internal Server Error</i></p>	PXF cannot find a required JAR file, probably HBase's
<p><i>ERROR: remote component error: 500 (/HTTP/1.1 500 org/apache/zookeeper/KeeperException</i></p>	PXF cannot find Zookeeper's JAR
<p><i>ERROR: remote component error: 500 (/HTTP/1.1 500 Illegal HBase column name name</i></p>	Required HBase table does not contain the requested column
<b>Hive Specific Errors</b>	

<i>ERROR: remote component error: 500</i>  DETAIL: Problem accessing <i>/gpdb/v&lt;X&gt;/Fragmenter/getFragments. Reason:</i> <i>java.net.ConnectException: Connection refused</i>	Hive Metastore service is down.
<i>ERROR: remote component error: 500</i>  DETAIL: Problem accessing <i>/gpdb/v&lt;X&gt;/Fragmenter/getFragments. Reason:</i> <i>NoSuchObjectException(message:default.players table not found)</i>	Table doesn't exist in Hive.

## 2 Pivotal Extension Framework External Table and API Reference - 2.1.0

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.

### 2.1 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=FragmenterForX&ACCESSOR=AccessorForX&RESOLVER=ResolverForX&ANALYZER=AnalyzerForX&
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*.

### 2.2 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: 'FragmenterForX'
ACCESSOR: 'AccessorForX'
RESOLVER: 'ResolverForX'
```

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

The Java classes can be described as follows:

```
/*
 * 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);
}
```

Attributes are available through the *com.pivotal.pxf.utilities.InputData* class. The following example shows how *inputData.getProperty('USERINFO1')* returns *optional\_info*.

```
/*
 * Common configuration of all MetaData classes
 * Provides read-only access to common parameters supplied using system properties
 */
public class InputData
{
    /* Constructor of InputData
     * Parses greenplum.* configuration variables
     */
    public InputData(Map<String, String> paramsMap);

    /*
     * Expose the parameters map
     */
}
```



```
public Map<String, String> getParametersMap();

/* Copy constructor 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 a data fragment
    */
    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 path to the resource required
     * (might be a file path or a table name)
     */
    public String path();

    /** 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 Schema GetAvroFileSchema();

    /**
     * Returns table name
     */
    public String tableName();

    /**
     * The avroSchema is set from the outside by the AvroFileAccessor
     */
    public void SetAvroFileSchema(Schema theAvroSchema);

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

## 2.2.1 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 1.x and higher:

```
HdfsDataFragmenter  
HBaseDataFragmenter  
HiveDataFragmenter
```

The fragmenter uses *com.pivotal.pxf.fragmenters.FragmentsOutput* to return information:

```
/*  
 * Class holding information about fragments (FragmentInfo)  
 */  
public class FragmentsOutput  
{  
    public FragmentsOutput();  
    public void addFragment(String sourceName, String[] hosts);  
    public void addFragment(String sourceName, String[] hosts, String userData);  
    public List<FragmentInfo> getFragments();  
}
```

Any Fragmenter class needs to extend *com.pivotal.pxf.fragmenters.Fragmenter*.

## **com.pivotal.pxf.fragmenters.Fragmenter**

```

/*
 * 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 FragmentsOutput fragments;

    public Fragmenter(InputData metaData)
    {
        super(metaData);
        fragments = new FragmentsOutput();
    }

    /*
     * returns the data fragments
     */
    public abstract FragmentsOutput 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.

## 2.2.2 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
<i>HdfsAtomicDataAccessor</i>	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

<i>HdfsSplittableDataAccessor</i>	Base class for accessing HDFS files using <i>RecordReaders</i> .  <div> LineBreakAccessor - Accessor for TEXT files (replaced the deprecated TextFileAccessor, LineReaderAccessor)  AvroFileAccessor - Accessor for Avro files </div>
HiveAccessor	Accessor for Hive tables
HBaseAccessor	Accessor for HBase tables

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

- *com.pivotal.pxf.accessors.IReadAccessor*
- *com.pivotal.pxf.accessors.IWriteAccessor*

```

/*
 * 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 IReadAccessor
{
    public boolean openForRead() throws Exception;
    public OneRow readNextObject() throws Exception;
    public void closeForRead() throws Exception;
}

/*
 * 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 IWriteAccessor
{
    public boolean openForWrite() throws Exception;
    public OneRow writeNextObject(OneRow onerow) throws Exception;
    public void closeForWrite() throws Exception;
}

```

The Accessor calls *openForRead()* to read existing data. After reading the data, it calls *closeForRead()*. *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.format.OneRow:

```
/*
 * 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()
}
```

## 2.2.3 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
<i>StringPassResolver</i>	<p>Supports:</p> <pre>GPDBWritable VARCHAR</pre> <p><i>StringPassResolver</i> replaced the deprecated <i>TextResolver</i>. It passes whole records (composed of any data types) as strings without parsing them</p>
<i>WritableResolver</i>	<p>Resolver for custom Hadoop Writable implementations. Custom class can be specified with the schema <code>{{,}}</code> and supports the following:</p> <pre>GPDBWritable.BOOLEAN GPDBWritable.INTEGER GPDBWritable.BIGINT GPDBWritable.REAL GPDBWritable.FLOAT8 GPDBWritable.VARCHAR GPDBWritable.BYTEA</pre>
<i>AvroResolver</i>	Supports the same field objects as <i>WritableResolver</i> .
<i>HBaseResolver</i>	<p>Supports the same field objects as <i>WritableResolver</i> and also supports the following:</p> <pre>GPDBWritable.SMALLINT GPDBWritable.NUMERIC GPDBWritable.TEXT GPDBWritable.BPCHAR GPDBWritable.TIMESTAMP</pre>
<i>HiveResolver</i>	<p>Supports the same field objects as <i>WritableResolver</i> and also supports the following:</p> <pre>GPDBWritable.SMALLINT GPDBWritable.TEXT GPDBWritable.TIMESTAMP</pre>

The class needs to extend the `com.pivotal.pxf.resolvers.Plugin` class, and implement one or both interfaces:

- *com.pxf.resolvers.IReadResolver*
- *com.pxf.resolvers.IWriteResolver*

```
/*
 * 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 IReadResolver
{
    public List<OneField> getFields(OneRow row) throws Exception;
}

/*
 * 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 IWriteResolver
{
    public OneRow setFields(DataInputStream inputStream) throws Exception;
}
```



#### Notes

- *getFields* should return a list of *com.pivotal.pxf.format.OneField*, each representing a single field.
- *setFields* should return a single *OneRow* object, given an input stream.

## com.pivotal.pxf.format.OneField



```

/*
 * 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)
    {
        type = Type;
        val = Val;
    }

    public int type;
    public Object val;
}

```

The value of *Type* should follow the *com.pivotal.pxf.hadoop.io.GPDBWritable* type *enums*. *Val* is the appropriate Java class. Supported types are as follows:

**Table: Resolver supported types**

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

## 2.2.4 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.analyzers.Analyzer*.

## com.pivotal.pxf.analyzers.Analyzer

```
/*
 * Abstract class that defines getting statistics for ANALYZE.
 * GetEstimatedStats returns statistics for a given path
 * (block size, number of blocks, number of tuples (rows)).
 * 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 metaData)
    {
        super(metaData);
    }

    /**
     * path is a 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 String GetEstimatedStats(String data) throws Exception
    {
        /* Return default values */
        return DataSourceStatsInfo.dataToJSON(new DataSourceStatsInfo());
    }
}
```

*GetEstimatedStats* creates a *DataSourceStatsInfo* class, and returns the result *DataSourceStatsInfo.dataToJSON*.

## com.pivotal.pxf.analyzers.DataSourceStatsInfo

```
/*
 * DataSourceStatsInfo is a public class that represents the size
 * information of given path.
 */
public class DataSourceStatsInfo
{
    public DataSourceStatsInfo(long blockSize, long numberOfBlocks, long numberOfTuples);

    /*
     * Default values
     */
    public DataSourceStatsInfo();

    /*
     * Given a FragmentsSizeInfo, serialize it in JSON to be used as the result string for Hawq.
     An example result is as
     * follows:
     * {"PXFFilesSize":{"blockSize":67108864,"numberOfBlocks":1,"numberOfTuples":5}}
     */
    public static String dataToJSON(DataSourceStatsInfo info) throws IOException;

    /*
     * Given a size info 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(DataSourceStatsInfo info, String datapath);

    public long getBlockSize();

    public long getNumberOfBlocks();

    public long getNumberOfTuples();
}
```

## 2.3 About Custom Profiles

Administrators can add new profiles or edit the built-in profiles in

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>CustomProfileFragmenter</fragmenter>
    <accessor>CustomProfileAccessor</accessor>
    <customPlugin1>MyCustomPluginValue1</customPlugin1>
    <customPlugin2>MyCustomPluginValue2</customPlugin2>
  </plugins>
</profile>
```

## 2.4 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

### 2.4.1 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>

## 2.4.2 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.IFilterBuilder* 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 IFilterBuilder
{
    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.

## 2.4.3 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;
    }
}
```

## 2.4.4 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.IFilterBuilder
{
    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);
    ...
}
```

## 2.4.5 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...
```

## 2.5 Reference

This section contains the following information:

- External Table Samples
- Plugin Examples
- Configuration Files

### 2.5.1 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=HdfsDataFragmenter&accessor=SequenceFileA
'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?fragmenter=HBaseDataFragmenter&accessor=HBaseAccessor&resolver=HBa
'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?fragmenter=HBaseDataFragmenter&accessor=HBaseAccessor&resolver=HBa
'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?accessor=LineBreakAccessor&resolver=StringPassResol
'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=HdfsDataFragmenter&accessor=SequenceFileAccesso:
'custom' (formatter='pxfwritable_export');
```

## 2.5.2 Plugin Examples

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

### Dummy Fragmenter

```
package com.pivotal.pxf.examples;
import java.net.InetAddress;
import com.pivotal.pxf.utilities.InputData;
import com.pivotal.pxf.utilities.Plugin;
import com.pivotal.pxf.fragmenters.Fragmenter;
import com.pivotal.pxf.fragmenters.FragmentsOutput;
public class DummyFragmenter extends Fragmenter
{
    public DummyFragmenter(InputData metaData)
    {
        super(metaData);
    }
    /*
    * path is a data source name (e.g, file, dir, wildcard, table name).
    * returns the data fragments
    */

    public FragmentsOutput GetFragments() throws Exception
    {
        String localhostname = java.net.InetAddress.getLocalHost().getHostName();
        fragments.addFragment(this.inputData.path() + ".1" /* source name */,
        new String[]{localhostname} /*
        available hosts list

        */);
        fragments.addFragment(this.inputData.path() + ".2" /* source name */,
        new String[]{localhostname} /*
        available hosts list

        */);
        fragments.addFragment(this.inputData.path() + ".3" /* source name */,
        new String[]{localhostname} /*
        available hosts list

        */);
        return fragments;
    }
}
```

## Dummy Accessor

```
package com.pivotal.pxf.examples;

import com.pivotal.pxf.format.OneRow;
import com.pivotal.pxf.utilities.InputData;
import com.pivotal.pxf.utilities.Plugin;
import com.pivotal.pxf.accessors.IReadAccessor;

import com.pivotal.pxf.accessors.IWriteAccessor;
```

```
/*
 * 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 IReadAccessor
,IWriteAccessor
{
    private int rowNumber;
    private int fragmentNumber;
    private Log Log;

    public DummyAccessor(InputData metaData)
    {
        super(metaData);
        rowNumber = 0;
        fragmentNumber = 0;
        Log = LogFactory.getLog(DummyAccessor.class);
    }

    public boolean openForRead() throws Exception
    {
        /* fopen or similar */
        return true;
    }

    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 = this.inputData.getDataFragment();
        /* generate row */
        OneRow row = new OneRow(Integer.toString(fragment) + "." + Integer.toString(rowNumber),
/* key */
                                Integer.toString(rowNumber) + ",text," +
Integer.toString(fragment) /* value */);
        /* advance */
        rowNumber += 1;
        if (rowNumber == 2) {
            rowNumber = 0;
            fragmentNumber += 1;
        }
        /* return data */
        return row;
    }

    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



```
package com.pivotal.pxf.examples;
import java.util.List;
import java.util.LinkedList;

import com.pivotal.pxf.format.OneField;
import com.pivotal.pxf.format.OneRow;
import com.pivotal.pxf.utilities.InputData;
import com.pivotal.pxf.utilities.Plugin;
import com.pivotal.pxf.resolvers.IReadResolver;
import com.pivotal.pxf.resolvers.IWriteResolver;
import com.pivotal.pxf.hadoop.io.GPDBWritable;

/*
 * 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 IReadResolver, IWriteResolver
{
    private int rowNumber;

    public DummyResolver(InputData metaData)
    {
        super(metaData);
        rowNumber = 0;
    }

    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(GPDBWritable.INTEGER /* type */,Integer.parseInt(fields[0]) /*
value */));
        output.add(new OneField(GPDBWritable.VARCHAR ,fields[1]));
        output.add(new OneField(GPDBWritable.INTEGER ,Integer.parseInt(fields[2])));

        return output;
    }

    @Override
    public OneRow setFields(DataInputStream inputStream) throws Exception {
        /* should read inputStream row by row */
        if (rowNumber > 5)
            return null;
        return new OneRow(null, new String("row number " + rowNumber++));
    }
}
```

## Dummy Analyzer

```
package com.pivotal.pxf.examples;

import com.pivotal.pxf.utilities.InputData;
import com.pivotal.pxf.utilities.Plugin;
import com.pivotal.pxf.analyzers.Analyzer;
import com.pivotal.pxf.analyzers.DataSourceStatsInfo;
/*
 * 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.
 * Dummy implementation, for documentation
 */

public class DummyAnalyzer extends Analyzer
{
    public DummyAnalyzer(InputData metaData)
    {
        super(metaData);
    }
    /*
     * path is a data source name (e.g, file, dir, wildcard, table name).
     * returns the data statistics in json format
     */

    public String GetEstimatedStats(String data) throws Exception
    {
        return DataSourceStatsInfo.dataToJSON(new DataSourceStatsInfo(16 /* disk block size in bytes */,
        3 /* number of disk blocks */,
        2 /* number of rows in disk block (avg) */));
    }
}
```

## Usage Example

```

psql=# CREATE EXTERNAL TABLE dummy_tbl (int1 integer, word text, int2 integer)
location
('pxf://localhost:50070/eran_location?FRAGMENTER=com.pivotal.pxf.examples.DummyFragmenter&ACCESSOR=
format'custom' (formatter = 'pxfwritable_import');

CREATE EXTERNAL TABLE
psql=# SELECT * FROM dummy_tbl;
int1 | word | int2
-----+-----+-----
0 | text | 0
1 | text | 0
0 | text | 0
1 | text | 0
0 | text | 0
1 | text | 0
(6 rows)

psql=# CREATE WRITABLE EXTERNAL TABLE dummy_tbl_write (int1 integer, word text, int2 integer)
location
('pxf://localhost:50070/eran_location?ACCESSOR=com.pivotal.pxf.examples.DummyAccessor&RESOLVER=com
format'custom' (formatter = 'pxfwritable_import');

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

```

## 2.5.3 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=\
$HADOOP_CLASSPATH:\
$GPHD_ROOT/pxf/pxf.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/pxf/avro-1.5.4.jar:\
$GPHD_ROOT/pxf/avro-mapred-1.5.4.jar:\
$GPHD_ROOT/hadoop-mapreduce-2.0.2_alpha_gphd_2_0_1_0/
hadoop-mapreduce-client-core-2.0.2-alpha-gphd-2.0.1.0-SNAPSHOT.jar:\
```

## HDFS and HBase

```
export GPHD_ROOT=/usr/lib/gphd
export GPHD_CONF=/etc/gphd
export HADOOP_CLASSPATH=\
$HADOOP_CLASSPATH:\
$GPHD_ROOT/pxf/pxf.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/pxf/avro-1.5.4.jar:\
$GPHD_ROOT/pxf/avro-mapred-1.5.4.jar:\
$GPHD_ROOT/zookeeper/zookeeper.jar:\
$GPHD_ROOT/hbase/hbase.jar:\
$GPHD_CONF/hbase/conf:\
$GPHD_ROOT/hadoop-mapreduce-2.0.2_alpha_gphd_2_0_1_0/hadoop-mapreduce-client-core-2.0.2-alpha-gphd
```

## HDFS, HBase, and Hive

```
export GPHD_ROOT=/usr/lib/gphd
export GPHD_CONF=/etc/gphd
export HIVE_LIB_ROOT=$GPHD_ROOT/hive/lib
export HADOOP_CLASSPATH=\
$HADOOP_CLASSPATH:\
$GPHD_ROOT/pxf/pxf.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/pxf/avro-1.5.4.jar:\
$GPHD_ROOT/pxf/avro-mapred-1.5.4.jar:\
$GPHD_ROOT/zookeeper/zookeeper.jar:\
$GPHD_ROOT/hbase/hbase.jar:\
$GPHD_CONF/hbase/conf:\
$HIVE_LIB_ROOT/hive-service-0.9.1-gphd-2.0.1.jar:\
$HIVE_LIB_ROOT/libthrift-0.7.0.jar:\
$HIVE_LIB_ROOT/hive-metastore-0.9.1-gphd-2.0.1.jar:\
$HIVE_LIB_ROOT/libfb303-0.7.0.jar:\
$HIVE_LIB_ROOT/hive-common-0.9.1-gphd-2.0.1.jar:\
$HIVE_LIB_ROOT/hive-exec-0.9.1-gphd-2.0.1.jar:\
$GPHD_ROOT/hadoop-mapreduce-2.0.2_alpha_gphd_2_0_1_0/hadoop-mapreduce-client-core-2.0.2-alpha-gphd
```

## 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 file. *hbase-site.xml* needs to be configured to match the hbase settings on all nodes (Namenode and Datanodes).

```
export GPHD_ROOT=/usr/lib/gphd
export GPHD_CONF=/etc/gphd
export HADOOP_CLASSPATH=\
$HADOOP_CLASSPATH:\
$GPHD_ROOT/pxf/pxf.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/pxf/avro-1.5.4.jar:\
$GPHD_ROOT/pxf/avro-mapred-1.5.4.jar:\
$GPHD_ROOT/zookeeper/zookeeper.jar:\
$GPHD_ROOT/hbase/hbase.jar:\
```

## HDFS, HBase and Hive

```
export GPHD_ROOT=/usr/lib/gphd
export HIVE_LIB_ROOT=$GPHD_ROOT/hive/lib
export HADOOP_CLASSPATH=\
$GPHD_ROOT/pxf/pxf.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/pxf/avro-1.5.4.jar:\
$GPHD_ROOT/pxf/avro-mapred-1.5.4.jar:\
$GPHD_ROOT/zookeeper/zookeeper.jar:\
$GPHD_ROOT/hbase/hbase.jar:\
```