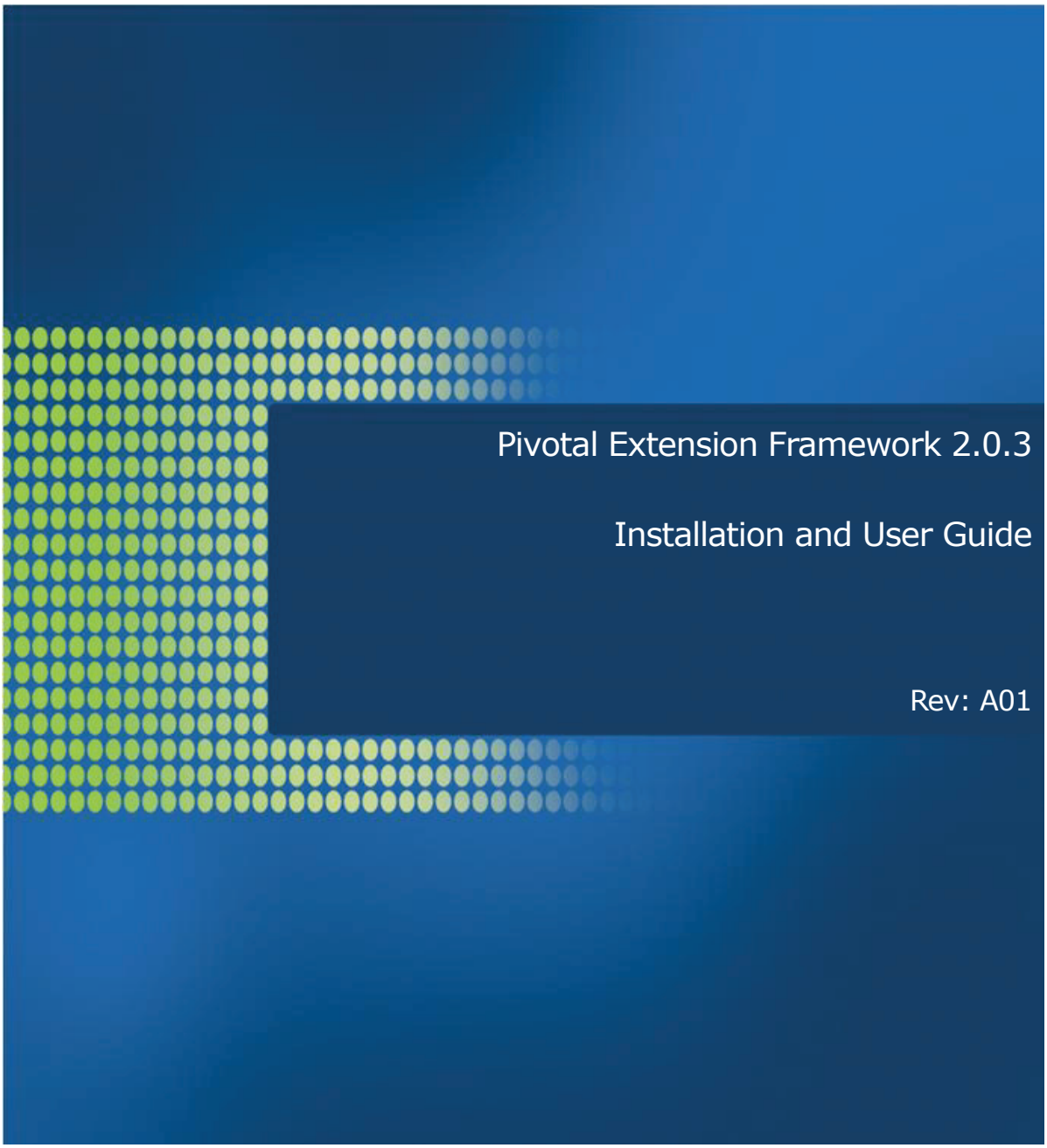
Pivotal Extension Framework 2.0.3
Installation and User Guide
Rev: A01

Pivotal Extension Framework Installation and Administration

This chapter provides information for system administrators and database superusers responsible for installing, administering, and using HAWQ with Pivotal Extension Framework (PXF).

This chapter contains the following information:

About PXF

PXF is an extensible framework that allows HAWQ to query external system data in parallel. PXF comes with a number of built-in connectors for accessing data that exists inside HDFS files, Hive tables, and HBase tables. There is also an option for the users to create their own connectors to any other type of parallel data store or processing engine. For more information about the JAVA plugins, see the *Pivotal Extension Framework API and Reference Guide* .

Prerequisites

Check that you have met the following requirements before you install PXF:

- HAWQ installed and running
- Pivotal Hadoop (PHD)
- Hadoop File System (HDFS) with REST service enabled on the Namenode and all Datanodes
- Hive installed and Hive Metastore service running (anywhere) and property *hive.matastore.uri* set in *hive-site.xml* on the Namenode (only needed if interested in PXF Hive support)
- HBase installed (only needed if interested in PXF HBase support)

Upgrading from GPXF

If you have a previous version of HAWQ, or GPXF installed, see the Pivotal ADS 1.1 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:
    a. Change the protocol name in the LOCATION clause from gpxf to pxf.
    b. Ensure that Fragmenter, Accessor, and Resolver are *always* specified for the table.
    c. 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
```

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) manager and ICM. If you used the ICM to install PXF, you can refer to the next topic, *Accessing Any System Data with PXF*.

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.

Installing the PXF JAR File

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

```
sudo rpm -i pxf-2.0.3-x.rpm
```

- The *pxf-2.0.3-x.rpm* resides in the Pivotal ADS/HAWQ stack file. For example, *PADS-1.1.1-x/pxf-2.0.3-x.noarch.rpm*
- The script installs the JAR files at the default location at */usr/lib/gphd/pxf-2.0.3*. A *softlink* 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.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.

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 Installation and Configuration Manager (ICM) to start the cluster.
2. Optionally, you can examine the classpath with the following command:

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

Accessing Data with PXF

PXF comes with a number of built-in connectors for accessing data that exists inside HDFS files, Hive tables, and HBase tables. 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
- Accessing HIVE Data with PXF
- Accessing HBASE Data with PXF

Accessing HDFS File Data with PXF

> **NOTE**
> Pivotal recommends that you test PXF on HDFS before connecting to Hive or HBase.

The syntax for accessing an HDFS file is as follows:

Syntax

```
CREATE EXTERNAL TABLE pxf_hdfs_test (id int)
LOCATION ('pxf://<name node hostname:50070>/<path to file or
directory>?Fragmenter=HdfsDataFragmenter&Accessor=<selected accessor
class>&Resolver=<selected resolver class>[&Data-Schema=<name>]'
FORMAT '[TEXT | CSV | CUSTOM]' (<formatting properties>);

SELECT * FROM pxf_hdfs_test;
```

When reading a HDFS file or a directory of files, the user should select the FORMAT and the Fragmenter, Accessor, Resolver classes based on the underlying data in the files. Currently, all are mandatory.

This topic contains and high-level description about the following information:

- FORMAT clause
- Fragmenter
- Accessor

- Resolver

> **Note**
> For more details about the API and classes, see the *Pivotal Extension Framework API and Reference Guide*.

FORMAT clause

The following formats are available to be used with any PXF connectors to read data:

- **FORMAT 'TEXT'**: to be used with plain delimited text files on HDFS.
- **FORMAT 'CSV'**: to be used with comma separated value files on HDFS.
- **FORMAT 'CUSTOM'**: to be used with anything else (binary formats). Must always be used with built-in formatter '*pxfwritable_import*'.

Fragmenter

Always use *HdfsDataFragmenter* for HDFS file data.

Note: You must include a Fragmenter in the table definition.

Accessor

The choice of an Accessor depends on the HDFS data file type. Note that it is mandatory to include an Accessor in the table definition.

File Type	Accessor	FORMAT clause	Comment
Plain Text delimited	LineReaderAccessor	FORMAT 'TEXT' (<format param list>)	
Plain Text CSV	LineReaderAccessor or QuotedLineBreakAccessor	FORMAT 'CSV' (<format param list>)	LineReaderAccessor is parallel and faster, while QuotedLineBreakAccessor is slower and non parallel. The choice depends on the actual data. If the data includes embedded (quoted) linefeed characters inside data fields then QuotedLineBreakAccessor must be used. Otherwise if each logical data row is a physical data line, LineReaderAccessor should be fine.
SequenceFile	SequenceFileAccessor	FORMAT 'CUSTOM' (formatter='pxfwritable_import')	
AvroFile	AvroFileAccessor	FORMAT 'CUSTOM' (formatter='pxfwritable_import')	

Resolver

The choice of a Resolver depends on how data records are serialized in the HDFS file.

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

Record Serialization	Resolver	Comments
Avro	AvroResolver	Note that avro serialization isn't the same as Avro file type. Avro files include the record schema. But Avro serialization could be used in other file types (e.g, Sequence File). For Avro serialized records not inside an Avro file, a schema file name (usually .avsc) must be included in the url under the optional *Schema-Data* option and exist in the public stage directory
Java Writable	WritableResolver	The name of the Java class that uses Writable serialization must be included in the url under the optional *Schema-Data* option and exist in the public stage directory
None (plain text)	StringPassResolver	Plain text records aren't serialized and their parsing is deferred to the database to do. Pass records be as they are.

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 directory is created by the RPM at the default location: */usr/lib/gphd/publicstage*

Check that all HDFS users have read permissions to HDFS services and write permissions only for specific users.

Accessing Hive Data with PXF

NOTE
Check the following before adding PXF support on Hive:

- You are running the Hive Metastore service (on any machine) and property *hive.metastore.uris* is set in *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 the same way in PXF regardless of the underlying file storage format. The PXF Hive plugins automatically detect the source table that can be:

- 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>?FRAGMENTER=HiveDataFragmenter&ACCESSOR=HiveAccessor&RESOLVER=HiveResolver')
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 a sample Hive table and load data into it 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 that correspond to the 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?FRAGMENTER=HiveDataFragmenter&ACCESSOR
=HiveAccessor&RESOLVER=HiveResolver')
FORMAT 'custom' (FORMATTER='pxfwritable_import');
```

Partition Filtering

The PXF Hive plugin uses the Hive partitioning feature and directory structure. This enables partition exclusion on the HDFS files that contain the Hive table. Using this feature reduces network traffic and I/O when running a PXF query that uses a WHERE clause to refers to a specific partition in the partitioned Hive table.

To take advantage of PXF Partition filtering push down, you need to name the partition fields in the external table. These names must be the same as those stored in the Hive table. Otherwise, PXF Partition filtering is ignored and the filtering is performed on the HAWQ side. This has performance impact.

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 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.tx
t
/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.** This matches the behavior of Hive Query Language (HiveQL) where 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?FRAGMENTER=HiveDataFragmenter&ACCESSOR=HiveAcce
ssor&RESOLVER=HiveResolver')
FORMAT 'custom' (FORMATTER='pxfwritable_import');

SELECT * FROM pxf_sales_part;
```

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.

In the following example the HAWQ query filters on the city partition *Sacramento*, but *ignores* all other partitions into *delivery_state CALIFORNIA*. 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* i s transferred for processing.

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

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

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

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.

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>?FRAGMENTER=HBaseDataFragmenter&ACCESSOR=HBaseAccessor&Resolver=HBaseResolver')
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, the PXF HBase plugin allows you to specify a 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, if you have defined an HBase table called *hbase_sales* with multiple column families and many qualifiers and want to see the *rowkey*, the qualifier *saleid* in the column family *cf1*, and the qualifier *comments* in the column family *cf8*. A resulting attribute section of the CREATE EXTERNAL TABLE appears as follows:

```
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 you can perform the following:

- Since HAWQ has a 32 character limit on attribute name size, indirect mapping helps read long HBase qualifier names.
- HAWQ attribute names are character based, while HBase qualifier names can be binary or non-printable.

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*>, an example of a lookup table is as follows:

(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>?FRAGMENTER=GemFireXDFragmenter&ACCESSOR=GemFireXDAccessor&Resolver=GemFireXDReso
lver')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');

SELECT * FROM <pxf tblname>;
```

Troubleshooting

The following table describes some common errors while using PXF:

Table: PXF Errors and Explanation

Error	Common Explanation
ERROR: invalid URI pxf://localhost:50070/demo/file1: missing options section	*LOCATION* doesn't include any options after the file name: *<path>?<key>=<value>&<key>=<value>...*
ERROR: protocol "pxf" does not exist	HAWQ is not compiled with PXF protocol, currently this requires a special HAWQ version, GPSQL.
ERROR: remote component error: 0 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.
ERROR: Missing FRAGMENTER option in the pxf uri: pxf://localhost:50070/demo/file1?a=a	No *FRAGMENTER* option was specified in *LOCATION*.
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason:*org.apache.hadoop.mapred.InvalidInputException: Input path does not exist: hdfs://0.0.0.0:8020/demo/file1*	File or pattern given in *LOCATION* doesn't exist on specified path.
ERROR: *remote component error: 500* DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *org.apache.hadoop.mapred.InvalidInputException: Input Pattern hdfs://0.0.0.0:8020/demo/file** matches 0 files	File or pattern given in *LOCATION* doesn't exist on specified path.
ERROR: remote component error: 500 PXF not correctly installed in CLASSPATH	Cannot find PXF Jar
ERROR: GPHD component not found	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
HBase Specific Errors	
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *org.apache.hadoop.hbase.client.NoServerForRegionException: Unable to find region for t1,,99999999999999* after 10 tries.	HBase service is down, probably HRegionServer.
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *org.apache.hadoop.hbase.TableNotFoundException:* nosuch	HBase cannot find the requested table
ERROR: remote component error: 500 (/HTTP/1.1 500 Internal Server Error	PXF cannot find a required JAR file, probably HBase's
ERROR: remote component error: 500 (/HTTP/1.1 500 org/apache/zookeeper/KeeperException	PXF cannot find Zookeeper's JAR
ERROR: remote component error: 500 (/HTTP/1.1 500 Illegal HBase column name name	Required HBase table does not contain the requested column
Hive Specific Errors	
ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *java.net.ConnectException:* Connection refused	Hive Metastore service is down.

ERROR: remote component error: 500 DETAIL: Problem accessing */gpdb/v<X>/Fragmenter/getFragments*. Reason: *NoSuchObjectException(message:default.players table not found)*	Table doesn't exist in Hive.

Pivotal Extension Framework 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.

This chapter contains the following information:

- Creating an External Table
- About the Java Class Services and Formats
    - Fragmenter
    - Accessor
    - Resolver
    - Analyzer
- About Query Filter Push-Down
    - Filter Availability and Ordering
    - Creating a Filter Builder Class
    - Filter Operations
        - Filter Operands
        - Column Index
        - Constant
        - Filter Object
    - Sample Implementation
    - Using Filters
- Reference
    - External Table Examples
        - Example 1
        - Example 2
        - Example 3
    - Plugin Examples
        - Dummy Fragmenter
        - Dummy Accessor
        - Dummy Resolver
        - Dummy Analyzer
        - Usage Example
    - Configuration Files
        - hadoop-env.sh
        - hbase-site.xml

Creating an External Table

When you define an *EXTERNAL TABLE*, the *LOCATION* must include the following attributes. You may need additional optional attributes depending on the service.

```
CREATE EXTERNAL TABLE ext_table (a int, b text)
LOCATION('pxf://<namenode>:<port>/some/path/and/file/name?FRAGMENTER=FragmenterForX&AC
CESSOR=AccessorForX&RESOLVER=ResolverForX&USERINFO1=optional_info&USERINFO2=more_info'
)
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.

For more information about this example, see *"About the Java Class Services and Formats".*

About the Java Class Services and Formats

The Java class names you must include in the PXF classpath are Fragmenter, Accessor, and Resolver. The Fragmenter attribute requires a value since it describes the service type. If you do not specify a value for Accessor or Resolver, the API uses the default value. You can also use a combination of classes. Also, you can reuse a previously-defined Accessor or Resolver data format.

All the attributes are passed as headers to the PXF Java service that retreves the data and converts it to the HAWQ format. Any additional inforrmation can be passed to the user-implemented services.

The example in *"Creating an External Table"* shows the following available keys and associated values:

```
FRAGMENTER: ‘FragmenterForX’
ACCESSOR: ‘AccessorForX’
RESOLVER: ‘ResolverForX’
USERINFO1: ‘optional_info’
USERINFO2: ‘more_info’
```

It also contains attributes that are passed in from the HAWQ side. All four plugins extend the *com.pivotal.pxf.utilities.Plugin* class.

The 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. This example shows how *inputData.getProperty('USERINFO1')* return s *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 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 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);
}
```

Fragmenter

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 data fragment containing a HDFS file block. This fragment includes the location of the block. If the source data is an HBase table, the Fragmenter returns a table region, including the location of the table.

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();
}
```

The Fragmenter class needs to implement the interface *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.

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. The following implementations are shipped with PXF 1.x and higher.

Table: Accessor base classes

Accessor class	Description
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`

HdfsSplittableDataAccessor	Base class for accessing HDFS files using *RecordReaders*:
	`LineReaderAccessor - Accessor for TEXT files (replaced the deprecated TextFileAccessor)`
	`AvroFileAccessor - Accessor for Avro files`
	`HiveAccessor - Accessor for Hive tables`
	`HBaseAccessor - Accessor for HBase tables`

The class needs to extend the class *com.pivotal.pxf.accessors.Accessor*:

```
/*
 * 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 stores
 * must respect this interface
 */
public abstract class Accessor extends Plugin
{
    public Accessor(InputData metaData)
    {
        super(metaData);
    }
    abstract public boolean Open() throws Exception;
    abstract public OneRow LoadNextObject() throws Exception;
    abstract public void Close() throws Exception;
}
```

The Accessor calls *Open()*. After reading, it calls *Close()*. *LoadNextObject()* returns a single record, encapsulated in a OneRow object, or null if it reaches *EOF.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()
}
```

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 *GPDBWritable* format read by HAWQ. The following implementations are shipped with PXF 1.x and higher:

Table: Resolver base classes

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

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

AvroResolver	Supports the same field objects as *WritableResolver*.

HBaseResolver	Supports the same field objects as *WritableResolver* and also supports the following:
	`GPDBWritable.SMALLINT`
	`GPDBWritable.NUMERIC`
	`GPDBWritable.TEXT`
	`GPDBWritable.BPCHAR`
	`GPDBWritable.TIMESTAMP`

HiveResolver	Supports the same field objects as *WritableResolver* and also supports the following:
	`GPDBWritable.SMALLINT`
	`GPDBWritable.TEXT`
	`GPDBWritable.TIMESTAMP`

The class needs to extend the class *com.pivotal.pxf.resolvers.Resolver*:

```
/*
 * Abstract class that defines the deserializtion of one record brought from
 * the data Accessor. Every implementation of a deserialization method
 * (e.g, Writable, Avro,...) must inherit this abstract class
 */
public abstract class Resolver extends Plugin
{
    public Resolver(InputData metaData)
    {
        super(metaData);
    }
    public abstract List<OneField> GetFields(OneRow row) throws Exception;
}
```

GetFields should return a list of *com.pivotal.pxf.format.OneField*, each representing a single field.

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. For supported types are as follows:

Table: Resolver supported types

GPDBWritable recognized OID	Field value
GPDBWritable.SMALLINT	*Short*
GPDBWritable.INTEGER	*Integer*
GPDBWritable.BIGINT	*Long*
GPDBWritable.REAL	*Float*
GPDBWritable.FLOAT8	*Double*
GPDBWritable.NUMERIC	*String ("65168746513546843216842 1")*
GPDBWritable.BOOLEAN	Boolean

GPDBWritable.VARCHAR	*GPDBWritable.BPCHAR* *GPDBWritable..TEXT - String*
GPDBWritable.BYTEA	*byte []*
GPDBWritable.TIMESTAMP	*Timestamp*

Analyzer

The Analyzer provides PXF statistical data for the HAWQ query optimizer. For detailed explanation on HAWQ statistical data gathering, see *ANALYZE* in the *Pivotal HAWQ Administrator Guide*. PXF analyzer is implemented for the HDFS files: text file, sequence file and AVRO file. For HBase tables and Hive tables Analyzer returns default values.

Notes:

- The new *boolean guc pxf_enable_stat_collection* requests statistics and the default value is *on*. Turning it off means that the statistics collected will reflect default values.
- Pivotal recommends the best practice and implement 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();
}
```

About Query Filter Push-Down

HAWQ may push all or some queries to PXF if an underlying client query includes a number of WHERE clause filters. 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 the filter evaluation conditions. This reduces data processing and reduces network traffic from the SQL engine.

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:

a. Simple Expression: <Column Index> <Operation> <Constant>
b. Compound Expression: <Filter Object> AND <Filter Object>
c. Compound Expression: <List of Filter Objects> AND <Filter Object>

This topic includes the following information:

- Creating a Filter Builder class
- Filter Operations
- Sample Implementation
- Using Filters

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/filter pushed down to PXF. Implementing this function returns some Filter object/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.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 the Accessor, the 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=
HdfsDataFragmenter&accessor=SequenceFile
Accessor&resolver=WritableResolver&schema=SaleItem')
FORMAT ‘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
('gpxf://10.76.72.26:50070/sales?fragmenter=
HBaseDataFragmenter')
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 *SEL ECT 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
('gpxf://10.76.72.26:8080/sales?fragmenter=
HBaseDataFragmenter')
FORMAT ‘custom’ (formatter='pxfwritable_import');
```

Plugin Examples

This section contains sample dummy implantations 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.Accessor;

public class DummyAccessor extends Accessor
{
private int rowNumber;
private int fragmentNumber;
```

```
public DummyAccessor(InputData metaData)
{
super(metaData);
rowNumber = 0;
fragmentNumber = 0;

}

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

}

public OneRow LoadNextObject() throws Exception
{
/* return next row */
/* 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 Close() 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.Resolver;
import com.pivotal.pxf.hadoop.io.GPDBWritable;
/*
* Abstract 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 Resolver
{
public DummyResolver(InputData metaData)
{
super(metaData);
}

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;
}
}
```

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 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 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.DummyFragmen
ter&ACCESSOR=com.pivotal.pxf.examples.DummyAccessor&RESOLVER=com.pivotal.pxf.examples.
DummyResolver&ANALYZER=com.pivotal.pxf.examples.DummyAnalyzer') 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)
```

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/gpxf/gpxf.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/gpxf/avro-1.5.4.jar:\
$GPHD_ROOT/gpxf/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/gpxf/gpxf.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/gpxf/avro-1.5.4.jar:\
$GPHD_ROOT/gpxf/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-2.0.1.0-SNAPSHOT.jar:\
```

HDFS, HBase, and Hive

```
export GPHD_ROOT=/usr/lib/gphd
export GPHD_CONF=/etc/gphd
export HIVELIB_ROOT=$GPHD_ROOT/hive/lib
export HADOOP_CLASSPATH=\
$HADOOP_CLASSPATH:\
$GPHD_ROOT/gpxf/gpxf.jar:\
$GPHD_ROOT/publicstage:\
$GPHD_ROOT/gpxf/avro-1.5.4.jar:\
$GPHD_ROOT/gpxf/avro-mapred-1.5.4.jar:\
$GPHD_ROOT/zookeeper/zookeeper.jar:\
$GPHD_ROOT/hbase/hbase.jar:\
$GPHD_CONF/hbase/conf:\
$HIVELIB_ROOT/hive-service-0.9.1-gphd-2.0.1.jar:\
$HIVELIB_ROOT/libthrift-0.7.0.jar:\
$HIVELIB_ROOT/hive-metastore-0.9.1-gphd-2.0.1.jar:\
$HIVELIB_ROOT/libfb303-0.7.0.jar:\
$HIVELIB_ROOT/hive-common-0.9.1-gphd-2.0.1.jar:\
$HIVELIB_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-2.0.1.0-SNAPSHOT.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 GPXF JAR file. hbase-site.xml needs to be configured to match the hbase settings on all nodes (name node and data nodes).

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