VMware Tanzu Greenplum Platform Extension Framework v5.14 Documentation

VMware Tanzu Greenplum Platform Extension Framework 5.14

vmware®

You can find the most up-to-date technical documentation on the VMware website at:
https://docs.vmware.com/

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

Pivotal Greenplum® Platform Extension Framework (PXF)

Revised: 2020-07-07

The Greenplum Platform Extension Framework (PXF) provides parallel, high throughput data access and federated queries across heterogeneous data sources via built-in connectors that map a Greenplum Database external table definition to an external data source. PXF has its roots in the Apache HAWQ project.

- Release Notes
- Installing PXF
- Uninstalling PXF
- Upgrading PXF
- Overview of PXF
- Introduction to PXF

  This topic introduces PXF concepts and usage.
- Administering PXF

  This set of topics details the administration of PXF including configuration, initialization, and management procedures.
- Accessing Hadoop with PXF

  This set of topics describe the PXF Hadoop connectors, the data types they support, and the profiles that you can use to read from and write to HDFS.
- Accessing Azure, Google Cloud Storage, Minio, and S3 Object Stores with PXF

  This set of topics describe the PXF object storage connectors, the data types they support, and the profiles that you can use to read data from and write data to the object stores.
- Accessing an SQL Database with PXF (JDBC)

  This topic describes how to use the PXF JDBC connector to read from and write to an external SQL database such as Postgres or MySQL.
- Troubleshooting PXF

  This topic details the service- and database- level logging configuration procedures for PXF. It also identifies some common PXF errors and describes how to address PXF memory issues.
- PXF Utility Reference

  The PXF utility reference.

Pivotal Greenplum Platform Extension Framework 5.x Release Notes

The Pivotal Greenplum Platform Extension Framework (PXF) is included in the Pivotal Greenplum Database distribution in Greenplum versions including and older than 5.28 and 6.9. PXF for Redhat/CentOS and Oracle Enterprise Linux is updated and distributed independently of Greenplum Database starting with PXF version 5.13.0. You may need to download and install the PXF package to obtain the most recent version of this component.

Supported Platforms

The independent PXF distribution is compatible with these operating system platform and Greenplum versions:

OS Version	Greenplum Version
RHEL 7.x, CentOS 7.x	5.21.2+, 6.x
RHEL 6.x, CentOS 6.x	5.21.2+
OEL 7.x	6.x

Starting in 6.x, Greenplum does not bundle `cURL` and instead loads the system-provided library. PXF requires `cURL` version 7.29.0 or newer. The officially-supported `cURL` for the CentOS 6.x and Red Hat Enterprise Linux 6.x operating systems is version 7.19.*. Greenplum Database 6 does not support running PXF on CentOS 6.x or RHEL 6.x due to this limitation.

PXF is compatible with these Java and Hadoop component versions:

PXF Version	Java Versions	Hadoop Versions	Hive Server Versions	HBase Server Version
5.14, 5.13	8, 11	2.x, 3.1+	1.x, 2.x, 3.1+	1.3.2

Release 5.14.0

Release Date: July 7, 2020

New and Changed Features

PXF 5.14.0 includes these new and changed features:

- PXF supports the `deflate` and `snappy` compression codecs when writing Avro data to an external data store. By default, PXF now compresses all Avro data with the `deflate` codec before writing it to the external store.
- Before writing Avro data, PXF converts `smallint`-type columns to the `int` data type. You must specify an `int`-type column in an external table definition to read this data.

Resolved Issues

PXF 5.14.0 resolves these issues:

Issue #	Summary
30708	PXF can now compress Avro data before writing it to an external data store.
30671	PXF fixes an issue where it did not correctly handle writing Avro data when the external table definition included a `smallint` type column.

Release 5.13.0

Release Date: June 30, 2020

New and Changed Features

PXF 5.13.0 includes these new and changed features since PXF 5.12.0:

- PXF 5.13.0 is the first standalone release of PXF for RedHat/CentOS that is distributed separately from Greenplum Database.

Resolved Issues

PXF 5.13.0 resolves these issues:

Issue #	Summary
364	PXF fixes an issue where it did not correctly read from an external table when the `LOCATION` clause included back-quoted text.
30640	The use of the `pxf.service.user.name` property was not clear. The PXF `pxf-site.xml` template file now includes an enhanced description of the property.

Deprecated Features

Deprecated features may be removed in a future major release of PXF. PXF version 5.x deprecates:

- The `PXF_USER_IMPERSONATION`, `PXF_PRINCIPAL`, and `PXF_KEYTAB` settings in the `pxf-env.sh` file. You can use the `pxf-site.xml` file to configure Kerberos and impersonation settings for your new Hadoop server configurations (deprecated since PXF version 5.10.0).
- The `pxf.impersonation.jdbc` property setting in the `jdbc-site.xml` file. You can use the `pxf.service.user.impersonation` property to configure user impersonation for a new JDBC server configuration (deprecated since PXF version 5.10.0).
- The HDFS profile names for the Text, Avro, JSON, Parquet, and SequenceFile data formats (deprecated since PXF version 5.0.1). Refer to Connectors, Data Formats, and Profiles in the PXF Hadoop documentation for more information.

Known Issues and Limitations

PXF 5.x has these known issues and limitations:

Issue #	Description
168957894	The PXF Hive Connector does not support using the `Hive*` profiles to access Hive transactional tables. **Workaround**: Use the PXF JDBC Connector to access Hive.

Installing PXF

The Greenplum Platform Extension Framework (PXF) for Greenplum Database 5.21.2+ and 6.x for CentOS 7.x and RHEL 7.x platforms is available from a separately-downloadable package on Pivotal Network.

The PXF download package is an `.rpm` file that installs PXF libraries, executables, and script files on a Greenplum Database host.

When you install PXF, you will:

1. Satisfy the prerequisites.
2. Download the PXF package.
3. Install the PXF `.rpm` on every host in your Greenplum Database cluster.
4. Check out Next Steps for post-install topics.

Prerequisites

The recommended deployment model is to install PXF on all Greenplum Database hosts. Before you install PXF, ensure that you meet the following prerequisites:

- Greenplum version 5.21.2 or later or 6.x is installed in the cluster.
- You have access to all hosts (master, standby master, and segment hosts) in your Greenplum Database cluster.
- You must be an operating system superuser, or have `sudo` or other privileges, to install the PXF package in the default location `/usr/local/pxf-gp<greenplum-major-version>`. Or, you can choose to install the package into a directory in which you do have access by specifying the `--prefix` option to the install command.
- You have installed Java 8 or 11 on all Greenplum Database hosts as described in Installing Java for PXF.
- You can identify the operating system user that will own the PXF installation. This user must be the same user that owns the Greenplum Database installation, or a user that has write privileges to the Greenplum Database installation directory.
- If you have previously configured and are using PXF in your Greenplum installation:
    1. Identify and note the current PXF version number.
    2. Stop PXF as described in Stopping PXF.

If this is your first installation of a PXF package, and the `$GPHOME/pxf` directory exists in your Greenplum installation, you may choose to remove the directory on all Greenplum hosts **after** you confirm that you have installed and configured PXF correctly and that it is working as expected.

If you choose to remove this directory, you may encounter `warning: <pxf-filename>: remove failed: No such file or directory` messages when you upgrade Greenplum. You can ignore these warnings for PXF files.

Downloading the PXF Package

PXF is available as a separate download for Greenplum Database 5.x or 6.x for CentOS 7.x and RHEL 7.x platforms from Pivotal Network:

1. Download the package by navigating to Pivotal Network and locating and selecting the *Release Download* directory named *Greenplum Platform Extension Framework*.

   The format of the PXF download filename is `pxf-gp<greenplum-major-version>-<pxf-version>.<platform>.<file_type>`. For example:

   ```
   pxf-gp6-5.13.0-1.el7.x86_64.rpm
   ```

2. Make note of the directory to which the file was downloaded.

Installing the PXF Package

You must install the PXF package on the Greenplum Database master and standby master hosts, and on each segment host.

If you installed an older version of the PXF package on your hosts, installing a newer package removes the existing PXF installation, and installs the new version.

The install procedure follows:

1. Locate the installer file that you downloaded from Pivotal Network.
2. Create a text file that lists your Greenplum Database standby master host and segment hosts, one host name per line. For example, a file named `gphostfile` may include:

   ```
   gpmaster
   mstandby
   seghost1
   seghost2
   seghost3
   ```

3. Install the package on each Greenplum Database host using your package management utility. If a previous installation of PXF exists for the same Greenplum version, the files and runtime directories from the older version are removed before the current package is installed.

   To install PXF into the default location on all Greenplum hosts:

   1. Install the package. For example, if you are installing PXF for Greenplum 6 on a CentOS 7 system:

      ```
      gphost$ gpssh -e -v -f gphostfile sudo rpm -Uvh pxf-gp6-5.13.0-1.el7.x86_64.rpm
      ```

      When you install the PXF `.rpm` package to the default location, PXF is installed to `/usr/local/pxf-gp<greenplum-major-version>`.

2. Set the ownership and permissions of the PXF installation files to enable access by the `gpadmin` user:

```
gphost$ gpssh -e -v -f gphostfile sudo chown -R gpadmin:gpadmin /usr/local/pxf-gp*
```

To install PXF into a custom location on all Greenplum hosts as a non-root user:

```
gpadmin@gphost$ gpssh -e -v -f gphostfile rpm -Uvh pxf-gp6-5.13.0-1.el7.x86_64.rpm --prefix <install-location>
```

4. (Optional) Add the PXF `bin` directory to the PXF owner's `$PATH`. For example, if you installed PXF for Greenplum 6 in the default location, you could add the following text to the `.bash_profile` shell initialization script for the `gpadmin` user:

```
export PATH=$PATH:/usr/local/pxf-gp6/bin
```

Be sure to remove any previously-added `$PATH` entries for PXF in `$GPHOME/pxf/bin`.

Next Steps

PXF is not active after installation. You must explicitly initialize and start the PXF server before you can use PXF.

- See About the PXF Installation and Configuration Directories for a list and description of important PXF files and directories.
- If this is your first time using PXF, review Configuring PXF for a description of the initialization and configuration procedures that you must perform before you can use PXF.
- If you installed the PXF `.rpm` into a Greenplum cluster in which you had already configured and were using PXF, you may be required to perform some upgrade actions. Recall the original version of PXF (before you installed the `.rpm`), and perform Step 2 of the PXF upgrade procedure.

Installing Java for PXF

PXF is a Java service. It requires a Java 8 or Java 11 installation on each Greenplum Database host.

Prerequisites

Ensure that you have access to, or superuser permissions to install, Java 8 or Java 11 on each Greenplum Database host.

Procedure

Perform the following procedure to install Java on the master, standby master, and on each segment host in your Greenplum Database cluster. You will use the `gpssh` utility where possible to run a command on multiple hosts.

1. Log in to your Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. Determine the version(s) of Java installed on the system:

```
gpadmin@gpmaster$ rpm -qa | grep java
```

3. If the system does not include a Java version 8 or 11 installation, install one of these Java versions on the master, standby master, and on each Greenplum Database segment host.

    1. Create a text file that lists your Greenplum Database standby master host and segment hosts, one host name per line. For example, a file named `gphostfile` may include:

    ```
    gpmaster
    mstandby
    seghost1
    seghost2
    seghost3
    ```

    2. Install the Java package on each host. For example, to install Java version 8:

    ```
    gpadmin@gpmaster$ gpssh -e -v -f gphostfile sudo yum -y install java-1.8.
    0-openjdk-1.8.0*
    ```

4. Identify the Java 8 or 11 `$JAVA_HOME` setting for PXF. For example:

    If you installed Java 8:

    ```
    JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.x86_64/jre
    ```

    If you installed Java 11:

    ```
    JAVA_HOME=/usr/lib/jvm/java-11-openjdk-11.0.4.11-0.el7_6.x86_64
    ```

    If the superuser configures the newly-installed Java alternative as the system default:

    ```
    JAVA_HOME=/usr/lib/jvm/jre
    ```

5. Note the `$JAVA_HOME` setting; you provide this value when you initialize PXF.

Uninstalling PXF

The PXF download package is an `.rpm` file that installs PXF libraries, executables, and script files on a Greenplum Database host.

If you want to remove PXF from the Greenplum cluster and from your hosts, you will:

1. Satisfy the prerequisites.
2. Uninstall PXF from every host in your Greenplum Database cluster.

Prerequisites

Before you uninstall PXF, ensure that you meet the following prerequisites:

- You have access to all hosts (master, standby master, and segment hosts) in your Greenplum Database cluster.
- You must be an operating system superuser, or have `sudo` or other privileges, to remove the PXF package from the default location `/usr/local/pxf-gp<greenplum-major-version>`.

Uninstalling PXF

Follow these steps to remove PXF from your Greenplum Database cluster:

1. Log in to the Greenplum Database master node. For example:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Stop PXF as described in Stopping PXF.
3. Remove the PXF library and extension files from your Greenplum installation:

   ```
   gpadmin@gpmaster$ rm $GPHOME/lib/postgresql/pxf.so
   gpadmin@gpmaster$ rm $GPHOME/share/postgresql/extension/pxf*
   ```

4. Remove PXF from each Greenplum Database master, standby, and segment host. If you installed the `.rpm` package into the default location, you must have operating system `sudo` or other privileges to remove the package. For example, if you installed PXF for Greenplum 6 in the default location on a CentOS 7 system, the following command removes the PXF package on all hosts listed in `gphostfile`:

   ```
   gpadmin@gpmaster$ gpssh -e -v -f gphostfile sudo rpm -e pxf-gp6
   ```

   The command removes the PXF files installed by the `.rpm` package on all Greenplum hosts. The command also removes the PXF runtime directories on all hosts.

The PXF configuration directory `$PXF_CONF` is not affected by this command and remains on the

Greenplum hosts.

Upgrading PXF

If you have installed the PXF `.rpm` and have initialized, configured, and are using PXF in your current Greenplum Database 5.21.2+ or 6.x installation, you must perform some upgrade actions when you install a new version of the PXF `.rpm` .

The PXF upgrade procedure has two parts. You perform one procedure before, and one procedure after, you install a new version to upgrade PXF:

- Step 1: Complete the PXF Pre-Upgrade Actions
- Install the new version of PXF
- Step 2: Upgrade PXF

Step 1: Complete the PXF Pre-Upgrade Actions

Perform this procedure before you upgrade to a new version of PXF:

1. Log in to the Greenplum Database master node. For example:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Identify and note the version of PXF currently running in your Greenplum cluster:

   ```
   gpadmin@gpmaster$ pxf version
   ```

3. If the `$GPHOME/pxf` directory exists, and you are running PXF version 5.12.x or older, back up the Greenplum PXF embedded installation. For example:

   ```
   gpadmin@gpmaster$ mkdir $HOME/pxf_gp_backup
   gpadmin@gpmaster$ cp -r $GPHOME/pxf $HOME/pxf_gp_backup/
   gpadmin@gpmaster$ cp $GPHOME/share/postgresql/extension/pxf* $HOME/pxf_gp_backu
   p/
   gpadmin@gpmaster$ cp $GHOME/lib/postgresql/pxf* $HOME/pxf_gp_backup/
   ```

4. Stop PXF on each segment host as described in Stopping PXF.
5. Install the new version of PXF, identify and note the new PXF version number, and then continue your PXF upgrade with Step 2: Completing the PXF Upgrade.

Step 2: Upgrade PXF

After you install the new version of PXF, perform the following procedure:

1. Log in to the Greenplum Database master node. For example:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Initialize PXF on each segment host as described in Initializing PXF. You may choose to use your existing `$PXF_CONF` for the initialization.
3. **If you are upgrading from PXF version 5.9.x or earlier** and you have configured any JDBC servers that access Kerberos-secured Hive, you must now set the `hadoop.security.authentication` property to the `jdbc-site.xml` file to explicitly identify use of the Kerberos authentication method. Perform the following for each of these server configs:
    1. Navigate to the server configuration directory.
    2. Open the `jdbc-site.xml` file in the editor of your choice and uncomment or add the following property block to the file:

        ```
        <property>
            <name>hadoop.security.authentication</name>
            <value>kerberos</value>
        </property>
        ```
    3. Save the file and exit the editor.
4. **If you are upgrading from PXF version 5.11.x or earlier**: The PXF `Hive` and `HiveRC` profiles now support column projection using column name-based mapping. If you have any existing PXF external tables that specify one of these profiles, and the external table relied on column index-based mapping, you may be required to drop and recreate the tables:
    1. Identify all PXF external tables that you created that specify a `Hive` or `HiveRC` profile.
    2. For *each* external table that you identify in step 1, examine the definitions of both the PXF external table and the referenced Hive table. If the column names of the PXF external table *do not* match the column names of the Hive table:
        1. Drop the existing PXF external table. For example:

            ```
            DROP EXTERNAL TABLE pxf_hive_table1;
            ```
        2. Recreate the PXF external table using the Hive column names. For example:

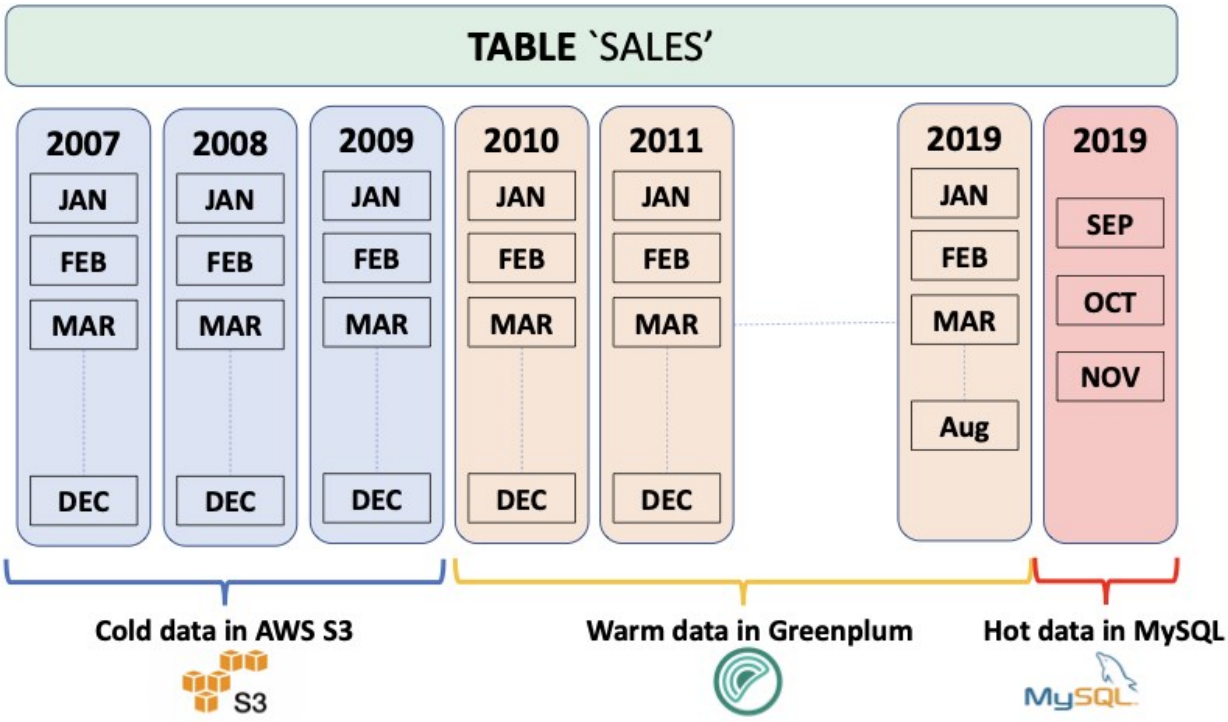
            ```
            CREATE EXTERNAL TABLE pxf_hive_table1( hivecolname int, hivecolnam
            e2 text )
              LOCATION( 'pxf://default.hive_table_name?PROFILE=Hive')
            FORMAT 'custom' (FORMATTER='pxfwritable_import');
            ```
        3. Review any SQL scripts that you may have created that reference the PXF external table, and update column names if required.
5. Synchronize the PXF configuration from the master host to the standby master and each Greenplum Database segment host. For example:

    ```
    gpadmin@gpmaster$ pxf cluster sync
    ```
6. Start PXF on each segment host as described in Starting PXF.

Greenplum Platform Extension Framework (PXF)

With the explosion of data stores and cloud services, data now resides across many disparate systems and in a variety of formats. Often, data is classified both by its location and the operations performed on the data, as well as how often the data is accessed: real-time or transactional (hot), less frequent (warm), or archival (cold).

The diagram below describes a data source that tracks monthly sales across many years. Real-time operational data is stored in MySQL. Data subject to analytic and business intelligence operations is stored in Greenplum Database. The rarely accessed, archival data resides in AWS S3.



When multiple, related data sets exist in external systems, it is often more efficient to join data sets remotely and return only the results, rather than negotiate the time and storage requirements of performing a rather expensive full data load operation. The *Greenplum Platform Extension Framework (PXF)*, a Greenplum extension that provides parallel, high throughput data access and federated query processing, provides this capability.

With PXF, you can use Greenplum and SQL to query these heterogeneous data sources:

- Hadoop, Hive, and HBase
- Azure Blob Storage and Azure Data Lake
- AWS S3
- Minio

- Google Cloud Storage
- SQL databases including Apache Ignite, Hive, MySQL, ORACLE, Microsoft SQL Server, DB2, and PostgreSQL (via JDBC)

And these data formats:

- Avro, AvroSequenceFile
- JSON
- ORC
- Parquet
- RCFile
- SequenceFile
- Text (plain, delimited, embedded line feeds)

Basic Usage

You use PXF to map data from an external source to a Greenplum Database *external table* definition. You can then use the PXF external table and SQL to:

- Perform queries on the external data, leaving the referenced data in place on the remote system.
- Load a subset of the external data into Greenplum Database.
- Run complex queries on local data residing in Greenplum tables and remote data referenced via PXF external tables.
- Write data to the external data source.

Check out the PXF introduction for a high level overview important PXF concepts.

Get Started Configuring PXF

The Greenplum Database administrator manages PXF, Greenplum Database user privileges, and external data source configuration. Tasks include:

- Installing, configuring, starting, monitoring, and troubleshooting the PXF service.
- Managing PXF upgrade.
- Configuring and publishing one or more server definitions for each external data source. This definition specifies the location of, and access credentials to, the external data source.
- Granting Greenplum user access to PXF and PXF external tables.

Get Started Using PXF

A Greenplum Database user creates a PXF external table that references a file or other data in the external data source, and uses the external table to query or load the external data in Greenplum. Tasks are external data store-dependent:

- See Accessing Hadoop with PXF when the data resides in Hadoop.

- See Accessing Azure, Google Cloud Storage, Minio, and S3 Object Stores with PXF when the data resides in an object store.
- See Accessing an SQL Database with PXF when the data resides in an external SQL database.

Introduction to PXF

The Greenplum Platform Extension Framework (PXF) provides *connectors* that enable you to access data stored in sources external to your Greenplum Database deployment. These connectors map an external data source to a Greenplum Database *external table* definition. When you create the Greenplum Database external table, you identify the external data store and the format of the data via a *server* name and a *profile* name that you provide in the command.

You can query the external table via Greenplum Database, leaving the referenced data in place. Or, you can use the external table to load the data into Greenplum Database for higher performance.

Supported Platforms

Operating Systems

PXF supports the Red Hat Enterprise Linux 64-bit 7.x, CentOS 64-bit 7.x, and Ubuntu 18.04 LTS operating system platforms.

Starting in 6.x, Greenplum does not bundle `cURL` and instead loads the system-provided library. PXF requires `cURL` version 7.29.0 or newer. The officially-supported `cURL` for the CentOS 6.x and Red Hat Enterprise Linux 6.x operating systems is version 7.19.*. Greenplum Database 6 does not support running PXF on CentOS 6.x or RHEL 6.x due to this limitation.

Java

PXF supports Java 8 and Java 11.

Hadoop

PXF bundles all of the Hadoop JAR files on which it depends, and supports the following Hadoop component versions:

PXF Version	Hadoop Version	Hive Server Version	HBase Server Version
5.9+	2.x, 3.1+	1.x, 2.x, 3.1+	1.3.2
5.8	2.x	1.x	1.3.2

Architectural Overview

Your Greenplum Database deployment consists of a master node and multiple segment hosts. A single PXF agent process on each Greenplum Database segment host allocates a worker thread for each segment instance on a segment host that participates in a query against an external table. The PXF agents on multiple segment hosts communicate with the external data store in parallel.

About Connectors, Servers, and Profiles

Connector is a generic term that encapsulates the implementation details required to read from or write to an external data store. PXF provides built-in connectors to Hadoop (HDFS, Hive, HBase), object stores (Azure, Google Cloud Storage, Minio, S3), and SQL databases (via JDBC).

A PXF *Server* is a named configuration for a connector. A server definition provides the information required for PXF to access an external data source. This configuration information is data-store-specific, and may include server location, access credentials, and other relevant properties.

The Greenplum Database administrator will configure at least one server definition for each external data store that they will allow Greenplum Database users to access, and will publish the available server names as appropriate.

You specify a `SERVER=<server_name>` setting when you create the external table to identify the server configuration from which to obtain the configuration and credentials to access the external data store.

The default PXF server is named `default` (reserved), and when configured provides the location and access information for the external data source in the absence of a `SERVER=<server_name>` setting.

Finally, a PXF *profile* is a named mapping identifying a specific data format or protocol supported by a specific external data store. PXF supports text, Avro, JSON, RCFile, Parquet, SequenceFile, and ORC data formats, and the JDBC protocol, and provides several built-in profiles as discussed in the following section.

Creating an External Table

PXF implements a Greenplum Database protocol named `pxf` that you can use to create an external table that references data in an external data store. The syntax for a `CREATE EXTERNAL TABLE` command that specifies the `pxf` protocol follows:

```
CREATE [WRITABLE] EXTERNAL TABLE <table_name>
        ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION('pxf://<path-to-data>?PROFILE=<profile_name>[&SERVER=<server_name>][&<custom-
option>=<value>[...]]')
FORMAT '[TEXT|CSV|CUSTOM]' (<formatting-properties>);
```

The `LOCATION` clause in a `CREATE EXTERNAL TABLE` statement specifying the `pxf` protocol is a URI. This URI identifies the path to, or other information describing, the location of the external data. For example, if the external data store is HDFS, the <path-to-data> identifies the absolute path to a specific HDFS file. If the external data store is Hive, <path-to-data> identifies a schema-qualified Hive table name.

You use the query portion of the URI, introduced by the question mark (?), to identify the PXF server and profile names.

PXF may require additional information to read or write certain data formats. You provide profile-specific information using the optional <custom-option>=<value> component of the `LOCATION` string and formatting information via the <formatting-properties> component of the string. The custom options and formatting properties supported by a specific profile vary; they are identified in usage documentation for the profile.

Table 1. CREATE EXTERNAL TABLE Parameter Values and Descriptions

Keyword	Value and Description
<path-to-data>	A directory, file name, wildcard pattern, table name, etc. The syntax of <path-to-data> is dependent upon the external data source.
PROFILE=<profile_name>	The profile that PXF uses to access the data. PXF supports profiles that access text, Avro, JSON, RCFile, Parquet, SequenceFile, and ORC data in Hadoop services, object stores, and other SQL databases.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. PXF uses the `default` server if not specified.
<custom-option>=<value>	Additional options and their values supported by the profile or the server.
FORMAT <value>	PXF profiles support the `TEXT`, `CSV`, and `CUSTOM` formats.
<formatting-properties>	Formatting properties supported by the profile; for example, the `FORMATTER` or `delimiter`.

Note: When you create a PXF external table, you cannot use the `HEADER` option in your formatter specification.

Other PXF Features

Certain PXF connectors and profiles support filter pushdown and column projection. Refer to the following topics for detailed information about this support:

- About PXF Filter Pushdown
- About Column Projection in PXF

About PXF Filter Pushdown

PXF supports filter pushdown. When filter pushdown is enabled, the constraints from the `WHERE` clause of a `SELECT` query can be extracted and passed to the external data source for filtering. This process can improve query performance, and can also reduce the amount of data that is transferred to Greenplum Database.

You enable or disable filter pushdown for all external table protocols, including `pxf`, by setting the `gp_external_enable_filter_pushdown` server configuration parameter. The default value of this configuration parameter is `on`; set it to `off` to disable filter pushdown. For example:

```
SHOW gp_external_enable_filter_pushdown;
SET gp_external_enable_filter_pushdown TO 'on';
```

Note: Some external data sources do not support filter pushdown. Also, filter pushdown may not be supported with certain data types or operators. If a query accesses a data source that does not support filter push-down for the query constraints, the query is instead executed without filter pushdown (the data is filtered after it is transferred to Greenplum Database).

PXF filter pushdown can be used with these data types (connector- and profile-specific):

- `INT2`, `INT4`, `INT8`
- `CHAR`, `TEXT`
- `FLOAT`

- `NUMERIC` (not available with the S3 connector when using S3 Select)
- `BOOL`
- `DATE`, `TIMESTAMP` (available only with the JDBC connector and the S3 connector when using S3 Select)

You can use PXF filter pushdown with these arithmetic and logical operators (connector- and profile-specific):

- `<`, `<=`, `>=`, `>`
- `<>`, `=`
- `AND`, `OR`, `NOT`
- `LIKE` (`TEXT` fields, JDBC connector only)

PXF accesses data sources using profiles exposed by different connectors, and filter pushdown support is determined by the specific connector implementation. The following PXF profiles support some aspect of filter pushdown:

Profile	<, >, <=, >=, =, <>	LIKE	IS [NOT] NULL	IN	AND	OR	NOT
Jdbc	Y	Y	Y	Y	Y	Y	Y
*:parquet	Y[1]	N	Y[1]	N	Y[1]	Y[1]	Y[1]
s3:parquet and s3:text with S3-Select	Y	N	Y	Y	Y	Y	Y
HBase	Y	N	Y	N	Y	Y	N
Hive	Y[2]	N	N	N	Y[2]	Y[2]	N
HiveText	Y[2]	N	N	N	Y[2]	Y[2]	N
HiveRC	Y[2]	N	N	N	Y[2]	Y[2]	N
HiveORC	Y, Y[2]	N	Y	Y	Y, Y[2]	Y, Y[2]	Y
HiveVectorizedORC	Y, Y[2]	N	Y	Y	Y, Y[2]	Y, Y[2]	Y

[1] PXF applies the predicate, rather than the remote system, reducing CPU usage and the memory footprint.

[2] PXF supports partition pruning based on partition keys.

PXF does not support filter pushdown for any profile not mentioned in the table above, including: *:avro, *:AvroSequenceFile, *:SequenceFile, *:json, *:text, and *:text:multi.

To summarize, all of the following criteria must be met for filter pushdown to occur:

- You enable external table filter pushdown by setting the `gp_external_enable_filter_pushdown` server configuration parameter to `'on'`.
- The Greenplum Database protocol that you use to access external data source must support filter pushdown. The `pxf` external table protocol supports pushdown.

- The external data source that you are accessing must support pushdown. For example, HBase and Hive support pushdown.
- For queries on external tables that you create with the `pxf` protocol, the underlying PXF connector must also support filter pushdown. For example, the PXF Hive, HBase, and JDBC connectors support pushdown.
    - Refer to Hive Partition Filter Pushdown for more information about Hive support for this feature.

About Column Projection in PXF

PXF supports column projection, and it is always enabled. With column projection, only the columns required by a `SELECT` query on an external table are returned from the external data source. This process can improve query performance, and can also reduce the amount of data that is transferred to Greenplum Database.

Note: Some external data sources do not support column projection. If a query accesses a data source that does not support column projection, the query is instead executed without it, and the data is filtered after it is transferred to Greenplum Database.

Column projection is automatically enabled for the `pxf` external table protocol. PXF accesses external data sources using different connectors, and column projection support is also determined by the specific connector implementation. The following PXF connector and profile combinations support column projection on read operations:

Data Source	Connector	Profile(s)
External SQL database	JDBC Connector	Jdbc
Hive	Hive Connector	Hive, HiveRC, HiveORC, HiveVectorizedORC
Hadoop	HDFS Connector	hdfs:parquet
Amazon S3	S3-Compatible Object Store Connectors	s3:parquet
Amazon S3 using S3 Select	S3-Compatible Object Store Connectors	s3:parquet, s3:text
Google Cloud Storage	GCS Object Store Connector	gs:parquet
Azure Blob Storage	Azure Object Store Connector	wasbs:parquet
Azure Data Lake	Azure Object Store Connector	adl:parquet

Note: PXF may disable column projection in cases where it cannot successfully serialize a query filter; for example, when the `WHERE` clause resolves to a `boolean` type.

To summarize, all of the following criteria must be met for column projection to occur:

- The external data source that you are accessing must support column projection. For example, Hive supports column projection for ORC-format data, and certain SQL databases support column projection.
- The underlying PXF connector and profile implementation must also support column projection. For example, the PXF Hive and JDBC connector profiles identified above support column projection, as do the PXF connectors that support reading Parquet data.

- PXF must be able to serialize the query filter.

About the PXF Installation and Configuration Directories

PXF is installed to `$GPHOME/pxf` on your master and segment nodes when you install Greenplum Database. If you install the PXF `.rpm` package, PXF is installed to `/usr/local/pxf-gp<greenplum-major-version>`, or to a directory of your choosing. This documentation uses `$PXF_HOME` to refer to the PXF installation directory.

PXF Installation Directories

The following PXF files and directories are installed to `$PXF_HOME` when you install Greenplum Database or the PXF `.rpm`:

Directory	Description
apache-tomcat/	The PXF Tomcat directory.
bin/	The PXF script and executable directory.
commit.sha	The commit identifier for this PXF release.
conf/	The PXF internal configuration directory. This directory contains the `pxf-env-default.sh` and `pxf-profiles-default.xml` configuration files. After initializing PXF, this directory will also include the `pxf-private.classpath` file.
gpextable/	The PXF extension files. The files in this directory are copied to the Greenplum installation (`$GPHOME`) on a single host or all hosts in the cluster when you run `pxf [cluster] register`.
lib/	The PXF library directory.
templates/	Configuration templates for PXF.
version	The PXF version.

PXF Runtime Directories

During initialization and startup, PXF creates the following internal directories in `$PXF_HOME`:

Directory	Description
pxf-service/	After initializing PXF, the PXF service instance directory.
run/	After starting PXF, the PXF run directory. Includes a PXF catalina process id file.

PXF User Configuration Directories

Also during initialization, PXF populates a user configuration directory that you specify via the

`PXF_CONF` environment variable with the following subdirectories and template files:

Directory	Description
conf/	The location of user-customizable PXF configuration files: `pxf-env.sh`, `pxf-log4j.properties`, and `pxf-profiles.xml`.
keytabs/	The default location for the PXF service Kerberos principal keytab file.
lib/	The default PXF user runtime library directory.
logs/	The PXF runtime log file directory. Includes `pxf-service.log` and the Tomcat-related log `catalina.out`. The `logs/` directory and log files are readable only by the `gpadmin` user.
servers/	The server configuration directory; each subdirectory identifies the name of a server. The default server is named `default`. The Greenplum Database administrator may configure other servers.
templates/	The configuration directory for connector server template files.

Refer to Initializing PXF and Starting PXF for detailed information about the PXF initialization and startup commands and procedures.

Configuring PXF

Your Greenplum Database deployment consists of a master node and multiple segment hosts. When you initialize and configure the Greenplum Platform Extension Framework (PXF), you start a single PXF JVM process on each Greenplum Database segment host.

PXF provides connectors to Hadoop, Hive, HBase, object stores, and external SQL data stores. You must configure PXF to support the connectors that you plan to use.

To configure PXF, you must:

1. Install Java 8 or 11 on each Greenplum Database segment host as described in Installing Java for PXF.
2. Initialize the PXF Service.
3. If you plan to use the Hadoop, Hive, or HBase PXF connectors, you must perform the configuration procedure described in Configuring PXF Hadoop Connectors.
4. If you plan to use the PXF connectors to access the Azure, Google Cloud Storage, Minio, or S3 object store(s), you must perform the configuration procedure described in Configuring Connectors to Azure, Google Cloud Storage, Minio, and S3 Object Stores.
5. If you plan to use the PXF JDBC Connector to access an external SQL database, perform the configuration procedure described in Configuring the JDBC Connector.
6. Start PXF.

Initializing PXF

The PXF server is a Java application. You must explicitly initialize the PXF Java service instance. This one-time initialization copies PXF extension files to the Greenplum Database installation, creates the PXF service web application, and generates PXF configuration files and templates.

PXF provides two management commands that you can use for initialization:

- `pxf cluster init` - initialize all PXF service instances in the Greenplum Database cluster
- `pxf init` - initialize the PXF service instance on the current Greenplum Database host

PXF also provides similar `reset` commands that you can use to reset your PXF configuration.

The procedures in this topic assume that you have added the `$PXF_HOME/bin` directory to your `$PATH`.

Configuration Properties

PXF supports both internal and user-customizable configuration properties.

PXF internal configuration files are located in your PXF installation in the `$PXF_HOME/conf` directory.

You identify the PXF user configuration directory at PXF initialization time via an environment

variable named `$PXF_CONF`. If you do not set `$PXF_CONF` prior to initializing PXF, PXF may prompt you to accept or decline the default user configuration directory, `$HOME/pxf`, during the initialization process.

Note: Choose a `$PXF_CONF` directory location that you can back up, and ensure that it resides outside of your Greenplum Database installation directory.

Refer to PXF User Configuration Directories for a list of `$PXF_CONF` subdirectories and their contents.

Initialization Overview

The PXF server runs on Java 8 or 11. You identify the PXF `$JAVA_HOME` and `$PXF_CONF` settings at PXF initialization time.

Initializing PXF:

- Copies PXF extension files from `$PXF_HOME/gpextable` to the Greenplum Database install directory when the `$GPHOME` environment variable is set.
- Creates the PXF Java web application.
- Generates PXF internal configuration files, setting default properties specific to your configuration.

Initializing PXF also creates the `$PXF_CONF` user configuration directory if it does not already exist, and then populates `conf` and `templates` subdirectories with the following:

- `conf/` - user-customizable files for PXF runtime and logging configuration settings
- `templates/` - template configuration files

PXF remembers the `JAVA_HOME` setting that you specified during initialization by updating the property of the same name in the `$PXF_CONF/conf/pxf-env.sh` user configuration file. PXF sources this environment file on startup, allowing it to run with a Java installation that is different than the system default Java.

If the `$PXF_CONF` directory that you specify during initialization already exists, PXF updates only the `templates` subdirectory and the `$PXF_CONF/conf/pxf-env.sh` environment configuration file.

Prerequisites

Before initializing PXF in your Greenplum Database cluster, ensure that:

- Your Greenplum Database cluster is up and running.
- You have identified the PXF user configuration directory filesystem location, `$PXF_CONF`, and that the `gpadmin` user has the necessary permissions to create, or write to, this directory.
- You can identify the Java 8 or 11 `$JAVA_HOME` setting for PXF.

Procedure

Perform the following procedure to initialize PXF on each segment host in your Greenplum Database cluster.

1. Log in to the Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. Export the PXF `JAVA_HOME` setting in your shell. For example:

```
gpadmin@gpmaster$ export JAVA_HOME=/usr/lib/jvm/jre
```

3. Run the `pxf cluster init` command to initialize the PXF service on the master, standby master, and on each segment host. For example, the following command specifies `/usr/local/greenplum-pxf` as the PXF user configuration directory for initialization:

```
gpadmin@gpmaster$ PXF_CONF=/usr/local/greenplum-pxf pxf cluster init
```

Note: The PXF service runs only on the segment hosts. However,`pxf cluster init` also sets up the PXF user configuration directories on the Greenplum Database master and standby master hosts.

Resetting PXF

Should you need to, you can reset PXF to its uninitialized state. You might choose to reset PXF if you specified an incorrect `PXF_CONF` directory, or if you want to start the initialization procedure from scratch.

When you reset PXF, PXF prompts you to confirm the operation. If you confirm, PXF removes the following runtime files and directories:

- `$PXF_HOME/conf/pxf-private.classpath`
- `$PXF_HOME/pxf-service`
- `$PXF_HOME/run`

PXF does not remove the `$PXF_CONF` directory during a reset operation, nor does PXF remove any PXF extension files that may have been copied to your Greenplum Database installation.

You must stop the PXF service instance on a segment host before you can reset PXF on the host.

Procedure

Perform the following procedure to reset PXF on each segment host in your Greenplum Database cluster.

1. Log in to the Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. Stop the PXF service instances on each segment host:

```
gpadmin@gpmaster$ pxf cluster stop
```

3. Reset the PXF service instances on all Greenplum hosts:

```
gpadmin@gpmaster$ pxf cluster reset
```

Note: After you reset PXF, you must initialize and start PXF to use the service again.

Configuring PXF Servers

This topic provides an overview of PXF server configuration. To configure a server, refer to the topic specific to the connector that you want to configure.

You read from or write data to an external data store via a PXF connector. To access an external data store, you must provide the server location. You may also be required to provide client access credentials and other external data store-specific properties. PXF simplifies configuring access to external data stores by:

- Supporting file-based connector and user configuration
- Providing connector-specific template configuration files

A PXF *Server* definition is simply a named configuration that provides access to a specific external data store. A PXF server name is the name of a directory residing in `$PXF_CONF/servers/`. The information that you provide in a server configuration is connector-specific. For example, a PXF JDBC Connector server definition may include settings for the JDBC driver class name, URL, username, and password. You can also configure connection-specific and session-specific properties in a JDBC server definition.

PXF provides a server template file for each connector; this template identifies the typical set of properties that you must configure to use the connector.

You will configure a server definition for each external data store that Greenplum Database users need to access. For example, if you require access to two Hadoop clusters, you will create a PXF Hadoop server configuration for each cluster. If you require access to an Oracle and a MySQL database, you will create one or more PXF JDBC server configurations for each database.

A server configuration may include default settings for user access credentials and other properties for the external data store. You can allow Greenplum Database users to access the external data store using the default settings, or you can configure access and other properties on a per-user basis. This allows you to configure different Greenplum Database users with different external data store access credentials in a single PXF server definition.

About Server Template Files

The configuration information for a PXF server resides in one or more `<connector>-site.xml` files in `$PXF_CONF/servers/<server_name>/`.

PXF provides a template configuration file for each connector. These server template configuration files are located in the `$PXF_CONF/templates/` directory after you initialize PXF:

```
gpadmin@gpmaster$ ls $PXF_CONF/templates
adl-site.xml    hbase-site.xml  jdbc-site.xml    pxf-site.xml    yarn-site.xml
core-site.xml   hdfs-site.xml   mapred-site.xml  s3-site.xml
gs-site.xml     hive-site.xml   minio-site.xml   wasbs-site.xml
```

For example, the contents of the `s3-site.xml` template file follow:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property>
        <name>fs.s3a.access.key</name>
```

```
        <value>YOUR_AWS_ACCESS_KEY_ID</value>
    </property>
    <property>
        <name>fs.s3a.secret.key</name>
        <value>YOUR_AWS_SECRET_ACCESS_KEY</value>
    </property>
    <property>
        <name>fs.s3a.fast.upload</name>
        <value>true</value>
    </property>
</configuration>
```

You specify credentials to PXF in clear text in configuration files.

Note: The template files for the Hadoop connectors are not intended to be modified and used for configuration, as they only provide an example of the information needed. Instead of modifying the Hadoop templates, you will copy several Hadoop `*-site.xml` files from the Hadoop cluster to your PXF Hadoop server configuration.

About the Default Server

PXF defines a special server named `default`. When you initialize PXF, it automatically creates a `$PXF_CONF/servers/default/` directory. This directory, initially empty, identifies the default PXF server configuration. You can configure and assign the default PXF server to any external data source. For example, you can assign the PXF default server to a Hadoop cluster, or to a MySQL database that your users frequently access.

PXF automatically uses the `default` server configuration if you omit the `SERVER=<server_name>` setting in the `CREATE EXTERNAL TABLE` command `LOCATION` clause.

Configuring a Server

When you configure a PXF connector to an external data store, you add a named PXF server configuration for the connector. Among the tasks that you perform, you may:

1. Determine if you are configuring the `default` PXF server, or choose a new name for the server configuration.
2. Create the directory `$PXF_CONF/servers/<server_name>`.
3. Copy template or other configuration files to the new server directory.
4. Fill in appropriate default values for the properties in the template file.
5. Add any additional configuration properties and values required for your environment.
6. Configure one or more users for the server configuration as described in About Configuring a PXF User.
7. Synchronize the server and user configuration to the Greenplum Database cluster.

Note: You must re-sync the PXF configuration to the Greenplum Database cluster after you add or update PXF server configuration.

After you configure a PXF server, you publish the server name to Greenplum Database users who need access to the data store. A user only needs to provide the server name when they create an external table that accesses the external data store. PXF obtains the external data source location

and access credentials from server and user configuration files residing in the server configuration directory identified by the server name.

To configure a PXF server, refer to the connector configuration topic:

- To configure a PXF server for Hadoop, refer to Configuring PXF Hadoop Connectors .
- To configure a PXF server for an object store, refer to Configuring Connectors to Minio and S3 Object Stores and Configuring Connectors to Azure and Google Cloud Storage Object Stores.
- To configure a PXF JDBC server, refer to Configuring the JDBC Connector .

About Kerberos and User Impersonation Configuration (pxf-site.xml)

PXF includes a template file named `pxf-site.xml`. You use the `pxf-site.xml` template file to specify Kerberos and/or user impersonation settings for a server configuration.

The settings in this file apply only to Hadoop and JDBC server configurations; they do not apply to object store server configurations.

You configure properties in the `pxf-site.xml` file for a PXF server when one or more of the following conditions hold:

- The remote Hadoop system utilizes Kerberos authentication.
- You want to enable/disable user impersonation on the remote Hadoop or external database system.

`pxf-site.xml` includes the following properties:

Property	Description	Default Value
pxf.service.kerberos.principal	The Kerberos principal name.	gpadmin/_HOST@EXAMPLE.COM
pxf.service.kerberos.keytab	The file system path to the Kerberos keytab file.	$PXF_CONF/keytabs/pxf.service.keytab
pxf.service.user.name	The log in user for the remote system.	The operating system user that starts the pxf process, typically `gpadmin`.
pxf.service.user.impersonation	Enables/disables user impersonation on the remote system.	The value of the (deprecated) `PXF_USER_IMPERSONATION` property when that property is set. If the `PXF_USER_IMPERSONATION` property does not exist and the `pxf.service.user.impersonation` property is missing from `pxf-site.xml`, the default is `false`, user impersonation is disabled on the remote system.

Refer to Configuring PXF Hadoop Connectors and Configuring the JDBC Connector for information about relevant `pxf-site.xml` property settings for Hadoop and JDBC server configurations, respectively.

Configuring a PXF User

You can configure access to an external data store on a per-server, per-Greenplum-user basis.

PXF per-server, per-user configuration provides the most benefit for JDBC servers.

You configure external data store user access credentials and properties for a specific Greenplum Database user by providing a `<greenplum_user_name>-user.xml` user configuration file in the PXF server configuration directory, `$PXF_CONF/servers/<server_name>/`. For example, you specify the properties for the Greenplum Database user named `bill` in the file `$PXF_CONF/servers/<server_name>/bill-user.xml`. You can configure zero, one, or more users in a PXF server configuration.

The properties that you specify in a user configuration file are connector-specific. You can specify any configuration property supported by the PXF connector server in a `<greenplum_user_name>-user.xml` configuration file.

For example, suppose you have configured access to a PostgreSQL database in the PXF JDBC server configuration named `pgsrv1`. To allow the Greenplum Database user named `bill` to access this database as the PostgreSQL user named `pguser1`, password `changeme`, you create the user configuration file `$PXF_CONF/servers/pgsrv1/bill-user.xml` with the following properties:

```
<configuration>
    <property>
        <name>jdbc.user</name>
        <value>pguser1</value>
    </property>
    <property>
        <name>jdbc.password</name>
        <value>changeme</value>
    </property>
</configuration>
```

If you want to configure a specific search path and a larger read fetch size for `bill`, you would also add the following properties to the `bill-user.xml` user configuration file:

```
    <property>
        <name>jdbc.session.property.search_path</name>
        <value>bill_schema</value>
    </property>
    <property>
        <name>jdbc.statement.fetchSize</name>
        <value>2000</value>
    </property>
```

Procedure

For each PXF user that you want to configure, you will:

1. Identify the name of the Greenplum Database user.
2. Identify the PXF server definition for which you want to configure user access.
3. Identify the name and value of each property that you want to configure for the user.
4. Create/edit the file `$PXF_CONF/servers/<server_name>/<greenplum_user_name>-user.xml`, and add the outer configuration block:

```
<configuration>
</configuration>
```

5. Add each property/value pair that you identified in Step 3 within the configuration block in the `<greenplum_user_name>-user.xml` file.
6. If you are adding the PXF user configuration to previously configured PXF server definition, synchronize the user configuration to the Greenplum Database cluster.

About Configuration Property Precedence

A PXF server configuration may include default settings for user access credentials and other properties for accessing an external data store. Some PXF connectors, such as the S3 and JDBC connectors, allow you to directly specify certain server properties via custom options in the `CREATE EXTERNAL TABLE` command `LOCATION` clause. A `<greenplum_user_name>-user.xml` file specifies property settings for an external data store that are specific to a Greenplum Database user.

For a given Greenplum Database user, PXF uses the following precedence rules (highest to lowest) to obtain configuration property settings for the user:

1. A property that you configure in `<server_name>/<greenplum_user_name>-user.xml` overrides any setting of the property elsewhere.
2. A property that is specified via custom options in the `CREATE EXTERNAL TABLE` command `LOCATION` clause overrides any setting of the property in a PXF server configuration.
3. Properties that you configure in the `<server_name>` PXF server definition identify the default property values.

These precedence rules allow you create a single external table that can be accessed by multiple Greenplum Database users, each with their own unique external data store user credentials.

Using a Server Configuration

To access an external data store, the Greenplum Database user specifies the server name in the `CREATE EXTERNAL TABLE` command `LOCATION` clause `SERVER=<server_name>` option. The `<server_name>` that the user provides identifies the server configuration directory from which PXF obtains the configuration and credentials to access the external data store.

For example, the following command accesses an S3 object store using the server configuration defined in the `$PXF_CONF/servers/s3srvcfg/s3-site.xml` file:

```
CREATE EXTERNAL TABLE pxf_ext_tbl(name text, orders int)
  LOCATION ('pxf://BUCKET/dir/file.txt?PROFILE=s3:text&SERVER=s3srvcfg')
FORMAT 'TEXT' (delimiter=E',');
```

PXF automatically uses the `default` server configuration when no `SERVER=<server_name>` setting is provided.

For example, if the `default` server configuration identifies a Hadoop cluster, the following example command references the HDFS file located at `/path/to/file.txt`:

```
CREATE EXTERNAL TABLE pxf_ext_hdfs(location text, miles int)
  LOCATION ('pxf://path/to/file.txt?PROFILE=hdfs:text')
```

```
FORMAT 'TEXT' (delimiter=E',');
```

A Greenplum Database user who queries or writes to an external table accesses the external data store with the credentials configured for the `<server_name>` user. If no user-specific credentials are configured for `<server_name>`, the Greenplum user accesses the external data store with the default credentials configured for `<server_name>`.

Configuring PXF Hadoop Connectors (Optional)

PXF is compatible with Cloudera, Hortonworks Data Platform, MapR, and generic Apache Hadoop distributions. This topic describes how configure the PXF Hadoop, Hive, and HBase connectors.

If you do not want to use the Hadoop-related PXF connectors, then you do not need to perform this procedure.

Prerequisites

Configuring PXF Hadoop connectors involves copying configuration files from your Hadoop cluster to the Greenplum Database master host. If you are using the MapR Hadoop distribution, you must also copy certain JAR files to the master host. Before you configure the PXF Hadoop connectors, ensure that you can copy files from hosts in your Hadoop cluster to the Greenplum Database master.

Procedure

Perform the following procedure to configure the desired PXF Hadoop-related connectors on the Greenplum Database master host. After you configure the connectors, you will use the `pxf cluster sync` command to copy the PXF configuration to the Greenplum Database cluster.

In this procedure, you use the `default`, or create a new, PXF server configuration. You copy Hadoop configuration files to the server configuration directory on the Greenplum Database master host. You identify Kerberos and user impersonation settings required for access, if applicable. You may also copy libraries to `$PXF_CONF/lib` for MapR support. You then synchronize the PXF configuration on the master host to the standby master and segment hosts. (PXF creates the`$PXF_CONF/*` directories when you run `pxf cluster init`.)

1. Log in to your Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Identify the name of your PXF Hadoop server configuration.

3. If you are not using the `default` PXF server, create the `$PXF_CONF/servers/<server_name>` directory. For example, use the following command to create a Hadoop server configuration named `hdp3`:

   ```
   gpadmin@gpmaster$ mkdir $PXF_CONF/servers/hdp3
   ```

4. Change to the server directory. For example:

   ```
   gpadmin@gpmaster$ cd $PXF_CONF/servers/default
   ```

   Or,

```
gpadmin@gpmaster$ cd $PXF_CONF/servers/hdp3
```

5. PXF requires information from `core-site.xml` and other Hadoop configuration files. Copy the `core-site.xml`, `hdfs-site.xml`, `mapred-site.xml`, and `yarn-site.xml` Hadoop configuration files from your Hadoop cluster NameNode host to the current host using your tool of choice. Your file paths may differ based on the Hadoop distribution in use. For example, these commands use `scp` to copy the files:

```
gpadmin@gpmaster$ scp hdfsuser@namenode:/etc/hadoop/conf/core-site.xml .
gpadmin@gpmaster$ scp hdfsuser@namenode:/etc/hadoop/conf/hdfs-site.xml .
gpadmin@gpmaster$ scp hdfsuser@namenode:/etc/hadoop/conf/mapred-site.xml .
gpadmin@gpmaster$ scp hdfsuser@namenode:/etc/hadoop/conf/yarn-site.xml .
```

6. If you plan to use the PXF Hive connector to access Hive table data, similarly copy the Hive configuration to the Greenplum Database master host. For example:

```
gpadmin@gpmaster$ scp hiveuser@hivehost:/etc/hive/conf/hive-site.xml .
```

7. If you plan to use the PXF HBase connector to access HBase table data, similarly copy the HBase configuration to the Greenplum Database master host. For example:

```
gpadmin@gpmaster$ scp hbaseuser@hbasehost:/etc/hbase/conf/hbase-site.xml .
```

8. If you are using PXF with the MapR Hadoop distribution, you must copy certain JAR files from your MapR cluster to the Greenplum Database master host. (Your file paths may differ based on the version of MapR in use.) For example, these commands use `scp` to copy the files:

```
gpadmin@gpmaster$ cd $PXF_CONF/lib
gpadmin@gpmaster$ scp mapruser@maprhost:/opt/mapr/hadoop/hadoop-2.7.0/share/had
oop/common/lib/maprfs-5.2.2-mapr.jar .
gpadmin@gpmaster$ scp mapruser@maprhost:/opt/mapr/hadoop/hadoop-2.7.0/share/had
oop/common/lib/hadoop-auth-2.7.0-mapr-1707.jar .
gpadmin@gpmaster$ scp mapruser@maprhost:/opt/mapr/hadoop/hadoop-2.7.0/share/had
oop/common/hadoop-common-2.7.0-mapr-1707.jar .
```

9. Synchronize the PXF configuration to the Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

10. PXF accesses Hadoop services on behalf of Greenplum Database end users. By default, PXF tries to access HDFS, Hive, and HBase using the identity of the Greenplum Database user account that logs into Greenplum Database. In order to support this functionality, you must configure proxy settings for Hadoop, as well as for Hive and HBase if you intend to use those PXF connectors. Follow procedures in Configuring User Impersonation and Proxying to configure user impersonation and proxying for Hadoop services, or to turn off PXF user impersonation.

11. Grant read permission to the HDFS files and directories that will be accessed as external tables in Greenplum Database. If user impersonation is enabled (the default), you must grant this permission to each Greenplum Database user/role name that will use external tables that reference the HDFS files. If user impersonation is not enabled, you must grant this

permission to the `gpadmin` user.

12. If your Hadoop cluster is secured with Kerberos, you must configure PXF and generate Kerberos principals and keytabs for each segment host as described in Configuring PXF for Secure HDFS.

About Updating the Hadoop Configuration

If you update your Hadoop, Hive, or HBase configuration while the PXF service is running, you must copy the updated configuration to the `$PXF_CONF/servers/<server_name>` directory and re-sync the PXF configuration to your Greenplum Database cluster. For example:

```
gpadmin@gpmaster$ cd $PXF_CONF/servers/<server_name>
gpadmin@gpmaster$ scp hiveuser@hivehost:/etc/hive/conf/hive-site.xml .
gpadmin@gpmaster$ pxf cluster sync
```

Configuring the Hadoop User, User Impersonation, and Proxying

PXF accesses Hadoop services on behalf of Greenplum Database end users.

When user impersonation is enabled (the default), PXF accesses Hadoop services using the identity of the Greenplum Database user account that logs in to Greenplum and performs an operation that uses a PXF connector. Keep in mind that PXF uses only the *login* identity of the user when accessing Hadoop services. For example, if a user logs in to Greenplum Database as the user `jane` and then execute `SET ROLE` or `SET SESSION AUTHORIZATION` to assume a different user identity, all PXF requests still use the identity `jane` to access Hadoop services. When user impersonation is enabled, you must explicitly configure each Hadoop data source (HDFS, Hive, HBase) to allow PXF to act as a proxy for impersonating specific Hadoop users or groups.

When user impersonation is disabled, PXF executes all Hadoop service requests as the PXF process owner (usually `gpadmin`) or the Hadoop user identity that you specify. This behavior provides no means to control access to Hadoop services for different Greenplum Database users. It requires that this user have access to all files and directories in HDFS, and all tables in Hive and HBase that are referenced in PXF external table definitions.

You configure the Hadoop user and PXF user impersonation setting for a server via the `pxf-site.xml` server configuration file. Refer to About Kerberos and User Impersonation Configuration (pxf-site.xml) for more information about the configuration properties in this file.

The following table describes some of the PXF configuration scenarios for Hadoop access:

Scenario	pxf-site.xml Required	Impersonation Setting	Required Configuration
PXF accesses Hadoop using the identity of the Greenplum Database user.	yes	true	Enable user impersonation, identify the Hadoop proxy user in the `pxf.service.user.name`, and configure Hadoop proxying for this Hadoop user identity.

Scenario	pxf-site.xml Required	Impersonation Setting	Required Configuration
PXF accesses Hadoop using the identity of the operating system user that started the PXF process.	yes	false	Disable user impersonation.
PXF accesses Hadoop using a user identity that you specify.	yes	false	Disable user impersonation and identify the Hadoop user identity in the `pxf.service.user.name` property setting.

Configure the Hadoop User

By default, PXF accesses Hadoop using the identity of the Greenplum Database user, and you are required to set up a proxy Hadoop user. You can configure PXF to access Hadoop as a different user on a per-server basis.

Perform the following procedure to configure the Hadoop user:

1. Log in to your Greenplum Database master node as the administrative user:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Identify the name of the PXF Hadoop server configuration that you want to update.
3. Navigate to the server configuration directory. For example, if the server is named `hdp3`:

   ```
   gpadmin@gpmaster$ cd $PXF_CONF/servers/hdp3
   ```

4. If the server configuration does not yet include a `pxf-site.xml` file, copy the template file to the directory. For example:

   ```
   gpadmin@gpmaster$ cp $PXF_CONF/templates/pxf-site.xml .
   ```

5. Open the `pxf-site.xml` file in the editor of your choice, and configure the Hadoop user name. When impersonation is disabled, this name identifies the Hadoop user identity that PXF will use to access the Hadoop system. When user impersonation is enabled, this name identifies the PXF proxy Hadoop user. For example, if you want to access Hadoop as the user `hdfsuser1`:

   ```
   <property>
       <name>pxf.service.user.name</name>
       <value>hdfsuser1</value>
   </property>
   ```

6. Save the `pxf-site.xml` file and exit the editor.
7. Use the `pxf cluster sync` command to synchronize the PXF Hadoop server configuration to your Greenplum Database cluster:

   ```
   gpadmin@gpmaster$ pxf cluster sync
   ```

Configure PXF User Impersonation

PXF user impersonation is enabled by default for Hadoop servers. You can configure PXF user impersonation on a per-server basis. Perform the following procedure to turn PXF user impersonation on or off for the Hadoop server configuration:

In previous versions of Greenplum Database, you configured user impersonation globally for Hadoop clusters via the now deprecated `PXF_USER_IMPERSONATION` setting in the `pxf-env.sh` configuration file.

1. Navigate to the server configuration directory. For example, if the server is named `hdp3`:

   ```
   gpadmin@gpmaster$ cd $PXF_CONF/servers/hdp3
   ```

2. If the server configuration does not yet include a `pxf-site.xml` file, copy the template file to the directory. For example:

   ```
   gpadmin@gpmaster$ cp $PXF_CONF/templates/pxf-site.xml .
   ```

3. Open the `pxf-site.xml` file in the editor of your choice, and update the user impersonation property setting. For example, if you do not require user impersonation for this server configuration, set the `pxf.service.user.impersonation` property to `false`:

   ```
   <property>
       <name>pxf.service.user.impersonation</name>
       <value>false</value>
   </property>
   ```

   If you require user impersonation, turn it on:

   ```
   <property>
       <name>pxf.service.user.impersonation</name>
       <value>true</value>
   </property>
   ```

4. If you enabled user impersonation, you must configure Hadoop proxying as described in Configure Hadoop Proxying. You must also configure Hive User Impersonation and HBase User Impersonation if you plan to use those services.

5. Save the `pxf-site.xml` file and exit the editor.

6. Use the `pxf cluster sync` command to synchronize the PXF Hadoop server configuration to your Greenplum Database cluster:

   ```
   gpadmin@gpmaster$ pxf cluster sync
   ```

Configure Hadoop Proxying

When PXF user impersonation is enabled for a Hadoop server configuration, you must configure Hadoop to permit PXF to proxy Greenplum users. This configuration involves setting certain `hadoop.proxyuser.*` properties. Follow these steps to set up PXF Hadoop proxy users:

1. Log in to your Hadoop cluster and open the `core-site.xml` configuration file using a text editor, or use Ambari or another Hadoop cluster manager to add or edit the Hadoop property values described in this procedure.

2. Set the property `hadoop.proxyuser.<name>.hosts` to specify the list of PXF host names from which proxy requests are permitted. Substitute the PXF proxy Hadoop user for `<name>`. The PXF proxy Hadoop user is the `pxf.service.user.name` that you configured in the procedure above, or, if you are using Kerberos authentication to Hadoop, the proxy user identity is the *primary* component of the Kerberos principal. If you have not configured `pxf.service.user.name`, the proxy user is the operating system user that started PXF. Provide multiple PXF host names in a comma-separated list. For example, if the PXF proxy user is named `hdfsuser2`:

   ```
   <property>
       <name>hadoop.proxyuser.hdfsuser2.hosts</name>
       <value>pxfhost1,pxfhost2,pxfhost3</value>
   </property>
   ```

3. Set the property `hadoop.proxyuser.<name>.groups` to specify the list of HDFS groups that PXF as Hadoop user `<name>` can impersonate. You should limit this list to only those groups that require access to HDFS data from PXF. For example:

   ```
   <property>
       <name>hadoop.proxyuser.hdfsuser2.groups</name>
       <value>group1,group2</value>
   </property>
   ```

4. You must restart Hadoop for your `core-site.xml` changes to take effect.

5. Copy the updated `core-site.xml` file to the PXF Hadoop server configuration directory `$PXF_CONF/servers/<server_name>` on the Greenplum Database master and synchronize the configuration to the standby master and each Greenplum Database segment host.

Hive User Impersonation

The PXF Hive connector uses the Hive MetaStore to determine the HDFS locations of Hive tables, and then accesses the underlying HDFS files directly. No specific impersonation configuration is required for Hive, because the Hadoop proxy configuration in `core-site.xml` also applies to Hive tables accessed in this manner.

HBase User Impersonation

In order for user impersonation to work with HBase, you must enable the `AccessController` coprocessor in the HBase configuration and restart the cluster. See 61.3 Server-side Configuration for Simple User Access Operation in the Apache HBase Reference Guide for the required `hbase-site.xml` configuration settings.

Configuring PXF for Secure HDFS

When Kerberos is enabled for your HDFS filesystem, PXF, as an HDFS client, requires a principal and keytab file to authenticate access to HDFS. To read or write files on a secure HDFS, you must create and deploy Kerberos principals and keytabs for PXF, and ensure that Kerberos authentication is enabled and functioning.

PXF supports simultaneous access to multiple Kerberos-secured Hadoop clusters.

In previous versions of Greenplum Database, you configured the PXF Kerberos principal and keytab for the `default` Hadoop server via the now deprecated `PXF_PRINCIPAL` and `PXF_KEYTAB` settings in the `pxf-env.sh` configuration file.

When Kerberos is enabled, you access Hadoop with the PXF principal and keytab. You can also choose to access Hadoop using the identity of the Greenplum Database user.

You configure the impersonation setting and the Kerberos principal and keytab for a Hadoop server via the `pxf-site.xml` server-specific configuration file. Refer to About Kerberos and User Impersonation Configuration (pxf-site.xml) for more information about the configuration properties in this file.

Configure the Kerberos principal and keytab using the following `pxf-site.xml` properties:

Property	Description	Default Value
pxf.service.kerberos.principal	The Kerberos principal name.	gpadmin/_HOST@EXAMPLE.COM
pxf.service.kerberos.keytab	The file system path to the Kerberos keytab file.	$PXF_CONF/keytabs/pxf.service.keytab

The following table describes two scenarios for accessing Hadoop when Kerberos authentication is enabled:

Scenario	Required Configuration
PXF accesses Hadoop using the identity of the configured principal.	Set the `pxf.service.user.impersonation` property setting to `false` in the `pxf-site.xml` file to disable user impersonation.
PXF accesses Hadoop using the identity of the Greenplum Database user.	Set the `pxf.service.user.impersonation` property setting to `true` in the `pxf-site.xml` file to enable user impersonation. You must also configure Hadoop proxying for the Hadoop user identity specified in the *primary* component of the Kerberos principal.

Prerequisites

Before you configure PXF for access to a secure HDFS filesystem, ensure that you have:

- Configured a PXF server for the Hadoop cluster, and can identify the server configuration name.
- Initialized, configured, and started PXF as described in Configuring PXF.
- Verified that Kerberos is enabled for your Hadoop cluster.
- Verified that the HDFS configuration parameter `dfs.block.access.token.enable` is set to `true`. You can find this setting in the `hdfs-site.xml` configuration file on a host in your Hadoop cluster.
- Noted the host name or IP address of each Greenplum Database segment host (<seghost>) and the Kerberos Key Distribution Center (KDC) <kdc-server> host.
- Noted the name of the Kerberos <realm> in which your cluster resides.
- Installed the Kerberos client packages on **each** Greenplum Database segment host if they are not already installed. You must have superuser permissions to install operating system packages. For example:

```
root@gphost$ rpm -qa | grep krb
root@gphost$ yum install krb5-libs krb5-workstation
```

Procedure

There are different procedures for configuring PXF for secure HDFS with a Microsoft Active Directory KDC Server vs. with an MIT Kerberos KDC Server.

Configuring PXF with a Microsoft Active Directory Kerberos KDC Server

When you configure PXF for secure HDFS using an AD Kerberos KDC server, you will perform tasks on both the KDC server host and the Greenplum Database master host.

Perform the following steps the Active Directory domain controller:

1. Start **Active Directory Users and Computers**.
2. Expand the forest domain and the top-level UNIX organizational unit that describes your Greenplum user domain.
3. Select **Service Accounts**, right-click, then select **New->User**.
4. Type a name, eg. `ServiceGreenplumPROD1`, and change the login name to `gpadmin`. Note that the login name should be in compliance with POSIX standard and match `hadoop.proxyuser.<name>.hosts/groups` in the Hadoop `core-site.xml` and the Kerberos principal.
5. Type and confirm the Active Directory service account password. Select the **User cannot change password** and **Password never expires** check boxes, then click **Next**. For security reasons, if you can't have **Password never expires** checked, you will need to generate new keytab file (step 7) every time you change the password of the service account.
6. Click **Finish** to complete the creation of the new user principal.
7. Open Powershell or a command prompt and run the `ktpass` command to generate the keytab file. For example:

   ```
   powershell#>ktpass -out pxf.service.keytab -princ gpadmin@EXAMPLE.COM -mapUser
   ServiceGreenplumPROD1 -pass ******* -crypto all -ptype KRB5_NT_PRINCIPAL
   ```

   With Active Directory, the principal and the keytab file are shared by all Greenplum Database segment hosts.
8. Copy the `pxf.service.keytab` file to the Greenplum master host.

Perform the following procedure on the Greenplum Database master host:

1. Log in to the Greenplum Database master host. For example:

   ```
   $ ssh gpadmin@<gpmaster>
   ```
2. Identify the name of the PXF Hadoop server configuration, and navigate to the server configuration directory. For example, if the server is named `hdp3`:

   ```
   gpadmin@gpmaster$ cd $PXF_CONF/servers/hdp3
   ```

3. If the server configuration does not yet include a `pxf-site.xml` file, copy the template file to the directory. For example:

```
gpadmin@gpmaster$ cp $PXF_CONF/templates/pxf-site.xml .
```

4. Open the `pxf-site.xml` file in the editor of your choice, and update the keytab and principal property settings, if required. Specify the location of the keytab file and the Kerberos principal, substituting your realm. For example:

```
<property>
    <name>pxf.service.kerberos.principal</name>
    <value>gpadmin@EXAMPLE.COM</value>
</property>
<property>
    <name>pxf.service.kerberos.keytab</name>
    <value>${pxf.conf}/keytabs/pxf.service.keytab</value>
</property>
```

5. Enable user impersonation as described in Configure PXF User Impersonation, and configure or verify Hadoop proxying for the *primary* component of the Kerberos principal as described in Configure Hadoop Proxying. For example, if your principal is `gpadmin@EXAMPLE.COM`, configure proxying for the Hadoop user `gpadmin`.

6. Save the file and exit the editor.

7. Synchronize the PXF configuration to your Greenplum Database cluster and restart PXF:

```
gpadmin@master$ pxf cluster sync
gpadmin@master$ pxf cluster restart
```

8. Step 7 does not synchronize the keytabs in `$PXF_CONF`. You must distribute the keytab file to `$PXF_CONF/keytabs/`. Locate the keytab file, copy the file to the `$PXF_CONF` user configuration directory, and set required permissions. For example:

```
gpadmin@gpmaster$ gpscp -f hostfile_all pxf.service.keytab =:$PXF_CONF/keytabs/
gpadmin@gpmaster$ gpssh -f hostfile_all chmod 400 $PXF_CONF/keytabs/pxf.service
.keytab
```

Configuring PXF with an MIT Kerberos KDC Server

When you configure PXF for secure HDFS using an MIT Kerberos KDC server, you will perform tasks on both the KDC server host and the Greenplum Database master host.

Perform the following steps on the MIT Kerberos KDC server host:

1. Log in to the Kerberos KDC server as the `root` user.

```
$ ssh root@<kdc-server>
root@kdc-server$
```

2. Distribute the `/etc/krb5.conf` Kerberos configuration file on the KDC server host to **each** segment host in your Greenplum Database cluster if not already present. For example:

```
root@kdc-server$ scp /etc/krb5.conf seghost:/etc/krb5.conf
```

3. Use the `kadmin.local` command to create a Kerberos PXF service principal for **each** Greenplum Database segment host. The service principal should be of the form `gpadmin/<seghost>@<realm>` where <seghost> is the DNS resolvable, fully-qualified hostname of the segment host system (output of the `hostname -f` command).

   For example, these commands create PXF service principals for the hosts named host1.example.com, host2.example.com, and host3.example.com in the Kerberos realm named `EXAMPLE.COM`:

   ```
   root@kdc-server$ kadmin.local -q "addprinc -randkey -pw changeme gpadmin/host1.
   example.com@EXAMPLE.COM"
   root@kdc-server$ kadmin.local -q "addprinc -randkey -pw changeme gpadmin/host2.
   example.com@EXAMPLE.COM"
   root@kdc-server$ kadmin.local -q "addprinc -randkey -pw changeme gpadmin/host3.
   example.com@EXAMPLE.COM"
   ```

4. Generate a keytab file for each PXF service principal that you created in the previous step. Save the keytab files in any convenient location (this example uses the directory `/etc/security/keytabs`). You will deploy the keytab files to their respective Greenplum Database segment host machines in a later step. For example:

   ```
   root@kdc-server$ kadmin.local -q "xst -norandkey -k /etc/security/keytabs/pxf-h
   ost1.service.keytab gpadmin/host1.example.com@EXAMPLE.COM"
   root@kdc-server$ kadmin.local -q "xst -norandkey -k /etc/security/keytabs/pxf-h
   ost2.service.keytab gpadmin/host2.example.com@EXAMPLE.COM"
   root@kdc-server$ kadmin.local -q "xst -norandkey -k /etc/security/keytabs/pxf-h
   ost3.service.keytab gpadmin/host3.example.com@EXAMPLE.COM"
   ```

   Repeat the `xst` command as necessary to generate a keytab for each PXF service principal that you created in the previous step.

5. List the principals. For example:

   ```
   root@kdc-server$ kadmin.local -q "listprincs"
   ```

6. Copy the keytab file for each PXF service principal to its respective segment host. For example, the following commands copy each principal generated in step 4 to the PXF default keytab directory on the segment host when `PXF_CONF=/usr/local/greenplum-pxf`:

   ```
   root@kdc-server$ scp /etc/security/keytabs/pxf-host1.service.keytab host1.examp
   le.com:/usr/local/greenplum-pxf/keytabs/pxf.service.keytab
   root@kdc-server$ scp /etc/security/keytabs/pxf-host2.service.keytab host2.examp
   le.com:/usr/local/greenplum-pxf/keytabs/pxf.service.keytab
   root@kdc-server$ scp /etc/security/keytabs/pxf-host3.service.keytab host3.examp
   le.com:/usr/local/greenplum-pxf/keytabs/pxf.service.keytab
   ```

   Note the file system location of the keytab file on each PXF host; you will need this information for a later configuration step.

7. Change the ownership and permissions on the `pxf.service.keytab` files. The files must be owned and readable by only the `gpadmin` user. For example:

   ```
   root@kdc-server$ ssh host1.example.com chown gpadmin:gpadmin /usr/local/greenpl
   um-pxf/keytabs/pxf.service.keytab
   ```

```
root@kdc-server$ ssh host1.example.com chmod 400 /usr/local/greenplum-pxf/keyta
bs/pxf.service.keytab
root@kdc-server$ ssh host2.example.com chown gpadmin:gpadmin /usr/local/greenpl
um-pxf/keytabs/pxf.service.keytab
root@kdc-server$ ssh host2.example.com chmod 400 /usr/local/greenplum-pxf/keyta
bs/pxf.service.keytab
root@kdc-server$ ssh host3.example.com chown gpadmin:gpadmin /usr/local/greenpl
um-pxf/keytabs/pxf.service.keytab
root@kdc-server$ ssh host3.example.com chmod 400 /usr/local/greenplum-pxf/keyta
bs/pxf.service.keytab
```

Perform the following steps on the Greenplum Database master host:

1. Log in to the master host. For example:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Identify the name of the PXF Hadoop server configuration that requires Kerberos access.

3. Navigate to the server configuration directory. For example, if the server is named `hdp3`:

   ```
   gpadmin@gpmaster$ cd $PXF_CONF/servers/hdp3
   ```

4. If the server configuration does not yet include a `pxf-site.xml` file, copy the template file to the directory. For example:

   ```
   gpadmin@gpmaster$ cp $PXF_CONF/templates/pxf-site.xml .
   ```

5. Open the `pxf-site.xml` file in the editor of your choice, and update the keytab and principal property settings, if required. Specify the location of the keytab file and the Kerberos principal, substituting your realm. *The default values for these settings are identified below*:

   ```
   <property>
       <name>pxf.service.kerberos.principal</name>
       <value>gpadmin/_HOST@EXAMPLE.COM</value>
   </property>
   <property>
       <name>pxf.service.kerberos.keytab</name>
       <value>${pxf.conf}/keytabs/pxf.service.keytab</value>
   </property>
   ```

   PXF automatically replaces `_HOST` with the FQDN of the segment host.

6. If you want to access Hadoop as the Greenplum Database user:

   1. Enable user impersonation as described in Configure PXF User Impersonation.
   2. Configure Hadoop proxying for the *primary* component of the Kerberos principal as described in Configure Hadoop Proxying. For example, if your principal is `gpadmin/_HOST@EXAMPLE.COM`, configure proxying for the Hadoop user `gpadmin`.

7. If you want to access Hadoop using the identity of the Kerberos principal, disable user impersonation as described in Configure PXF User Impersonation.

8. PXF ignores the `pxf.service.user.name` property when it uses Kerberos authentication to Hadoop. You may choose to remove this property from the `pxf-site.xml` file.

9. Save the file and exit the editor.

10. Synchronize the PXF configuration to your Greenplum Database cluster:

```
gpadmin@seghost$ pxf cluster sync
```

Configuring Connectors to Minio and S3 Object Stores (Optional)

You can use PXF to access S3-compatible object stores. This topic describes how to configure the PXF connectors to these external data sources.

If you do not plan to use these PXF object store connectors, then you do not need to perform this procedure.

About Object Store Configuration

To access data in an object store, you must provide a server location and client credentials. When you configure a PXF object store connector, you add at least one named PXF server configuration for the connector as described in Configuring PXF Servers.

PXF provides a template configuration file for each object store connector. These template files are located in the `$PXF_CONF/templates/` directory.

Minio Server Configuration

The template configuration file for Minio is `$PXF_CONF/templates/minio-site.xml`. When you configure a Minio server, you must provide the following server configuration properties and replace the template values with your credentials:

Property	Description	Value
fs.s3a.endpoint	The Minio S3 endpoint to which to connect.	Your endpoint.
fs.s3a.access.key	The Minio account access key ID.	Your access key.
fs.s3a.secret.key	The secret key associated with the Minio access key ID.	Your secret key.

S3 Server Configuration

The template configuration file for S3 is `$PXF_CONF/templates/s3-site.xml`. When you configure an S3 server, you must provide the following server configuration properties and replace the template values with your credentials:

Property	Description	Value
fs.s3a.access.key	The AWS account access key ID.	Your access key.
fs.s3a.secret.key	The secret key associated with the AWS access key ID.	Your secret key.

If required, fine-tune PXF S3 connectivity by specifying properties identified in the S3A section of the Hadoop-AWS module documentation in your `s3-site.xml` server configuration file.

You can override the credentials for an S3 server configuration by directly specifying the S3 access ID and secret key via custom options in the CREATE EXTERNAL TABLE command LOCATION

clause. Refer to Overriding the S3 Server Configuration with DDL for additional information.

Configuring S3 Server-Side Encryption

PXF supports Amazon Web Service S3 Server-Side Encryption (SSE) for S3 files that you access with readable and writable Greenplum Database external tables that specify the `pxf` protocol and an `s3:*` profile. AWS S3 server-side encryption protects your data at rest; it encrypts your object data as it writes to disk, and transparently decrypts the data for you when you access it.

PXF supports the following AWS SSE encryption key management schemes:

- SSE with S3-Managed Keys (SSE-S3) - Amazon manages the data and master encryption keys.
- SSE with Key Management Service Managed Keys (SSE-KMS) - Amazon manages the data key, and you manage the encryption key in AWS KMS.
- SSE with Customer-Provided Keys (SSE-C) - You set and manage the encryption key.

Your S3 access key and secret key govern your access to all S3 bucket objects, whether the data is encrypted or not.

S3 transparently decrypts data during a read operation of an encrypted file that you access via a readable external table that is created by specifying the `pxf` protocol and an `s3:*` profile. No additional configuration is required.

To encrypt data that you write to S3 via this type of external table, you have two options:

- Configure the default SSE encryption key management scheme on a per-S3-bucket basis via the AWS console or command line tools (recommended).
- Configure SSE encryption options in your PXF S3 server `s3-site.xml` configuration file.

Configuring SSE via an S3 Bucket Policy (Recommended)

You can create S3 *Bucket Policy*(s) that identify the objects that you want to encrypt, the encryption key management scheme, and the write actions permitted on those objects. Refer to Protecting Data Using Server-Side Encryption in the AWS S3 documentation for more information about the SSE encryption key management schemes. How Do I Enable Default Encryption for an S3 Bucket? describes how to set default encryption bucket policies.

Specifying SSE Options in a PXF S3 Server Configuration

You must include certain properties in `s3-site.xml` to configure server-side encryption in a PXF S3 server configuration. The properties and values that you add to the file are dependent upon the SSE encryption key management scheme.

SSE-S3

To enable SSE-S3 on any file that you write to any S3 bucket, set the following encryption algorithm property and value in the `s3-site.xml` file:

```
<property>
  <name>fs.s3a.server-side-encryption-algorithm</name>
  <value>AES256</value>
</property>
```

To enable SSE-S3 for a specific S3 bucket, use the property name variant that includes the bucket name. For example:

```
<property>
  <name>fs.s3a.bucket.YOUR_BUCKET1_NAME.server-side-encryption-algorithm</name>
  <value>AES256</value>
</property>
```

Replace `YOUR_BUCKET1_NAME` with the name of the S3 bucket.

SSE-KMS

To enable SSE-KMS on any file that you write to any S3 bucket, set both the encryption algorithm and encryption key ID. To set these properties in the `s3-site.xml` file:

```
<property>
  <name>fs.s3a.server-side-encryption-algorithm</name>
  <value>SSE-KMS</value>
</property>
<property>
  <name>fs.s3a.server-side-encryption.key</name>
  <value>YOUR_AWS_SSE_KMS_KEY_ARN</value>
</property>
```

Substitute `YOUR_AWS_SSE_KMS_KEY_ARN` with your key resource name. If you do not specify an encryption key, the default key defined in the Amazon KMS is used. Example KMS key: `arn:aws:kms:us-west-2:123456789012:key/1a23b456-7890-12cc-d345-6ef7890g12f3`.

Note: Be sure to create the bucket and the key in the same Amazon Availability Zone.

To enable SSE-KMS for a specific S3 bucket, use property name variants that include the bucket name. For example:

```
<property>
  <name>fs.s3a.bucket.YOUR_BUCKET2_NAME.server-side-encryption-algorithm</name>
  <value>SSE-KMS</value>
</property>
<property>
  <name>fs.s3a.bucket.YOUR_BUCKET2_NAME.server-side-encryption.key</name>
  <value>YOUR_AWS_SSE_KMS_KEY_ARN</value>
</property>
```

Replace `YOUR_BUCKET2_NAME` with the name of the S3 bucket.

SSE-C

To enable SSE-C on any file that you write to any S3 bucket, set both the encryption algorithm and the encryption key (base-64 encoded). All clients must share the same key.

To set these properties in the `s3-site.xml` file:

```
<property>
  <name>fs.s3a.server-side-encryption-algorithm</name>
  <value>SSE-C</value>
</property>
<property>
  <name>fs.s3a.server-side-encryption.key</name>
```

```
  <value>YOUR_BASE64-ENCODED_ENCRYPTION_KEY</value>
</property>
```

To enable SSE-C for a specific S3 bucket, use the property name variants that include the bucket name as described in the SSE-KMS example.

Example Server Configuration Procedure

Ensure that you have initialized PXF before you configure an object store connector server.

In this procedure, you name and add a PXF server configuration in the `$PXF_CONF/servers` directory on the Greenplum Database master host for the S3 Cloud Storage connector. You then use the `pxf cluster sync` command to sync the server configuration(s) to the Greenplum Database cluster.

1. Log in to your Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Choose a name for the server. You will provide the name to end users that need to reference files in the object store.

3. Create the `$PXF_CONF/servers/<server_name>` directory. For example, use the following command to create a server configuration for an S3 server named `s3srvcfg`:

   ```
   gpadmin@gpmaster$ mkdir $PXF_CONF/servers/s3srvcfg
   ```

4. Copy the PXF template file for S3 to the server configuration directory. For example:

   ```
   gpadmin@gpmaster$ cp $PXF_CONF/templates/s3-site.xml $PXF_CONF/servers/s3srvcfg
   /
   ```

5. Open the template server configuration file in the editor of your choice, and provide appropriate property values for your environment. For example:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <configuration>
       <property>
           <name>fs.s3a.access.key</name>
           <value>access_key_for_user1</value>
       </property>
       <property>
           <name>fs.s3a.secret.key</name>
           <value>secret_key_for_user1</value>
       </property>
       <property>
           <name>fs.s3a.fast.upload</name>
           <value>true</value>
       </property>
   </configuration>
   ```

6. Save your changes and exit the editor.

7. Use the `pxf cluster sync` command to copy the new server configuration to the Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

Configuring Connectors to Azure and Google Cloud Storage Object Stores (Optional)

You can use PXF to access Azure Data Lake, Azure Blob Storage, and Google Cloud Storage object stores. This topic describes how to configure the PXF connectors to these external data sources.

If you do not plan to use these PXF object store connectors, then you do not need to perform this procedure.

About Object Store Configuration

To access data in an object store, you must provide a server location and client credentials. When you configure a PXF object store connector, you add at least one named PXF server configuration for the connector as described in Configuring PXF Servers.

PXF provides a template configuration file for each object store connector. These template files are located in the `$PXF_CONF/templates/` directory.

Azure Blob Storage Server Configuration

The template configuration file for Azure Blob Storage is `$PXF_CONF/templates/wasbs-site.xml`. When you configure an Azure Blob Storage server, you must provide the following server configuration properties and replace the template value with your account name:

Property	Description	Value
fs.adl.oauth2.access.token.provider.type	The token type.	Must specify `ClientCredential`.
fs.azure.account.key.<YOUR_AZURE_BLOB_STORAGE_ACCOUNT_NAME>.blob.core.windows.net	The Azure account key.	Replace with your account key.
fs.AbstractFileSystem.wasbs.impl	The file system class name.	Must specify `org.apache.hadoop.fs.azure.Wasbs`.

Azure Data Lake Server Configuration

The template configuration file for Azure Data Lake is `$PXF_CONF/templates/adl-site.xml`. When you configure an Azure Data Lake server, you must provide the following server configuration properties and replace the template values with your credentials:

Property	Description	Value
fs.adl.oauth2.access.token.provider.type	The type of token.	Must specify `ClientCredential`.
fs.adl.oauth2.refresh.url	The Azure endpoint to which to connect.	Your refresh URL.
fs.adl.oauth2.client.id	The Azure account client ID.	Your client ID (UUID).

Property	Description	Value
fs.adl.oauth2.credential	The password for the Azure account client ID.	Your password.

Google Cloud Storage Server Configuration

The template configuration file for Google Cloud Storage is `$PXF_CONF/templates/gs-site.xml`. When you configure a Google Cloud Storage server, you must provide the following server configuration properties and replace the template values with your credentials:

Property	Description	Value
google.cloud.auth.service.account.enable	Enable service account authorization.	Must specify `true`.
google.cloud.auth.service.account.json.keyfile	The Google Storage key file.	Path to your key file.
fs.AbstractFileSystem.gs.impl	The file system class name.	Must specify `com.google.cloud.hadoop.fs.gcs.GoogleHadoopFS`.

Example Server Configuration Procedure

Ensure that you have initialized PXF before you configure an object store connector server.

In this procedure, you name and add a PXF server configuration in the `$PXF_CONF/servers` directory on the Greenplum Database master host for the Google Cloud Storate (GCS) connector. You then use the `pxf cluster sync` command to sync the server configuration(s) to the Greenplum Database cluster.

1. Log in to your Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Choose a name for the server. You will provide the name to end users that need to reference files in the object store.

3. Create the `$PXF_CONF/servers/<server_name>` directory. For example, use the following command to create a server configuration for a Google Cloud Storage server named `gs_public`:

   ```
   gpadmin@gpmaster$ mkdir $PXF_CONF/servers/gs_public
   ```

4. Copy the PXF template file for GCS to the server configuration directory. For example:

   ```
   gpadmin@gpmaster$ cp $PXF_CONF/templates/gs-site.xml $PXF_CONF/servers/gs_publi
   c/
   ```

5. Open the template server configuration file in the editor of your choice, and provide appropriate property values for your environment. For example, if your Google Cloud Storage key file is located in `/home/gpadmin/keys/gcs-account.key.json`:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property>
        <name>google.cloud.auth.service.account.enable</name>
        <value>true</value>
    </property>
    <property>
        <name>google.cloud.auth.service.account.json.keyfile</name>
        <value>/home/gpadmin/keys/gcs-account.key.json</value>
    </property>
    <property>
        <name>fs.AbstractFileSystem.gs.impl</name>
        <value>com.google.cloud.hadoop.fs.gcs.GoogleHadoopFS</value>
    </property>
</configuration>
```

6. Save your changes and exit the editor.
7. Use the `pxf cluster sync` command to copy the new server configurations to the Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

Configuring the JDBC Connector (Optional)

You can use PXF to access an external SQL database including MySQL, ORACLE, Microsoft SQL Server, DB2, PostgreSQL, Hive, and Apache Ignite. This topic describes how to configure the PXF JDBC Connector to access these external data sources.

If you do not plan to use the PXF JDBC Connector, then you do not need to perform this procedure.

About JDBC Configuration

To access data in an external SQL database with the PXF JDBC Connector, you must:

- Register a compatible JDBC driver JAR file
- Specify the JDBC driver class name, database URL, and client credentials

In previous releases of Greenplum Database, you may have specified the JDBC driver class name, database URL, and client credentials via options in the `CREATE EXTERNAL TABLE` command. PXF now supports file-based server configuration for the JDBC Connector. This configuration, described below, allows you to specify these options and credentials in a file.

Note: PXF external tables that you previously created that directly specified the JDBC connection options will continue to work. If you want to move these tables to use JDBC file-based server configuration, you must create a server configuration, drop the external tables, and then recreate the tables specifying an appropriate `SERVER=<server_name>` clause.

JDBC Driver JAR Registration

The PXF JDBC Connector is installed with the `postgresql-42.2.5.jar` JAR file. If you require a different JDBC driver, ensure that you install the JDBC driver JAR file for the external SQL database

in the `$PXF_CONF/lib` directory on each segment host. Be sure to install JDBC driver JAR files that are compatible with your JRE version. See Registering PXF JAR Dependencies for additional information.

JDBC Server Configuration

When you configure the PXF JDBC Connector, you add at least one named PXF server configuration for the connector as described in Configuring PXF Servers. You can also configure one or more statically-defined queries to run against the remote SQL database.

PXF provides a template configuration file for the JDBC Connector. This server template configuration file, located in `$PXF_CONF/templates/jdbc-site.xml`, identifies properties that you can configure to establish a connection to the external SQL database. The template also includes optional properties that you can set before executing query or insert commands in the external database session.

The required properties in the `jdbc-site.xml` server template file follow:

Property	Description	Value
jdbc.driver	Class name of the JDBC driver.	The JDBC driver Java class name; for example `org.postgresql.Driver`.
jdbc.url	The URL that the JDBC driver uses to connect to the database.	The database connection URL (database-specific); for example `jdbc:postgresql://phost:pport/pdatabase`.
jdbc.user	The database user name.	The user name for connecting to the database.
jdbc.password	The password for `jdbc.user`.	The password for connecting to the database.

When you configure a PXF JDBC server, you specify the external database user credentials to PXF in clear text in a configuration file.

Connection-Level Properties

To set additional JDBC connection-level properties, add `jdbc.connection.property.<CPROP_NAME>` properties to `jdbc-site.xml`. PXF passes these properties to the JDBC driver when it establishes the connection to the external SQL database (`DriverManager.getConnection()`).

Replace `<CPROP_NAME>` with the connection property name and specify its value:

Property	Description	Value
jdbc.connection.property.<CPROP_NAME>	The name of a property (<CPROP_NAME>) to pass to the JDBC driver when PXF establishes the connection to the external SQL database.	The value of the <CPROP_NAME> property.

Example: To set the `createDatabaseIfNotExist` connection property on a JDBC connection to a PostgreSQL database, include the following property block in `jdbc-site.xml`:

```
<property>
    <name>jdbc.connection.property.createDatabaseIfNotExist</name>
    <value>true</value>
 </property>
```

Ensure that the JDBC driver for the external SQL database supports any connection-level property that you specify.

Connection Transaction Isolation Property

The SQL standard defines four transaction isolation levels. The level that you specify for a given connection to an external SQL database determines how and when the changes made by one transaction executed on the connection are visible to another.

The PXF JDBC Connector exposes an optional server configuration property named `jdbc.connection.transactionIsolation` that enables you to specify the transaction isolation level. PXF sets the level (`setTransactionIsolation()`) just after establishing the connection to the external SQL database.

The JDBC Connector supports the following `jdbc.connection.transactionIsolation` property values:

SQL Level	PXF Property Value
Read uncommitted	READ_UNCOMMITTED
Read committed	READ_COMMITTED
Repeatable Read	REPEATABLE_READ
Serializable	SERIALIZABLE

For example, to set the transaction isolation level to *Read uncommitted*, add the following property block to the `jdbc-site.xml` file:

```
<property>
    <name>jdbc.connection.transactionIsolation</name>
    <value>READ_UNCOMMITTED</value>
</property>
```

Different SQL databases support different transaction isolation levels. Ensure that the external database supports the level that you specify.

Statement-Level Properties

The PXF JDBC Connector executes a query or insert command on an external SQL database table in a *statement*. The Connector exposes properties that enable you to configure certain aspects of the statement before the command is executed in the external database. The Connector supports the following statement-level properties:

Property	Description	Value
jdbc.statement.batchSize	The number of rows to write to the external database table in a batch.	The number of rows. The default write batch size is 100.
jdbc.statement.fetchSize	The number of rows to fetch/buffer when reading from the external database table.	The number of rows. The default read fetch size is 1000.
jdbc.statement.queryTimeout	The amount of time (in seconds) the JDBC driver waits for a statement to execute. This timeout applies to statements created for both read and write operations.	The timeout duration in seconds. The default wait time is unlimited.

PXF uses the default value for any statement-level property that you do not explicitly configure.

Example: To set the read fetch size to 5000, add the following property block to `jdbc-site.xml`:

```
<property>
    <name>jdbc.statement.fetchSize</name>
    <value>5000</value>
</property>
```

Ensure that the JDBC driver for the external SQL database supports any statement-level property that you specify.

Session-Level Properties

To set session-level properties, add the `jdbc.session.property.<SPROP_NAME>` property to `jdbc-site.xml`. PXF will `SET` these properties in the external database before executing a query.

Replace `<SPROP_NAME>` with the session property name and specify its value:

Property	Description	Value
jdbc.session.property.<SPROP_NAME>	The name of a session property (<SPROP_NAME>) to set before query execution.	The value of the <SPROP_NAME> property.

Note: The PXF JDBC Connector passes both the session property name and property value to the external SQL database exactly as specified in the `jdbc-site.xml` server configuration file. To limit the potential threat of SQL injection, the Connector rejects any property name or value that contains the `;`, `\n`, `\b`, or `\0` characters.

The PXF JDBC Connector handles the session property `SET` syntax for all supported external SQL databases.

Example: To set the `search_path` parameter before running a query in a PostgreSQL database, add the following property block to `jdbc-site.xml`:

```
<property>
    <name>jdbc.session.property.search_path</name>
    <value>public</value>
</property>
```

Ensure that the JDBC driver for the external SQL database supports any property that you specify.

About JDBC Connection Pooling

The PXF JDBC Connector uses JDBC connection pooling implemented by HikariCP. When a user queries or writes to an external table, the Connector establishes a connection pool for the associated server configuration the first time that it encounters a unique combination of `jdbc.url`, `jdbc.user`, `jdbc.password`, connection property, and pool property settings. The Connector reuses connections in the pool subject to certain connection and timeout settings.

One or more connection pools may exist for a given server configuration, and user access to different external tables specifying the same server may share a connection pool.

Note: If you have enabled JDBC user impersonation in a server configuration, the JDBC Connector creates a separate connection pool for each Greenplum Database user that accesses any external table specifying that server configuration.

The `jdbc.pool.enabled` property governs JDBC connection pooling for a server configuration. Connection pooling is enabled by default. To disable JDBC connection pooling for a server configuration, set the property to false:

```
<property>
    <name>jdbc.pool.enabled</name>
    <value>false</value>
</property>
```

If you disable JDBC connection pooling for a server configuration, PXF does not reuse JDBC connections for that server. PXF creates a connection to the remote database for every partition of a query, and closes the connection when the query for that partition completes.

PXF exposes connection pooling properties that you can configure in a JDBC server definition. These properties are named with the `jdbc.pool.property.` prefix and *apply to each PXF JVM*. The JDBC Connector automatically sets the following connection pool properties and default values:

Property	Description	Default Value
jdbc.pool.property.maximumPoolSize	The maximum number of connections to the database backend.	5
jdbc.pool.property.connectionTimeout	The maximum amount of time, in milliseconds, to wait for a connection from the pool.	30000
jdbc.pool.property.idleTimeout	The maximum amount of time, in milliseconds, after which an inactive connection is considered idle.	30000
jdbc.pool.property.minimumIdle	The minimum number of idle connections maintained in the connection pool.	0

You can set other HikariCP-specific connection pooling properties for a server configuration by specifying `jdbc.pool.property.<HIKARICP_PROP_NAME>` and the desired value in the `jdbc-site.xml` configuration file for the server. Also note that the JDBC Connector passes along any property that you specify with a `jdbc.connection.property.` prefix when it requests a connection from the JDBC `DriverManager`. Refer to Connection-Level Properties above.

Tuning the Maximum Connection Pool Size

To not exceed the maximum number of connections allowed by the target database, and at the same time ensure that each PXF JVM services a fair share of the JDBC connections, determine the maximum value of `maxPoolSize` based on the size of the Greenplum Database cluster as follows:

```
max_conns_allowed_by_remote_db / #_greenplum_segment_hosts
```

For example, if your Greenplum Database cluster has 16 segment hosts and the target database allows 160 concurrent connections, calculate `maxPoolSize` as follows:

```
160 / 16 = 10
```

In practice, you may choose to set `maxPoolSize` to a lower value, since the number of concurrent connections per JDBC query depends on the number of partitions used in the query. When a query uses no partitions, a single PXF JVM services the query. If a query uses 12 partitions, PXF establishes 12 concurrent JDBC connections to the remote database. Ideally, these connections are distributed

equally among the PXF JVMs, but that is not guaranteed.

JDBC User Impersonation

The PXF JDBC Connector uses the `jdbc.user` setting or information in the `jdbc.url` to determine the identity of the user to connect to the external data store. When PXF JDBC user impersonation is disabled (the default), the behavior of the JDBC Connector is further dependent upon the external data store. For example, if you are using the JDBC Connector to access Hive, the Connector uses the settings of certain Hive authentication and impersonation properties to determine the user. You may be required to provide a `jdbc.user` setting, or add properties to the `jdbc.url` setting in the server `jdbc-site.xml` file. Refer to Configuring Hive Access via the JDBC Connector for more information on this procedure.

When you enable PXF JDBC user impersonation, the PXF JDBC Connector accesses the external data store on behalf of a Greenplum Database end user. The Connector uses the name of the Greenplum Database user that accesses the PXF external table to try to connect to the external data store.

When you enable JDBC user impersonation for a PXF server, PXF overrides the value of a `jdbc.user` property setting defined in either `jdbc-site.xml` or `<greenplum_user_name>-user.xml`, or specified in the external table DDL, with the Greenplum Database user name. For user impersonation to work effectively when the external data store requires passwords to authenticate connecting users, you must specify the `jdbc.password` setting for each user that can be impersonated in that user's `<greenplum_user_name>-user.xml` property override file. Refer to Configuring a PXF User for more information about per-server, per-Greenplum-user configuration.

The `pxf.service.user.impersonation` property in the `jdbc-site.xml` configuration file governs JDBC user impersonation.

In previous versions of Greenplum Database, you configured JDBC user impersonation via the now deprecated `pxf.impersonation.jdbc` property setting in the `jdbc-site.xml` configuration file.

Example Configuration Procedure

By default, PXF JDBC user impersonation is disabled. Perform the following procedure to turn PXF user impersonation on or off for a JDBC server configuration.

1. Log in to your Greenplum Database master node as the administrative user:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Identify the name of the PXF JDBC server configuration that you want to update.

3. Navigate to the server configuration directory. For example, if the server is named `mysqldb`:

   ```
   gpadmin@gpmaster$ cd $PXF_CONF/servers/mysqldb
   ```

4. Open the `jdbc-site.xml` file in the editor of your choice, and add or uncomment the user impersonation property and setting. For example, if you require user impersonation for this server configuration, set the `pxf.service.user.impersonation` property to `true`:

   ```
   <property>
       <name>pxf.service.user.impersonation</name>
   ```

```
    <value>true</value>
</property>
```

5. Save the `jdbc-site.xml` file and exit the editor.
6. Use the `pxf cluster sync` command to synchronize the PXF JDBC server configuration to your Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

About Session Authorization

Certain SQL databases, including PostgreSQL and DB2, allow a privileged user to change the effective database user that runs commands in a session. You might take advantage of this feature if, for example, you connect to the remote database as a proxy user and want to switch session authorization after establishing the database connection.

In databases that support it, you can configure a session property to switch the effective user. For example, in DB2, you use the `SET SESSION_USER <username>` command to switch the effective DB2 user. If you configure the DB2 `session_user` variable via a PXF session-level property (`jdbc.session.property.<SPROP_NAME>`) in your `jdbc-site.xml` file, PXF runs this command for you.

For example, to switch the effective DB2 user to the user named `bill`, you configure your `jdbc-site.xml` as follows:

```
<property>
    <name>jdbc.session.property.session_user</name>
    <value>bill</value>
</property>
```

After establishing the database connection, PXF implicitly runs the following command to set the `session_user` DB2 session variable to the value that you configured:

```
SET SESSION_USER = bill
```

PXF recognizes a synthetic property value, `${pxf.session.user}`, that identifies the Greenplum Database user name. You may choose to use this value when you configure a property that requires a value that changes based on the Greenplum user running the session.

A scenario where you might use `${pxf.session.user}` is when you authenticate to the remote SQL database with Kerberos, the primary component of the Kerberos principal identifies the Greenplum Database user name, and you want to run queries in the remote database using this effective user name. For example, if you are accessing DB2, you would configure your `jdbc-site.xml` to specify the Kerberos `securityMechanism` and `KerberosServerPrincipal`, and then set the `session_user` variable as follows:

```
<property>
    <name>jdbc.session.property.session_user</name>
    <value>${pxf.session.user}</value>
</property>
```

With this configuration, PXF `SET`s the DB2 `session_user` variable to the current Greenplum Database user name, and runs subsequent operations on the DB2 table as that user.

Session Authorization Considerations for Connection Pooling

When PXF performs session authorization on your behalf and JDBC connection pooling is enabled (the default), you may choose to set the `jdbc.pool.qualifier` property. Setting this property instructs PXF to include the property value in the criteria that it uses to create and reuse connection pools. In practice, you would not set this to a fixed value, but rather to a value that changes based on the user/session/transaction, etc. When you set this property to `${pxf.session.user}`, PXF includes the Greenplum Database user name in the criteria that it uses to create and re-use connection pools. The default setting is no qualifier.

To make use of this feature, add or uncomment the following property block in `jdbc-site.xml` to prompt PXF to include the Greenplum user name in connection pool creation/reuse criteria:

```
<property>
    <name>jdbc.pool.qualifier</name>
    <value>${pxf.session.user}</value>
</property>
```

JDBC Named Query Configuration

A PXF *named query* is a static query that you configure, and that PXF runs in the remote SQL database.

To configure and use a PXF JDBC named query:

1. You define the query in a text file.
2. You provide the query name to Greenplum Database users.
3. The Greenplum Database user references the query in a Greenplum Database external table definition.

PXF runs the query each time the user invokes a `SELECT` command on the Greenplum Database external table.

Defining a Named Query

You create a named query by adding the query statement to a text file that has the following naming format: `<query_name>.sql`. You can define one or more named queries for a JDBC server configuration. Each query must reside in a separate text file.

You must place a query text file in the PXF JDBC server configuration directory from which it will be accessed. If you want to make the query available to more than one JDBC server configuration, you must copy the query text file to the configuration directory for each JDBC server.

The query text file must contain a single query that you want to run in the remote SQL database. You must construct the query in accordance with the syntax supported by the database.

For example, if a MySQL database has a `customers` table and an `orders` table, you could include the following SQL statement in a query text file:

```
SELECT c.name, c.city, sum(o.amount) AS total, o.month
  FROM customers c JOIN orders o ON c.id = o.customer_id
  WHERE c.state = 'CO'
```

```
GROUP BY c.name, c.city, o.month
```

You may optionally provide the ending semicolon (`;`) for the SQL statement.

Query Naming

The Greenplum Database user references a named query by specifying the query file name without the extension. For example, if you define a query in a file named `report.sql`, the name of that query is `report`.

Named queries are associated with a specific JDBC server configuration. You will provide the available query names to the Greenplum Database users that you allow to create external tables using the server configuration.

Referencing a Named Query

The Greenplum Database user specifies `query:<query_name>` rather than the name of a remote SQL database table when they create the external table. For example, if the query is defined in the file `$PXF_CONF/servers/mydb/report.sql`, the `CREATE EXTERNAL TABLE LOCATION` clause would include the following components:

```
LOCATION ('pxf://query:report?PROFILE=JDBC&SERVER=mydb ...')
```

Refer to About Using Named Queries for information about using PXF JDBC named queries.

Overriding the JDBC Server Configuration

You can override the JDBC server configuration by directly specifying certain JDBC properties via custom options in the `CREATE EXTERNAL TABLE` command `LOCATION` clause. Refer to Overriding the JDBC Server Configuration via DDL for additional information.

Configuring Access to Hive

You can use the JDBC Connector to access Hive. Refer to Configuring the JDBC Connector for Hive Access for detailed information on this configuration procedure.

Example Configuration Procedure

Ensure that you have initialized PXF before you configure a JDBC Connector server.

In this procedure, you name and add a PXF JDBC server configuration for a PostgreSQL database and synchronize the server configuration(s) to the Greenplum Database cluster.

1. Log in to your Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Choose a name for the JDBC server. You will provide the name to Greenplum users that you choose to allow to reference tables in the external SQL database as the configured user.

   Note: The server name `default` is reserved.

3. Create the `$PXF_CONF/servers/<server_name>` directory. For example, use the following

command to create a JDBC server configuration named `pg_user1_testdb`:

```
gpadmin@gpmaster$ mkdir $PXF_CONF/servers/pg_user1_testdb
```

4. Copy the PXF JDBC server template file to the server configuration directory. For example:

```
gpadmin@gpmaster$ cp $PXF_CONF/templates/jdbc-site.xml $PXF_CONF/servers/pg_use
r1_testdb/
```

5. Open the template server configuration file in the editor of your choice, and provide appropriate property values for your environment. For example, if you are configuring access to a PostgreSQL database named `testdb` on a PostgreSQL instance running on the host named `pgserverhost` for the user named `user1`:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property>
        <name>jdbc.driver</name>
        <value>org.postgresql.Driver</value>
    </property>
    <property>
        <name>jdbc.url</name>
        <value>jdbc:postgresql://pgserverhost:5432/testdb</value>
    </property>
    <property>
        <name>jdbc.user</name>
        <value>user1</value>
    </property>
    <property>
        <name>jdbc.password</name>
        <value>changeme</value>
    </property>
</configuration>
```

6. Save your changes and exit the editor.

7. Use the `pxf cluster sync` command to copy the new server configuration to the Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

Configuring the JDBC Connector for Hive Access (Optional)

You can use the PXF JDBC Connector to retrieve data from Hive. You can also use a JDBC named query to submit a custom SQL query to Hive and retrieve the results using the JDBC Connector.

This topic describes how to configure the PXF JDBC Connector to access Hive. When you configure Hive access with JDBC, you must take into account the Hive user impersonation setting, as well as whether or not the Hadoop cluster is secured with Kerberos.

If you do not plan to use the PXF JDBC Connector to access Hive, then you do not need to perform this procedure.

JDBC Server Configuration

The PXF JDBC Connector is installed with the JAR files required to access Hive via JDBC, `hive-jdbc-<version>.jar` and `hive-service-<version>.jar`, and automatically registers these JARs.

When you configure a PXF JDBC server for Hive access, you must specify the JDBC driver class name, database URL, and client credentials just as you would when configuring a client connection to an SQL database.

To access Hive via JDBC, you must specify the following properties and values in the `jdbc-site.xml` server configuration file:

Property	Value
jdbc.driver	org.apache.hive.jdbc.HiveDriver
jdbc.url	jdbc:hive2://<hiveserver2_host>:<hiveserver2_port>/<database>

The value of the HiveServer2 authentication (`hive.server2.authentication`) and impersonation (`hive.server2.enable.doAs`) properties, and whether or not the Hive service is utilizing Kerberos authentication, will inform the setting of other JDBC server configuration properties. These properties are defined in the `hive-site.xml` configuration file in the Hadoop cluster. You will need to obtain the values of these properties.

The following table enumerates the Hive2 authentication and impersonation combinations supported by the PXF JDBC Connector. It identifies the possible Hive user identities and the JDBC server configuration required for each.

Table heading key:

- *authentication* -> Hive `hive.server2.authentication` Setting
- *enable.doAs* -> Hive `hive.server2.enable.doAs` Setting
- *User Identity* -> Identity that HiveServer2 will use to access data
- *Configuration Required* -> PXF JDBC Connector or Hive configuration required for *User Identity*

authentication	enable.doAs	User Identity	Configuration Required
`NOSASL`	n/a	No authentication	Must set `jdbc.connection.property.auth` = `noSasl`.
`NONE`, or not specified	`TRUE`	User name that you provide	Set `jdbc.user`.
`NONE`, or not specified	`TRUE`	Greenplum user name	Set `pxf.service.user.impersonation` to `true` in `jdbc-site.xml`.
`NONE`, or not specified	`FALSE`	Name of the user who started Hive, typically `hive`	None
`KERBEROS`	`TRUE`	Identity provided in the PXF Kerberos principal, typically `gpadmin`	Must set `hadoop.security.authentication` to `kerberos` in `jdbc-site.xml`.
`KERBEROS`	`TRUE`	User name that you provide	Set `hive.server2.proxy.user` in `jdbc.url` and set `hadoop.security.authentication` to `kerberos` in `jdbc-site.xml`.

authentication	enable.doAs	User Identity	Configuration Required
KERBEROS	TRUE	Greenplum user name	Set pxf.service.user.impersonation to true and hadoop.security.authentication to kerberos in jdbc-site.xml.
KERBEROS	FALSE	Identity provided in the jdbc.url principal parameter, typically hive	Must set hadoop.security.authentication to kerberos in jdbc-site.xml.

Note: There are additional configuration steps required when Hive utilizes Kerberos authentication.

Example Configuration Procedure

Perform the following procedure to configure a PXF JDBC server for Hive:

1. Log in to your Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Choose a name for the JDBC server.

3. Create the $PXF_CONF/servers/<server_name> directory. For example, use the following command to create a JDBC server configuration named hivejdbc1:

   ```
   gpadmin@gpmaster$ mkdir $PXF_CONF/servers/hivejdbc1
   ```

4. Navigate to the server configuration directory. For example:

   ```
   gpadmin@gpmaster$ cd $PXF_CONF/servers/hivejdbc1
   ```

5. Copy the PXF JDBC server template file to the server configuration directory. For example:

   ```
   gpadmin@gpmaster$ cp $PXF_CONF/templates/jdbc-site.xml .
   ```

6. When you access Hive secured with Kerberos, you also need to specify configuration properties in the pxf-site.xml file. *If this file does not yet exist in your server configuration*, copy the pxf-site.xml template file to the server config directory. For example:

   ```
   gpadmin@gpmaster$ cp $PXF_CONF/templates/pxf-site.xml .
   ```

7. Open the jdbc-site.xml file in the editor of your choice and set the jdbc.driver and jdbc.url properties. Be sure to specify your Hive host, port, and database name:

   ```
   <property>
       <name>jdbc.driver</name>
       <value>org.apache.hive.jdbc.HiveDriver</value>
   </property>
   <property>
       <name>jdbc.url</name>
       <value>jdbc:hive2://<hiveserver2_host>:<hiveserver2_port>/<database></value
   >
   </property>
   ```

8. Obtain the hive-site.xml file from your Hadoop cluster and examine the file.

9. If the `hive.server2.authentication` property in `hive-site.xml` is set to `NOSASL`, HiveServer2 performs no authentication. Add the following connection-level property to `jdbc-site.xml`:

    ```
    <property>
        <name>jdbc.connection.property.auth</name>
        <value>noSasl</value>
    </property>
    ```

    Alternatively, you may choose to add `;auth=noSasl` to the `jdbc.url`.

10. If the `hive.server2.authentication` property in `hive-site.xml` is set to `NONE`, or the property is not specified, you must set the `jdbc.user` property. The value to which you set the `jdbc.user` property is dependent upon the `hive.server2.enable.doAs` impersonation setting in `hive-site.xml`:

    1. If `hive.server2.enable.doAs` is set to `TRUE` (the default), Hive runs Hadoop operations on behalf of the user connecting to Hive. *Choose/perform one of the following options*:

        Set `jdbc.user` to specify the user that has read permission on all Hive data accessed by Greenplum Database. For example, to connect to Hive and run all requests as user `gpadmin`:

        ```
        <property>
            <name>jdbc.user</name>
            <value>gpadmin</value>
        </property>
        ```

        Or, turn on JDBC server-level user impersonation so that PXF automatically uses the Greenplum Database user name to connect to Hive; uncomment the `pxf.service.user.impersonation` property in `jdbc-site.xml` and set the value to `true:

        ```
        <property>
            <name>pxf.service.user.impersonation</name>
            <value>true</value>
        </property>
        ```

        If you enable JDBC impersonation in this manner, you must not specify a `jdbc.user` nor include the setting in the `jdbc.url`.

    2. If required, create a PXF user configuration file as described in Configuring a PXF User to manage the password setting.

    3. If `hive.server2.enable.doAs` is set to `FALSE`, Hive runs Hadoop operations as the user who started the HiveServer2 process, usually the user `hive`. PXF ignores the `jdbc.user` setting in this circumstance.

11. If the `hive.server2.authentication` property in `hive-site.xml` is set to `KERBEROS`:

    1. Identify the name of the server configuration.

    2. Ensure that you have configured Kerberos authentication for PXF as described in Configuring PXF for Secure HDFS, and that you have specified the Kerberos principal and keytab in the `pxf-site.xml` properties as described in the procedure.

3. Comment out the `pxf.service.user.impersonation` property in the `pxf-site.xml` file. If you require user impersonation, you will uncomment and set the property in an upcoming step.)

4. Uncomment the `hadoop.security.authentication` setting in `$PXF_CONF/servers/<name>/jdbc-site.xml`:

```
<property>
    <name>hadoop.security.authentication</name>
    <value>kerberos</value>
</property>
```

5. Add the `saslQop` property to `jdbc.url`, and set it to match the `hive.server2.thrift.sasl.qop` property setting in `hive-site.xml`. For example, if the `hive-site.xml` file includes the following property setting:

```
<property>
    <name>hive.server2.thrift.sasl.qop</name>
    <value>auth-conf</value>
</property>
```

   You would add `;saslQop=auth-conf` to the `jdbc.url`.

6. Add the HiverServer2 `principal` name to the `jdbc.url`. For example:

```
jdbc:hive2://hs2server:10000/default;principal=hive/hs2server@REALM;saslQop=
auth-conf
```

7. If `hive.server2.enable.doAs` is set to `TRUE` (the default), Hive runs Hadoop operations on behalf of the user connecting to Hive. *Choose/perform one of the following options*:

   Do not specify any additional properties. In this case, PXF initiates all Hadoop access with the identity provided in the PXF Kerberos principal (usually `gpadmin`).

   Or, set the `hive.server2.proxy.user` property in the `jdbc.url` to specify the user that has read permission on all Hive data. For example, to connect to Hive and run all requests as the user named `integration` use the following `jdbc.url`:

```
jdbc:hive2://hs2server:10000/default;principal=hive/hs2server@REALM;saslQ
op=auth-conf;hive.server2.proxy.user=integration
```

   Or, enable PXF JDBC impersonation in the `pxf-site.xml` file so that PXF automatically uses the Greenplum Database user name to connect to Hive. Add or uncomment the `pxf.service.user.impersonation` property and set the value to `true`. For example:

```
<property>
    <name>pxf.service.user.impersonation</name>
    <value>true</value>
</property>
```

   If you enable JDBC impersonation, you must not explicitly specify a

`hive.server2.proxy.user` in the `jdbc.url`.

8. If required, create a PXF user configuration file to manage the password setting.
9. If `hive.server2.enable.doAs` is set to `FALSE`, Hive runs Hadoop operations with the identity provided by the PXF Kerberos principal (usually `gpadmin`).

12. Save your changes and exit the editor.
13. Use the `pxf cluster sync` command to copy the new server configuration to the Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

Configuring the PXF Agent Host and Port (Optional)

By default, a PXF agent started on a segment host listens on port number `5888` on `localhost`. You can configure PXF to start on a different port number, or use a different hostname or IP address. To change the default configuration, you will set one or both of the environment variables identified below:

Environment Variable	Description
PXF_HOST	The name of the host or IP address. The default host name is `localhost`.
PXF_PORT	The port number on which the PXF agent listens for requests on the host. The default port number is `5888`.

Set the environment variables in the `gpadmin` user's `.bashrc` shell login file on each segment host.

You must restart both Greenplum Database and PXF when you configure the agent host and/or port in this manner. Consider performing this configuration during a scheduled down time.

Procedure

Perform the following procedure to configure the PXF agent host and/or port number on one or more Greenplum Database segment hosts:

1. Log in to your Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. For each Greenplum Database segment host:
    1. Identify the host name or IP address of the PXF agent.
    2. Identify the port number on which you want the PXF agent to run.
    3. Log in to the Greenplum Database segment host:

```
$ ssh gpadmin@<seghost>
```

    4. Open the `~/.bashrc` file in the editor of your choice.
    5. Set the `PXF_HOST` and/or `PXF_PORT` environment variables. For example, to set the PXF agent port number to 5998, add the following to the `.bashrc` file:

```
export PXF_PORT=5998
```

6. Save the file and exit the editor.

3. Restart Greenplum Database as described in Restarting Greenplum Database in the Greenplum Documentation.
4. Restart PXF on each Greenplum Database segment host as described in Restarting PXF.

Starting, Stopping, and Restarting PXF

PXF provides two management commands:

- `pxf cluster` - manage all PXF service instances in the Greenplum Database cluster
- `pxf` - manage the PXF service instance on a specific Greenplum Database host

The `pxf cluster` command supports `init`, `reset`, `start`, `restart`, `status`, `stop`, and `sync` subcommands. When you run a `pxf cluster` subcommand on the Greenplum Database master host, you perform the operation on all segment hosts in the Greenplum Database cluster. PXF also runs the `init`, `register`, and `sync` commands on the standby master host.

The `pxf` command supports `init`, `reset`, `start`, `stop`, `restart`, `status`, and `register` operations. These operations run locally. That is, if you want to start or stop the PXF agent on a specific Greenplum Database segment host, you log in to the host and run the command.

The procedures in this topic assume that you have added the `$PXF_HOME/bin` directory to your `$PATH`.

Starting PXF

After initializing PXF, you must start PXF on each segment host in your Greenplum Database cluster. The PXF service, once started, runs as the `gpadmin` user on default port 5888. Only the `gpadmin` user can start and stop the PXF service.

If you want to change the default PXF configuration, you must update the configuration before you start PXF.

`$PXF_CONF/conf` includes these user-customizable configuration files:

- `pxf-env.sh` - runtime configuration parameters
- `pxf-log4j.properties` - logging configuration parameters
- `pxf-profiles.xml` - custom profile definitions

The `pxf-env.sh` file exposes the following PXF runtime configuration parameters:

Parameter	Description	Default Value
JAVA_HOME	The Java JRE home directory.	/usr/java/default
PXF_LOGDIR	The PXF log directory.	$PXF_CONF/logs
PXF_JVM_OPTS	Default options for the PXF Java virtual machine.	-Xmx2g -Xms1g
PXF_MAX_THREADS	Default for the maximum number of PXF threads.	200
PXF_FRAGMENTER_CACHE	Enable/disable fragment caching.	Enabled

Parameter	Description	Default Value
PXF_OOM_KILL	Enable/disable PXF auto-kill on OutOfMemoryError.	Enabled
PXF_OOM_DUMP_PATH	Absolute pathname to dump file generated on OOM.	No dump file
PXF_KEYTAB	The absolute path to the PXF service Kerberos principal keytab file. *Deprecated*; specify the keytab in a server-specific `pxf-site.xml` file.	$PXF_CONF/keytabs/pxf.service.keytab
PXF_PRINCIPAL	The PXF service Kerberos principal. *Deprecated*; specify the principal in a server-specific `pxf-site.xml` file.	gpadmin/_HOST@EXAMPLE.COM
PXF_USER_IMPERSONATION	Enable/disable end user identity impersonation. Deprecated; enable/disable impersonation in a server-specific `pxf-site.xml` file.	true

You must synchronize any changes that you make to `pxf-env.sh`, `pxf-log4j.properties`, or `pxf-profiles.xml` to the Greenplum Database cluster, and (re)start PXF on each segment host.

Prerequisites

Before you start PXF in your Greenplum Database cluster, ensure that:

- Your Greenplum Database cluster is up and running.
- You have previously initialized PXF.

Procedure

Perform the following procedure to start PXF on each segment host in your Greenplum Database cluster.

1. Log in to the Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Run the `pxf cluster start` command to start PXF on each segment host:

   ```
   gpadmin@gpmaster$ pxf cluster start
   ```

Stopping PXF

If you must stop PXF, for example if you are upgrading PXF, you must stop PXF on each segment host in your Greenplum Database cluster. Only the `gpadmin` user can stop the PXF service.

Prerequisites

Before you stop PXF in your Greenplum Database cluster, ensure that your Greenplum Database cluster is up and running.

Procedure

Perform the following procedure to stop PXF on each segment host in your Greenplum Database cluster.

1. Log in to the Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. Run the `pxf cluster stop` command to stop PXF on each segment host:

```
gpadmin@gpmaster$ pxf cluster stop
```

Restarting PXF

If you must restart PXF, for example if you updated PXF user configuration files in `$PXF_CONF/conf`, you run `pxf cluster restart` to stop, and then start, PXF on all segment hosts in your Greenplum Database cluster.

Only the `gpadmin` user can restart the PXF service.

Prerequisites

Before you restart PXF in your Greenplum Database cluster, ensure that your Greenplum Database cluster is up and running.

Procedure

Perform the following procedure to restart PXF in your Greenplum Database cluster.

1. Log in to the Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. Restart PXF:

```
gpadmin@gpmaster$ pxf cluster restart
```

Granting Users Access to PXF

The Greenplum Platform Extension Framework (PXF) implements a protocol named `pxf` that you can use to create an external table that references data in an external data store. The PXF protocol and Java service are packaged as a Greenplum Database extension.

You must enable the PXF extension in each database in which you plan to use the framework to access external data. You must also explicitly `GRANT` permission to the `pxf` protocol to those users/roles who require access.

Enabling PXF in a Database

You must explicitly register the PXF extension in each Greenplum Database in which you plan to use the extension. You must have Greenplum Database administrator privileges to register an extension.

Perform the following procedure for ***each*** database in which you want to use PXF:

1. Connect to the database as the `gpadmin` user:

   ```
   gpadmin@gpmaster$ psql -d <dbname> -U gpadmin
   ```

2. Create the PXF extension. You must have Greenplum Database administrator privileges to create an extension. For example:

   ```
   dbname=# CREATE EXTENSION pxf;
   ```

   Creating the `pxf` extension registers the `pxf` protocol and the call handlers required for PXF to access external data.

Disabling PXF in a Database

When you no longer want to use PXF on a specific database, you must explicitly drop the PXF extension for that database. You must have Greenplum Database administrator privileges to drop an extension.

1. Connect to the database as the `gpadmin` user:

   ```
   gpadmin@gpmaster$ psql -d <dbname> -U gpadmin
   ```

2. Drop the PXF extension:

   ```
   dbname=# DROP EXTENSION pxf;
   ```

   The `DROP` command fails if there are any currently defined external tables using the `pxf` protocol. Add the `CASCADE` option if you choose to forcibly remove these external tables.

Granting a Role Access to PXF

To read external data with PXF, you create an external table with the `CREATE EXTERNAL TABLE` command that specifies the `pxf` protocol. You must specifically grant `SELECT` permission to the `pxf` protocol to all non-`SUPERUSER` Greenplum Database roles that require such access.

To grant a specific role access to the `pxf` protocol, use the `GRANT` command. For example, to grant the role named `bill` read access to data referenced by an external table created with the `pxf` protocol:

```
GRANT SELECT ON PROTOCOL pxf TO bill;
```

To write data to an external data store with PXF, you create an external table with the `CREATE WRITABLE EXTERNAL TABLE` command that specifies the `pxf` protocol. You must specifically grant `INSERT` permission to the `pxf` protocol to all non-`SUPERUSER` Greenplum Database roles that require such access. For example:

```
GRANT INSERT ON PROTOCOL pxf TO bill;
```

Registering PXF JAR Dependencies

You use PXF to access data stored on external systems. Depending upon the external data store, this access may require that you install and/or configure additional components or services for the external data store.

PXF depends on JAR files and other configuration information provided by these additional components. The `$PXF_HOME/conf/pxf-private.classpath` file identifies PXF internal JAR dependencies. In most cases, PXF manages the `pxf-private.classpath` file, adding entries as necessary based on the connectors that you use.

Should you need to add an additional JAR dependency for PXF, for example a JDBC driver JAR file, you must log in to the Greenplum Database master host, copy the JAR file to the PXF user configuration runtime library directory (`$PXF_CONF/lib`), sync the PXF configuration to the Greenplum Database cluster, and then restart PXF on each segment host. For example:

```
$ ssh gpadmin@<gpmaster>
gpadmin@gpmaster$ cp new_dependent_jar.jar $PXF_CONF/lib/
gpadmin@gpmaster$ pxf cluster sync
gpadmin@gpmaster$ pxf cluster restart
```

Monitoring PXF

The `pxf cluster status` command displays the status of the PXF service instance on all segment hosts in your Greenplum Database cluster. `pxf status` displays the status of the PXF service instance on the local (segment) host.

Only the `gpadmin` user can request the status of the PXF service.

Perform the following procedure to request the PXF status of your Greenplum Database cluster.

1. Log in to the Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Run the `pxf cluster status` command:

   ```
   gpadmin@gpmaster$ pxf cluster status
   ```

Accessing Hadoop with PXF

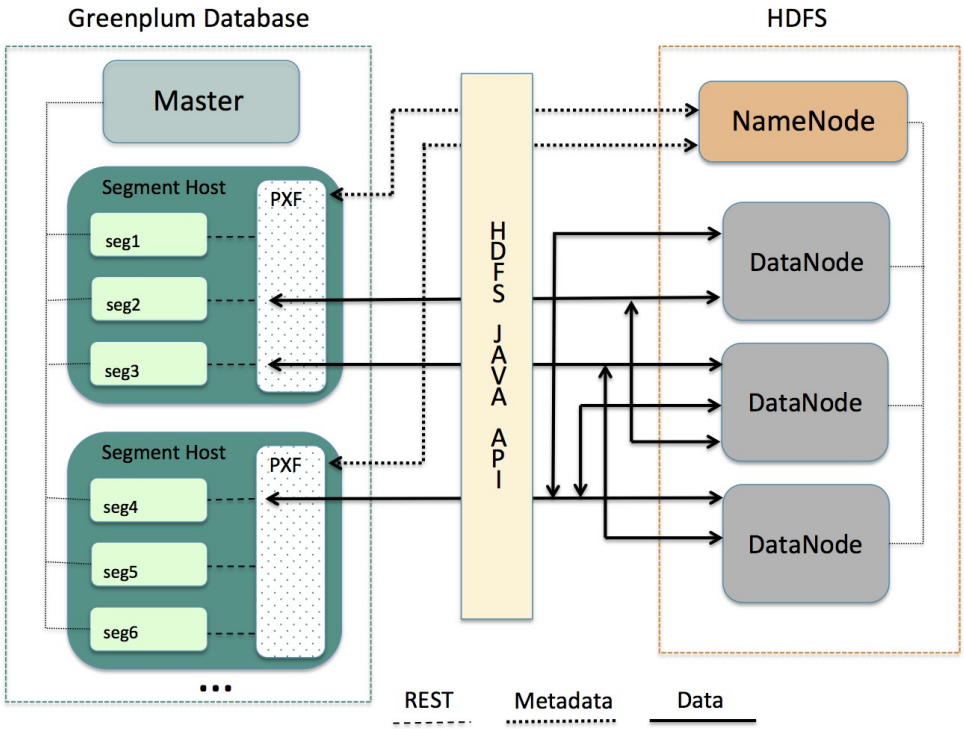
PXF is compatible with Cloudera, Hortonworks Data Platform, MapR, and generic Apache Hadoop distributions. PXF is installed with HDFS, Hive, and HBase connectors. You use these connectors to access varied formats of data from these Hadoop distributions.

Architecture

HDFS is the primary distributed storage mechanism used by Apache Hadoop. When a user or application performs a query on a PXF external table that references an HDFS file, the Greenplum Database master node dispatches the query to all segment hosts. Each segment instance contacts the PXF agent running on its host. When it receives the request from a segment instance, the PXF agent:

1. Allocates a worker thread to serve the request from a segment.
2. Invokes the HDFS Java API to request metadata information for the HDFS file from the HDFS NameNode.
3. Provides the metadata information returned by the HDFS NameNode to the segment instance.

Figure: PXF-to-Hadoop Architecture



A segment instance uses its Greenplum Database `gp_segment_id` and the file block information described in the metadata to assign itself a specific portion of the query data. The segment instance then sends a request to the PXF agent to read the assigned data. This data may reside on one or more HDFS DataNodes.

The PXF agent invokes the HDFS Java API to read the data and delivers it to the segment instance. The segment instance delivers its portion of the data to the Greenplum Database master node. This communication occurs across segment hosts and segment instances in parallel.

Prerequisites

Before working with Hadoop data using PXF, ensure that:

- You have configured and initialized PXF, and PXF is running on each Greenplum Database segment host. See Configuring PXF for additional information.
- You have configured the PXF Hadoop Connectors that you plan to use. Refer to Configuring PXF Hadoop Connectors for instructions. If you plan to access JSON-formatted data stored in a Cloudera Hadoop cluster, PXF requires a Cloudera version 5.8 or later Hadoop distribution.
- If user impersonation is enabled (the default), ensure that you have granted read (and write as appropriate) permission to the HDFS files and directories that will be accessed as external tables in Greenplum Database to each Greenplum Database user/role name that will access the HDFS files and directories. If user impersonation is not enabled, you must grant this permission to the `gpadmin` user.
- Time is synchronized between the Greenplum Database segment hosts and the external

Hadoop systems.

HDFS Shell Command Primer

Examples in the PXF Hadoop topics access files on HDFS. You can choose to access files that already exist in your HDFS cluster. Or, you can follow the steps in the examples to create new files.

A Hadoop installation includes command-line tools that interact directly with your HDFS file system. These tools support typical file system operations that include copying and listing files, changing file permissions, and so forth. You run these tools on a system with a Hadoop client installation. By default, Greenplum Database hosts do not include a Hadoop client installation.

The HDFS file system command syntax is `hdfs dfs <options> [<file>]`. Invoked with no options, `hdfs dfs` lists the file system options supported by the tool.

The user invoking the `hdfs dfs` command must have read privileges on the HDFS data store to list and view directory and file contents, and write permission to create directories and files.

The `hdfs dfs` options used in the PXF Hadoop topics are:

Option	Description
`-cat`	Display file contents.
`-mkdir`	Create a directory in HDFS.
`-put`	Copy a file from the local file system to HDFS.

Examples:

Create a directory in HDFS:

```
$ hdfs dfs -mkdir -p /data/exampledir
```

Copy a text file from your local file system to HDFS:

```
$ hdfs dfs -put /tmp/example.txt /data/exampledir/
```

Display the contents of a text file located in HDFS:

```
$ hdfs dfs -cat /data/exampledir/example.txt
```

Connectors, Data Formats, and Profiles

The PXF Hadoop connectors provide built-in profiles to support the following data formats:

- Text
- Avro
- JSON
- ORC
- Parquet
- RCFile

- SequenceFile
- AvroSequenceFile

The PXF Hadoop connectors expose the following profiles to read, and in many cases write, these supported data formats:

Data Source	Data Format	Profile Name(s)	Deprecated Profile Name
HDFS	delimited single line text	hdfs:text	HdfsTextSimple
HDFS	delimited text with quoted linefeeds	hdfs:text:multi	HdfsTextMulti
HDFS	Avro	hdfs:avro	Avro
HDFS	JSON	hdfs:json	Json
HDFS	Parquet	hdfs:parquet	Parquet
HDFS	AvroSequenceFile	hdfs:AvroSequenceFile	n/a
HDFS	SequenceFile	hdfs:SequenceFile	SequenceWritable
Hive	stored as TextFile	Hive, HiveText	n/a
Hive	stored as SequenceFile	Hive	n/a
Hive	stored as RCFile	Hive, HiveRC	n/a
Hive	stored as ORC	Hive, HiveORC, HiveVectorizedORC	n/a
Hive	stored as Parquet	Hive	n/a
HBase	Any	HBase	n/a

You provide the profile name when you specify the `pxf` protocol on a `CREATE EXTERNAL TABLE` command to create a Greenplum Database external table that references a Hadoop file, directory, or table. For example, the following command creates an external table that uses the default server and specifies the profile named `hdfs:text`:

```
CREATE EXTERNAL TABLE pxf_hdfs_text(location text, month text, num_orders int, total_s
ales float8)
   LOCATION ('pxf://data/pxf_examples/pxf_hdfs_simple.txt?PROFILE=hdfs:text')
FORMAT 'TEXT' (delimiter=E',');
```

Reading and Writing HDFS Text Data

The PXF HDFS Connector supports plain delimited and comma-separated value form text data. This section describes how to use PXF to access HDFS text data, including how to create, query, and insert data into an external table that references files in the HDFS data store.

Prerequisites

Ensure that you have met the PXF Hadoop Prerequisites before you attempt to read data from or write data to HDFS.

Reading Text Data

Use the `hdfs:text` profile when you read plain text delimited or .csv data where each row is a single record. The following syntax creates a Greenplum Database readable external table that references such a text file on HDFS:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-hdfs-file>?PROFILE=hdfs:text[&SERVER=<server_name>][&IGNORE_
MISSING_PATH=<boolean>]')
FORMAT '[TEXT|CSV]' (delimiter[=|<space>][E]'<delim_value>');
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-hdfs-file>	The absolute path to the directory or file in the HDFS data store.
PROFILE	The `PROFILE` keyword must specify `hdfs:text`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
IGNORE_MISSING_PATH=<boolean>	Specify the action to take when <path-to-hdfs-file> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.
FORMAT	Use `FORMAT 'TEXT'` when <path-to-hdfs-file> references plain text delimited data. Use `FORMAT 'CSV'` when <path-to-hdfs-file> references comma-separated value data.
delimiter	The delimiter character in the data. For `FORMAT 'CSV'`, the default <delim_value> is a comma `,`. Preface the <delim_value> with an `E` when the value is an escape sequence. Examples: `(delimiter=E'\t')`, `(delimiter ':')`.

Note: PXF does not support CSV files with a header row, nor does it support the `(HEADER)` formatter option in the `CREATE EXTERNAL TABLE` command.

Example: Reading Text Data on HDFS

Perform the following procedure to create a sample text file, copy the file to HDFS, and use the `hdfs:text` profile and the default PXF server to create two PXF external tables to query the data:

1. Create an HDFS directory for PXF example data files. For example:

   ```
   $ hdfs dfs -mkdir -p /data/pxf_examples
   ```

2. Create a delimited plain text data file named `pxf_hdfs_simple.txt`:

   ```
   $ echo 'Prague,Jan,101,4875.33
   Rome,Mar,87,1557.39
   Bangalore,May,317,8936.99
   Beijing,Jul,411,11600.67' > /tmp/pxf_hdfs_simple.txt
   ```

   Note the use of the comma `,` to separate the four data fields.

3. Add the data file to HDFS:

```
$ hdfs dfs -put /tmp/pxf_hdfs_simple.txt /data/pxf_examples/
```

4. Display the contents of the `pxf_hdfs_simple.txt` file stored in HDFS:

```
$ hdfs dfs -cat /data/pxf_examples/pxf_hdfs_simple.txt
```

5. Start the `psql` subsystem:

```
$ psql -d postgres
```

6. Use the PXF `hdfs:text` profile to create a Greenplum Database external table that references the `pxf_hdfs_simple.txt` file that you just created and added to HDFS:

```
postgres=# CREATE EXTERNAL TABLE pxf_hdfs_textsimple(location text, month text,
 num_orders int, total_sales float8)
            LOCATION ('pxf://data/pxf_examples/pxf_hdfs_simple.txt?PROFILE=hdfs
:text')
          FORMAT 'TEXT' (delimiter=E',');
```

7. Query the external table:

```
postgres=# SELECT * FROM pxf_hdfs_textsimple;
```

```
   location    | month | num_orders | total_sales
---------------+-------+------------+-------------
 Prague        | Jan   |        101 |     4875.33
 Rome          | Mar   |         87 |     1557.39
 Bangalore     | May   |        317 |     8936.99
 Beijing       | Jul   |        411 |    11600.67
(4 rows)
```

8. Create a second external table that references `pxf_hdfs_simple.txt`, this time specifying the `CSV FORMAT`:

```
postgres=# CREATE EXTERNAL TABLE pxf_hdfs_textsimple_csv(location text, month t
ext, num_orders int, total_sales float8)
            LOCATION ('pxf://data/pxf_examples/pxf_hdfs_simple.txt?PROFILE=hdfs
:text')
          FORMAT 'CSV';
postgres=# SELECT * FROM pxf_hdfs_textsimple_csv;
```

When you specify `FORMAT 'CSV'` for comma-separated value data, no `delimiter` formatter option is required because comma is the default delimiter value.

Reading Text Data with Quoted Linefeeds

Use the `hdfs:text:multi` profile to read plain text data with delimited single- or multi- line records that include embedded (quoted) linefeed characters. The following syntax creates a Greenplum Database readable external table that references such a text file on HDFS:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-hdfs-file>?PROFILE=hdfs:text:multi[&SERVER=<server_name>][&I
```

```
GNORE_MISSING_PATH=<boolean>]')
FORMAT '[TEXT|CSV]' (delimiter[=|<space>][E]'<delim_value>');
```

The specific keywords and values used in the CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-hdfs-file>	The absolute path to the directory or file in the HDFS data store.
PROFILE	The `PROFILE` keyword must specify `hdfs:text:multi`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
IGNORE_MISSING_PATH=<boolean>	Specify the action to take when <path-to-hdfs-file> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.
FORMAT	Use `FORMAT 'TEXT'` when <path-to-hdfs-file> references plain text delimited data. Use `FORMAT 'CSV'` when <path-to-hdfs-file> references comma-separated value data.
delimiter	The delimiter character in the data. For `FORMAT 'CSV'`, the default <delim_value> is a comma `,`. Preface the <delim_value> with an `E` when the value is an escape sequence. Examples: `(delimiter=E'\t')`, `(delimiter ':')`.

Example: Reading Multi-Line Text Data on HDFS

Perform the following steps to create a sample text file, copy the file to HDFS, and use the PXF `hdfs:text:multi` profile and the default PXF server to create a Greenplum Database readable external table to query the data:

1. Create a second delimited plain text file:

   ```
   $ vi /tmp/pxf_hdfs_multi.txt
   ```

2. Copy/paste the following data into `pxf_hdfs_multi.txt`:

   ```
   "4627 Star Rd.
   San Francisco, CA  94107":Sept:2017
   "113 Moon St.
   San Diego, CA  92093":Jan:2018
   "51 Belt Ct.
   Denver, CO  90123":Dec:2016
   "93114 Radial Rd.
   Chicago, IL  60605":Jul:2017
   "7301 Brookview Ave.
   Columbus, OH  43213":Dec:2018
   ```

   Notice the use of the colon `:` to separate the three fields. Also notice the quotes around the first (address) field. This field includes an embedded line feed separating the street address from the city and state.

3. Copy the text file to HDFS:

   ```
   $ hdfs dfs -put /tmp/pxf_hdfs_multi.txt /data/pxf_examples/
   ```

4. Use the `hdfs:text:multi` profile to create an external table that references the

`pxf_hdfs_multi.txt` HDFS file, making sure to identify the `:` (colon) as the field separator:

```
postgres=# CREATE EXTERNAL TABLE pxf_hdfs_textmulti(address text, month text, y
ear int)
            LOCATION ('pxf://data/pxf_examples/pxf_hdfs_multi.txt?PROFILE=hdfs:
text:multi')
          FORMAT 'CSV' (delimiter ':');
```

Notice the alternate syntax for specifying the `delimiter`.

5. Query the `pxf_hdfs_textmulti` table:

```
postgres=# SELECT * FROM pxf_hdfs_textmulti;
```

```
         address          | month | year
--------------------------+-------+------
 4627 Star Rd.            | Sept  | 2017
 San Francisco, CA  94107
 113 Moon St.             | Jan   | 2018
 San Diego, CA  92093
 51 Belt Ct.              | Dec   | 2016
 Denver, CO  90123
 93114 Radial Rd.         | Jul   | 2017
 Chicago, IL  60605
 7301 Brookview Ave.      | Dec   | 2018
 Columbus, OH  43213
(5 rows)
```

Writing Text Data to HDFS

The PXF HDFS connector “hdfs:text” profile supports writing single line plain text data to HDFS. When you create a writable external table with the PXF HDFS connector, you specify the name of a directory on HDFS. When you insert records into a writable external table, the block(s) of data that you insert are written to one or more files in the directory that you specified.

Note: External tables that you create with a writable profile can only be used for `INSERT` operations. If you want to query the data that you inserted, you must create a separate readable external table that references the HDFS directory.

Use the following syntax to create a Greenplum Database writable external table that references an HDFS directory:

```
CREATE WRITABLE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-hdfs-dir>
    ?PROFILE=hdfs:text[&SERVER=<server_name>][&<custom-option>=<value>[...]]')
FORMAT '[TEXT|CSV]' (delimiter[=|<space>][E]'<delim_value>');
[DISTRIBUTED BY (<column_name> [, ... ] ) | DISTRIBUTED RANDOMLY];
```

The specific keywords and values used in the CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-hdfs-dir>	The absolute path to the directory in the HDFS data store.

Keyword	Value
PROFILE	The `PROFILE` keyword must specify `hdfs:text`
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
<custom-option>	<custom-option>s are described below.
FORMAT	Use `FORMAT 'TEXT'` to write plain, delimited text to <path-to-hdfs-dir>. Use `FORMAT 'CSV'` to write comma-separated value text to <path-to-hdfs-dir>.
delimiter	The delimiter character in the data. For `FORMAT 'CSV'`, the default <delim_value> is a comma `,`. Preface the <delim_value> with an `E` when the value is an escape sequence. Examples: `(delimiter=E'\t')`, `(delimiter ':')`.
DISTRIBUTED BY	If you want to load data from an existing Greenplum Database table into the writable external table, consider specifying the same distribution policy or <column_name> on both tables. Doing so will avoid extra motion of data between segments on the load operation.

Writable external tables that you create using the `hdfs:text` profile can optionally use record or block compression. The PXF `hdfs:text` profile supports the following compression codecs:

- org.apache.hadoop.io.compress.DefaultCodec
- org.apache.hadoop.io.compress.GzipCodec
- org.apache.hadoop.io.compress.BZip2Codec

You specify the compression codec via custom options in the `CREATE EXTERNAL TABLE LOCATION` clause. The `hdfs:text` profile support the following custom write options:

Option	Value Description

COMPRESSION_CODEC	The compression codec Java class name. If this option is not provided, Greenplum Database performs no data compression. Supported compression codecs include:
	`org.apache.hadoop.io.compress.DefaultCodec`
	`org.apache.hadoop.io.compress.BZip2Codec`
	`org.apache.hadoop.io.compress.GzipCodec`

COMPRESSION_TYPE	The compression type to employ; supported values are `RECORD` (the default) or `BLOCK`.
THREAD-SAFE	Boolean value determining if a table query can run in multi-threaded mode. The default value is `TRUE`. Set this option to `FALSE` to handle all requests in a single thread for operations that are not thread-safe (for example, compression).

Example: Writing Text Data to HDFS

This example utilizes the data schema introduced in Example: Reading Text Data on HDFS.

Column Name	Data Type
location	text
month	text
number_of_orders	int
total_sales	float8

This example also optionally uses the Greenplum Database external table named

`pxf_hdfs_textsimple` that you created in that exercise.

Procedure

Perform the following procedure to create Greenplum Database writable external tables utilizing the same data schema as described above, one of which will employ compression. You will use the PXF `hdfs:text` profile and the default PXF server to write data to the underlying HDFS directory. You will also create a separate, readable external table to read the data that you wrote to the HDFS directory.

1. Create a Greenplum Database writable external table utilizing the data schema described above. Write to the HDFS directory `/data/pxf_examples/pxfwritable_hdfs_textsimple1`. Create the table specifying a comma `,` as the delimiter:

   ```
   postgres=# CREATE WRITABLE EXTERNAL TABLE pxf_hdfs_writabletbl_1(location text,
    month text, num_orders int, total_sales float8)
               LOCATION ('pxf://data/pxf_examples/pxfwritable_hdfs_textsimple1?PRO
   FILE=hdfs:text')
             FORMAT 'TEXT' (delimiter=',');
   ```

   You specify the `FORMAT` subclause `delimiter` value as the single ascii comma character `,`.

2. Write a few individual records to the `pxfwritable_hdfs_textsimple1` HDFS directory by invoking the SQL `INSERT` command on `pxf_hdfs_writabletbl_1`:

   ```
   postgres=# INSERT INTO pxf_hdfs_writabletbl_1 VALUES ( 'Frankfurt', 'Mar', 777,
    3956.98 );
   postgres=# INSERT INTO pxf_hdfs_writabletbl_1 VALUES ( 'Cleveland', 'Oct', 3812
   , 96645.37 );
   ```

3. (Optional) Insert the data from the `pxf_hdfs_textsimple` table that you created in Example: Reading Text Data on HDFS into `pxf_hdfs_writabletbl_1`:

   ```
   postgres=# INSERT INTO pxf_hdfs_writabletbl_1 SELECT * FROM pxf_hdfs_textsimple
   ;
   ```

4. In another terminal window, display the data that you just added to HDFS:

   ```
   $ hdfs dfs -cat /data/pxf_examples/pxfwritable_hdfs_textsimple1/*
   Frankfurt,Mar,777,3956.98
   Cleveland,Oct,3812,96645.37
   Prague,Jan,101,4875.33
   Rome,Mar,87,1557.39
   Bangalore,May,317,8936.99
   Beijing,Jul,411,11600.67
   ```

   Because you specified comma `,` as the delimiter when you created the writable external table, this character is the field separator used in each record of the HDFS data.

5. Greenplum Database does not support directly querying a writable external table. To query the data that you just added to HDFS, you must create a readable external Greenplum Database table that references the HDFS directory:

   ```
   postgres=# CREATE EXTERNAL TABLE pxf_hdfs_textsimple_r1(location text, month te
   xt, num_orders int, total_sales float8)
               LOCATION ('pxf://data/pxf_examples/pxfwritable_hdfs_textsimple1?PRO
   ```

```
FILE=hdfs:text')
            FORMAT 'CSV';
```

You specify the `'CSV' FORMAT` when you create the readable external table because you created the writable table with a comma `,` as the delimiter character, the default delimiter for `'CSV' FORMAT`.

6. Query the readable external table:

```
postgres=# SELECT * FROM pxf_hdfs_textsimple_r1 ORDER BY total_sales;
```

```
 location  | month | num_orders | total_sales
-----------+-------+------------+-------------
 Rome      | Mar   |         87 |     1557.39
 Frankfurt | Mar   |        777 |     3956.98
 Prague    | Jan   |        101 |     4875.33
 Bangalore | May   |        317 |     8936.99
 Beijing   | Jul   |        411 |    11600.67
 Cleveland | Oct   |       3812 |    96645.37
(6 rows)
```

The `pxf_hdfs_textsimple_r1` table includes the records you individually inserted, as well as the full contents of the `pxf_hdfs_textsimple` table if you performed the optional step.

7. Create a second Greenplum Database writable external table, this time using Gzip compression and employing a colon `:` as the delimiter:

```
postgres=# CREATE WRITABLE EXTERNAL TABLE pxf_hdfs_writabletbl_2 (location text
, month text, num_orders int, total_sales float8)
            LOCATION ('pxf://data/pxf_examples/pxfwritable_hdfs_textsimple2?PRO
FILE=hdfs:text&COMPRESSION_CODEC=org.apache.hadoop.io.compress.GzipCodec')
          FORMAT 'TEXT' (delimiter=':');
```

8. Write a few records to the `pxfwritable_hdfs_textsimple2` HDFS directory by inserting directly into the `pxf_hdfs_writabletbl_2` table:

```
gpadmin=# INSERT INTO pxf_hdfs_writabletbl_2 VALUES ( 'Frankfurt', 'Mar', 777,
3956.98 );
gpadmin=# INSERT INTO pxf_hdfs_writabletbl_2 VALUES ( 'Cleveland', 'Oct', 3812,
 96645.37 );
```

9. In another terminal window, display the contents of the data that you added to HDFS; use the `-text` option to `hdfs dfs` to view the compressed data as text:

```
$ hdfs dfs -text /data/pxf_examples/pxfwritable_hdfs_textsimple2/*
Frankfurt:Mar:777:3956.98
Cleveland:Oct:3812:96645.3
```

Notice that the colon `:` is the field separator in this HDFS data.

To query data from the newly-created HDFS directory named `pxfwritable_hdfs_textsimple2`, you can create a readable external Greenplum Database table as described above that references this HDFS directory and specifies `FORMAT 'CSV' (delimiter=':')`.

Reading and Writing HDFS Avro Data

Use the PXF HDFS Connector to read and write Avro-format data. This section describes how to use PXF to read and write Avro data in HDFS, including how to create, query, and insert into an external table that references an Avro file in the HDFS data store.

Note: PXF does not support reading or writing compressed Avro files.

Prerequisites

Ensure that you have met the PXF Hadoop Prerequisites before you attempt to read data from HDFS.

Working with Avro Data

Apache Avro is a data serialization framework where the data is serialized in a compact binary format. Avro specifies that data types be defined in JSON. Avro format data has an independent schema, also defined in JSON. An Avro schema, together with its data, is fully self-describing.

Data Type Mapping

Avro supports both primitive and complex data types.

To represent Avro primitive data types in Greenplum Database, map data values to Greenplum Database columns of the same type.

Avro supports complex data types including arrays, maps, records, enumerations, and fixed types. Map top-level fields of these complex data types to the Greenplum Database `TEXT` type. While Greenplum Database does not natively support reading these types, you can create Greenplum Database functions or application code to extract or further process subcomponents of these complex data types.

Read Mapping

PXF uses the following data type mapping when reading Avro data:

Avro Data Type	PXF/Greenplum Data Type
boolean	boolean
bytes	bytea
double	double
float	real
int	int
long	bigint
string	text
Complex type: Array, Map, Record, or Enum	text, with delimiters inserted between collection items, mapped key-value pairs, and record data.
Complex type: Fixed	bytea (supported for read operations only).

Avro Data Type	PXF/Greenplum Data Type
Union	Follows the above conventions for primitive or complex data types, depending on the union; must contain 2 elements, one of which must be null.

Write Mapping

PXF supports writing only Avro primitive data types. It does not support writing complex types to Avro.

PXF uses the following data type mapping when writing Avro data:

PXF/Greenplum Data Type	Avro Data Type
bigint	long
boolean	boolean
bytea	bytes
double	double
enum	string
int	int
real	float
smallint[1]	int
text	string
array ([]), enum, record	string

[1] PXF converts Greenplum `smallint` types to `int` before it writes the Avro data. Be sure to read the field into an `int`.

Avro Schemas and Data

Avro schemas are defined using JSON, and composed of the same primitive and complex types identified in the data type mapping section above. Avro schema files typically have a `.avsc` suffix.

Fields in an Avro schema file are defined via an array of objects, each of which is specified by a name and a type.

An Avro data file contains the schema and a compact binary representation of the data. Avro data files typically have the `.avro` suffix.

You can specify an Avro schema on both read and write operations to HDFS. You can provide either a binary `*.avro` file or a JSON-format `*.avsc` file for the schema file:

External Table Type	Schema Specified?	Description
readable	yes	PXF uses the specified schema; this overrides the schema embedded in the Avro data file.
readable	no	PXF uses the schema embedded in the Avro data file.

External Table Type	Schema Specified?	Description
writable	yes	PXF uses the specified schema.
writable	no	PXF creates the Avro schema based on the external table definition.

When you provide the Avro schema file to PXF, the file must reside in the same location on each Greenplum Database segment host **or** the file may reside on the Hadoop file system. PXF first searches for an absolute file path on the Greenplum segment hosts. If PXF does not find the schema file there, it searches for the file relative to the PXF classpath. If PXF cannot find the schema file locally, it searches for the file on HDFS.

The `$PXF_CONF/conf` directory is in the PXF classpath. PXF can locate an Avro schema file that you add to this directory on every Greenplum Database segment host.

See Writing Avro Data for additional schema considerations when writing Avro data to HDFS.

Creating the External Table

Use the `hdfs:avro` profile to read or write Avro-format data in HDFS. The following syntax creates a Greenplum Database readable external table that references such a file:

```
CREATE [WRITABLE] EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-hdfs-file>?PROFILE=hdfs:avro[&SERVER=<server_name>][&<custom
-option>=<value>[...]]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import'|'pxfwritable_export');
[DISTRIBUTED BY (<column_name> [, ... ] ) | DISTRIBUTED RANDOMLY];
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-hdfs-file>	The absolute path to the directory or file in the HDFS data store.
PROFILE	The `PROFILE` keyword must specify `hdfs:avro`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
<custom-option>	<custom-option>s are discussed below.
FORMAT 'CUSTOM'	Use `FORMAT` '`CUSTOM`' with `(FORMATTER='pxfwritable_export')` (write) or `(FORMATTER='pxfwritable_import')` (read).
DISTRIBUTED BY	If you want to load data from an existing Greenplum Database table into the writable external table, consider specifying the same distribution policy or <column_name> on both tables. Doing so will avoid extra motion of data between segments on the load operation.

For complex types, the PXF `hdfs:avro` profile inserts default delimiters between collection items and values before display. You can use non-default delimiter characters by identifying values for specific `hdfs:avro` custom options in the `CREATE EXTERNAL TABLE` command.

The `hdfs:avro` profile supports the following <custom-option>s:

Option Keyword	Description
COLLECTION_DELIM	The delimiter character(s) placed between entries in a top-level array, map, or record field when PXF maps an Avro complex data type to a text column. The default is the comma `,` character. (Read)
MAPKEY_DELIM	The delimiter character(s) placed between the key and value of a map entry when PXF maps an Avro complex data type to a text column. The default is the colon `:` character. (Read)
RECORDKEY_DELIM	The delimiter character(s) placed between the field name and value of a record entry when PXF maps an Avro complex data type to a text column. The default is the colon `:` character. (Read)
SCHEMA	The absolute path to the Avro schema file on the segment host or on HDFS, or the relative path to the schema file on the segment host. (Read and Write)
IGNORE_MISSING_PATH	A Boolean value that specifies the action to take when <path-to-hdfs-file> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment. (Read)

The PXF `hdfs:avro` profile supports encoding- and compression-related write options. You specify these write options in the `CREATE WRITABLE EXTERNAL TABLE LOCATION` clause. The `hdfs:avro` profile supports the following custom write options:

Write Option	Value Description
COMPRESSION_CODEC	The compression codec alias. Supported compression codecs for writing Avro data include: `snappy`, `deflate`, and `uncompressed` . If this option is not provided, PXF compresses the data using `deflate` compression.
CODEC_LEVEL	The compression level (applicable to the `deflate` codec only). This level controls the trade-off between speed and compression. Valid values are 1 (fastest) to 9 (most compressed). The default compression level when using the `deflate` codec is 6.

Example: Reading Avro Data

The examples in this section will operate on Avro data with the following field name and data type record schema:

- id - long
- username - string
- followers - array of string
- fmap - map of long
- relationship - enumerated type
- address - record comprised of street number (int), street name (string), and city (string)

You create an Avro schema and data file, and then create a readable external table to read the data.

Create Schema

Perform the following operations to create an Avro schema to represent the example schema described above.

1. Create a file named `avro_schema.avsc`:

```
$ vi /tmp/avro_schema.avsc
```

2. Copy and paste the following text into `avro_schema.avsc`:

```
{
"type" : "record",
  "name" : "example_schema",
  "namespace" : "com.example",
  "fields" : [ {
    "name" : "id",
    "type" : "long",
    "doc" : "Id of the user account"
  }, {
    "name" : "username",
    "type" : "string",
    "doc" : "Name of the user account"
  }, {
    "name" : "followers",
    "type" : {"type": "array", "items": "string"},
    "doc" : "Users followers"
  }, {
    "name": "fmap",
    "type": {"type": "map", "values": "long"}
  }, {
    "name": "relationship",
    "type": {
        "type": "enum",
        "name": "relationshipEnum",
        "symbols": ["MARRIED","LOVE","FRIEND","COLLEAGUE","STRANGER","ENEMY"]
    }
  }, {
    "name": "address",
    "type": {
        "type": "record",
        "name": "addressRecord",
        "fields": [
            {"name":"number", "type":"int"},
            {"name":"street", "type":"string"},
            {"name":"city", "type":"string"}]
    }
  } ],
  "doc:" : "A basic schema for storing messages"
}
```

Create Avro Data File (JSON)

Perform the following steps to create a sample Avro data file conforming to the above schema.

1. Create a text file named `pxf_avro.txt`:

```
$ vi /tmp/pxf_avro.txt
```

2. Enter the following data into `pxf_avro.txt`:

```
{"id":1, "username":"john","followers":["kate", "santosh"], "relationship": "FR
IEND", "fmap": {"kate":10,"santosh":4}, "address":{"number":1, "street":"renais
sance drive", "city":"san jose"}}
```

```
{"id":2, "username":"jim","followers":["john", "pam"], "relationship": "COLLEAG
UE", "fmap": {"john":3,"pam":3}, "address":{"number":9, "street":"deer creek",
"city":"palo alto"}}
```

The sample data uses a comma `,` to separate top level records and a colon `:` to separate map/key values and record field name/values.

3. Convert the text file to Avro format. There are various ways to perform the conversion, both programmatically and via the command line. In this example, we use the Java Avro tools; the jar `avro-tools-1.9.1.jar` file resides in the current directory:

```
$ java -jar ./avro-tools-1.9.1.jar fromjson --schema-file /tmp/avro_schema.avsc
 /tmp/pxf_avro.txt > /tmp/pxf_avro.avro
```

The generated Avro binary data file is written to `/tmp/pxf_avro.avro`.

4. Copy the generated Avro file to HDFS:

```
$ hdfs dfs -put /tmp/pxf_avro.avro /data/pxf_examples/
```

Reading Avro Data

Perform the following operations to create and query an external table that references the `pxf_avro.avro` file that you added to HDFS in the previous section. When creating the table:

- Use the PXF default server.
- Map the top-level primitive fields, `id` (type long) and `username` (type string), to their equivalent Greenplum Database types (bigint and text).
- Map the remaining complex fields to type text.
- Explicitly set the record, map, and collection delimiters using the `hdfs:avro` profile custom options.

1. Use the `hdfs:avro` profile to create a queryable external table from the `pxf_avro.avro` file:

```
postgres=# CREATE EXTERNAL TABLE pxf_hdfs_avro(id bigint, username text, follow
ers text, fmap text, relationship text, address text)
            LOCATION ('pxf://data/pxf_examples/pxf_avro.avro?PROFILE=hdfs:avro&
COLLECTION_DELIM=,&MAPKEY_DELIM=:&RECORDKEY_DELIM=:')
          FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

2. Perform a simple query of the `pxf_hdfs_avro` table:

```
postgres=# SELECT * FROM pxf_hdfs_avro;
```

```
 id | username |   followers    |        fmap         | relationship |
             address
----+----------+----------------+---------------------+--------------+----------
-----------------------------------------
  1 | john     | [kate,santosh] | {kate:10,santosh:4} | FRIEND       | {number:
1,street:renaissance drive,city:san jose}
  2 | jim      | [john,pam]     | {pam:3,john:3}      | COLLEAGUE    | {number:
9,street:deer creek,city:palo alto}
(2 rows)
```

The simple query of the external table shows the components of the complex type data separated with the delimiters specified in the `CREATE EXTERNAL TABLE` call.

3. Process the delimited components in the text columns as necessary for your application. For example, the following command uses the Greenplum Database internal `string_to_array` function to convert entries in the `followers` field to a text array column in a new view.

```
postgres=# CREATE VIEW followers_view AS
SELECT username, address, string_to_array(substring(followers FROM 2 FOR (char_
length(followers) - 2)), ',')::text[]
    AS followers
  FROM pxf_hdfs_avro;
```

4. Query the view to filter rows based on whether a particular follower appears in the view:

```
postgres=# SELECT username, address FROM followers_view WHERE followers @> '{jo
hn}';
```

```
 username |                   address
----------+-----------------------------------------------
 jim      | {number:9,street:deer creek,city:palo alto}
```

Writing Avro Data

The PXF HDFS connector `hdfs:avro` profile supports writing Avro data to HDFS. When you create a writable external table to write Avro data, you specify the name of a directory on HDFS. When you insert records into the writable external table, the block(s) of data that you insert are written to one or more files in the directory that you specify.

When you create a writable external table to write data to an Avro file, each table row is an Avro record and each table column is an Avro field.

If you do not specify a `SCHEMA` file, PXF generates a schema for the Avro file based on the Greenplum Database external table definition. PXF assigns the name of the external table column to the Avro field name. Because Avro has a `null` type and Greenplum external tables do not support the `NOT NULL` column qualifier, PXF wraps each data type in an Avro `union` of the mapped type and `null`. For example, for a writable external table column that you define with the Greenplum Database `text` data type, PXF generates the following schema element:

```
["string", "null"]
```

PXF returns an error if you provide a schema that does not include a `union` of the field data type with `null`, and PXF encounters a NULL data field.

PXF supports writing only Avro primitive data types. It does not support writing complex types to Avro:

- When you specify a `SCHEMA` file in the `LOCATION`, the schema must include only primitive data types.
- When PXF generates the schema, it writes any complex type that you specify in the writable external table column definition to the Avro file as a single Avro `string` type. For example, if

you write an array of integers, PXF converts the array to a `string`, and you must read this data with a Greenplum `text`-type column.

Example: Writing Avro Data

In this example, you create an external table that writes to an Avro file on HDFS, letting PXF generate the Avro schema. After you insert some data into the file, you create a readable external table to query the Avro data.

The Avro file that you create and read in this example includes the following fields:

- id: `int`
- username: `text`
- followers: `text[]`

Example procedure:

1. Create the writable external table:

```
postgres=# CREATE WRITABLE EXTERNAL TABLE pxf_avrowrite(id int, username text,
followers text[])
            LOCATION ('pxf://data/pxf_examples/pxfwrite.avro?PROFILE=hdfs:avro'
)
          FORMAT 'CUSTOM' (FORMATTER='pxfwritable_export');
```

1. Insert some data into the `pxf_avrowrite` table:

```
postgres=# INSERT INTO pxf_avrowrite VALUES (33, 'oliver', ARRAY['alex','frank'
]);
postgres=# INSERT INTO pxf_avrowrite VALUES (77, 'lisa', ARRAY['tom','mary']);
```

   PXF uses the external table definition to generate the Avro schema.

2. Create an external table to read the Avro data that you just inserted into the table:

```
postgres=# CREATE EXTERNAL TABLE read_pxfwrite(id int, username text, followers
 text)
            LOCATION ('pxf://data/pxf_examples/pxfwrite.avro?PROFILE=hdfs:avro'
)
          FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

   Notice that the `followers` column is of type `text`.

3. Read the Avro data by querying the `read_pxfwrite` table:

```
postgres=# SELECT * FROM read_pxfwrite;
```

```
 id | username |  followers
----+----------+--------------
 77 | lisa     | {tom,mary}
 33 | oliver   | {alex,frank}
(2 rows)
```

   `followers` is a single string comprised of the `text` array elements that you inserted into the table.

Reading JSON Data from HDFS

Use the PXF HDFS Connector to read JSON-format data. This section describes how to use PXF to access JSON data in HDFS, including how to create and query an external table that references a JSON file in the HDFS data store.

Prerequisites

Ensure that you have met the PXF Hadoop Prerequisites before you attempt to read data from HDFS.

Working with JSON Data

JSON is a text-based data-interchange format. JSON data is typically stored in a file with a `.json` suffix.

A `.json` file will contain a collection of objects. A JSON object is a collection of unordered name/value pairs. A value can be a string, a number, true, false, null, or an object or an array. You can define nested JSON objects and arrays.

Sample JSON data file content:

```
{
  "created_at":"MonSep3004:04:53+00002013",
  "id_str":"384529256681725952",
  "user": {
    "id":31424214,
    "location":"COLUMBUS"
  },
  "coordinates":{
    "type":"Point",
    "values":[
       13,
       99
    ]
  }
}
```

In the sample above, `user` is an object composed of fields named `id` and `location`. To specify the nested fields in the `user` object as Greenplum Database external table columns, use `.` projection:

```
user.id
user.location
```

`coordinates` is an object composed of a text field named `type` and an array of integers named `values`. Use `[]` to identify specific elements of the `values` array as Greenplum Database external table columns:

```
coordinates.values[0]
coordinates.values[1]
```

Refer to Introducing JSON for detailed information on JSON syntax.

JSON to Greenplum Database Data Type Mapping

To represent JSON data in Greenplum Database, map data values that use a primitive data type to Greenplum Database columns of the same type. JSON supports complex data types including projections and arrays. Use N-level projection to map members of nested objects and arrays to primitive data types.

The following table summarizes external mapping rules for JSON data.

Table 1. JSON Mapping

JSON Data Type	PXF/Greenplum Data Type
Primitive type (integer, float, string, boolean, null)	Use the corresponding Greenplum Database built-in data type; see Greenplum Database Data Types.
Array	Use `[]` brackets to identify a specific array index to a member of primitive type.
Object	Use dot `.` notation to specify each level of projection (nesting) to a member of a primitive type.

JSON Data Read Modes

PXF supports two data read modes. The default mode expects one full JSON record per line. PXF also supports a read mode operating on JSON records that span multiple lines.

In upcoming examples, you will use both read modes to operate on a sample data set. The schema of the sample data set defines objects with the following member names and value data types:

- “created_at” - text
- “id_str” - text
- “user” - object
    - “id” - integer
    - “location” - text
- “coordinates” - object (optional)
    - “type” - text
    - “values” - array
        - [0] - integer
        - [1] - integer

The single-JSON-record-per-line data set follows:

```
{"created_at":"FriJun0722:45:03+00002013","id_str":"343136551322136576","user":{
"id":395504494,"location":"NearCornwall"},"coordinates":{"type":"Point","values"
: [ 6, 50 ]}},
{"created_at":"FriJun0722:45:02+00002013","id_str":"343136547115253761","user":{
"id":26643566,"location":"Austin,Texas"}, "coordinates": null},
{"created_at":"FriJun0722:45:02+00002013","id_str":"343136547136233472","user":{
"id":287819058,"location":""}, "coordinates": null}
```

This is the data set for the multi-line JSON record data set:

```
{
  "root":[
    {
      "record_obj":{
        "created_at":"MonSep3004:04:53+00002013",
        "id_str":"384529256681725952",
        "user":{
          "id":31424214,
          "location":"COLUMBUS"
        },
        "coordinates":null
      },
      "record_obj":{
        "created_at":"MonSep3004:04:54+00002013",
        "id_str":"384529260872228864",
        "user":{
          "id":67600981,
          "location":"KryberWorld"
        },
        "coordinates":{
          "type":"Point",
          "values":[
             8,
             52
          ]
        }
      }
    }
  ]
}
```

You will create JSON files for the sample data sets and add them to HDFS in the next section.

Loading the Sample JSON Data to HDFS

The PXF HDFS connector reads native JSON stored in HDFS. Before you can use Greenplum Database to query JSON format data, the data must reside in your HDFS data store.

Copy and paste the single line JSON record sample data set above to a file named `singleline.json`. Similarly, copy and paste the multi-line JSON record data set to a file named `multiline.json`.

Note: Ensure that there are **no** blank lines in your JSON files.

Copy the JSON data files that you just created to your HDFS data store. Create the `/data/pxf_examples` directory if you did not do so in a previous exercise. For example:

```
$ hdfs dfs -mkdir /data/pxf_examples
$ hdfs dfs -put singleline.json /data/pxf_examples
$ hdfs dfs -put multiline.json /data/pxf_examples
```

Once the data is loaded to HDFS, you can use Greenplum Database and PXF to query and analyze the JSON data.

Creating the External Table

Use the `hdfs:json` profile to read JSON-format files from HDFS. The following syntax creates a Greenplum Database readable external table that references such a file:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-hdfs-file>?PROFILE=hdfs:json[&SERVER=<server_name>][&<custom
-option>=<value>[...]]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-hdfs-file>	The absolute path to the directory or file in the HDFS data store.
PROFILE	The `PROFILE` keyword must specify `hdfs:json`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
<custom-option>	<custom-option>s are discussed below.
FORMAT 'CUSTOM'	Use `FORMAT 'CUSTOM'` with the `hdfs:json` profile. The `CUSTOM FORMAT` requires that you specify `(FORMATTER='pxfwritable_import')`.

PXF supports single- and multi- line JSON records. When you want to read multi-line JSON records, you must provide an `IDENTIFIER` <custom-option> and value. Use this <custom-option> to identify the member name of the first field in the JSON record object.

The `hdfs:json` profile supports the following <custom-option>s:

Option Keyword	Syntax, Example(s)	Description
IDENTIFIER	`&IDENTIFIER=<value>` `&IDENTIFIER=created_at`	You must include the `IDENTIFIER` keyword and <value> in the `LOCATION` string only when you are accessing JSON data comprised of multi-line records. Use the <value> to identify the member name of the first field in the JSON record object.
IGNORE_MISSING_PATH	`&IGNORE_MISSING_PATH=<boolean>`	Specify the action to take when <path-to-hdfs-file> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.

Example: Reading a JSON File with Single Line Records

Use the following CREATE EXTERNAL TABLE SQL command to create a readable external table that references the single-line-per-record JSON data file and uses the PXF default server.

```
CREATE EXTERNAL TABLE singleline_json_tbl(
  created_at TEXT,
  id_str TEXT,
  "user.id" INTEGER,
  "user.location" TEXT,
  "coordinates.values[0]" INTEGER,
  "coordinates.values[1]" INTEGER
)
LOCATION('pxf://data/pxf_examples/singleline.json?PROFILE=hdfs:json')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

Notice the use of `.` projection to access the nested fields in the `user` and `coordinates` objects. Also notice the use of `[]` to access specific elements of the `coordinates.values[]` array.

To query the JSON data in the external table:

```
SELECT * FROM singleline_json_tbl;
```

Example: Reading a JSON file with Multi-Line Records

The SQL command to create a readable external table from the multi-line-per-record JSON file is very similar to that of the single line data set above. You must additionally specify the `LOCATION` clause `IDENTIFIER` keyword and an associated value when you want to read multi-line JSON records. For example:

```
CREATE EXTERNAL TABLE multiline_json_tbl(
  created_at TEXT,
  id_str TEXT,
  "user.id" INTEGER,
  "user.location" TEXT,
  "coordinates.values[0]" INTEGER,
  "coordinates.values[1]" INTEGER
)
LOCATION('pxf://data/pxf_examples/multiline.json?PROFILE=hdfs:json&IDENTIFIER=created_
at')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

`created_at` identifies the member name of the first field in the JSON record `record_obj` in the sample data schema.

To query the JSON data in this external table:

```
SELECT * FROM multiline_json_tbl;
```

Reading and Writing HDFS Parquet Data

Use the PXF HDFS connector to read and write Parquet-format data. This section describes how to read and write HDFS files that are stored in Parquet format, including how to create, query, and insert into external tables that reference files in the HDFS data store.

PXF currently supports reading and writing primitive Parquet data types only.

Prerequisites

Ensure that you have met the PXF Hadoop Prerequisites before you attempt to read data from or write data to HDFS.

Data Type Mapping

To read and write Parquet primitive data types in Greenplum Database, map Parquet data values to Greenplum Database columns of the same type.

Parquet supports a small set of primitive data types, and uses metadata annotations to extend the

data types that it supports. These annotations specify how to interpret the primitive type. For example, Parquet stores both `INTEGER` and `DATE` types as the `INT32` primitive type. An annotation identifies the original type as a `DATE`.

Read Mapping

PXF uses the following data type mapping when reading Parquet data:

Parquet Data Type	Original Type	PXF/Greenplum Data Type
binary (byte_array)	Date	Date
binary (byte_array)	Timestamp_millis	Timestamp
binary (byte_array)	all others	Text
binary (byte_array)	–	Bytea
boolean	–	Boolean
double	–	Float8
fixed_len_byte_array	–	Numeric
float	–	Real
int32	Date	Date
int32	Decimal	Numeric
int32	int_8	Smallint
int32	int_16	Smallint
int32	–	Integer
int64	Decimal	Numeric
int64	–	Bigint
int96	–	Timestamp

Note: PXF supports filter predicate pushdown on all parquet data types listed above, *except* the `fixed_len_byte_array` and `int96` types.

Write Mapping

PXF uses the following data type mapping when writing Parquet data:

PXF/Greenplum Data Type	Original Type	Parquet Data Type
Boolean	–	boolean
Bytea	–	binary
Bigint	–	int64
SmallInt	int_16	int32
Integer	–	int32
Real	–	float

PXF/Greenplum Data Type	Original Type	Parquet Data Type
Float8	–	double
Numeric/Decimal	Decimal	fixed_len_byte_array
Timestamp[1]	–	int96
Timestamptz[2]	–	int96
Date	utf8	binary
Time	utf8	binary
Varchar	utf8	binary
Text	utf8	binary
OTHERS	–	UNSUPPORTED

[1] PXF localizes a `Timestamp` to the current system timezone and converts it to universal time (UTC) before finally converting to `int96`.

[2] PXF converts a `Timestamptz` to a UTC `timestamp` and then converts to `int96`. PXF loses the time zone information during this conversion.

Creating the External Table

The PXF HDFS connector `hdfs:parquet` profile supports reading and writing HDFS data in Parquet-format. When you insert records into a writable external table, the block(s) of data that you insert are written to one or more files in the directory that you specified.

Use the following syntax to create a Greenplum Database external table that references an HDFS directory:

```
CREATE [WRITABLE] EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-hdfs-dir>
    ?PROFILE=hdfs:parquet[&SERVER=<server_name>][&<custom-option>=<value>[...]]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import'|'pxfwritable_export');
[DISTRIBUTED BY (<column_name> [, ... ] ) | DISTRIBUTED RANDOMLY];
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-hdfs-file>	The absolute path to the directory in the HDFS data store.
PROFILE	The `PROFILE` keyword must specify `hdfs:parquet`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
<custom-option>	<custom-option>s are described below.

Keyword	Value
FORMAT 'CUSTOM'	Use `FORMAT` '`CUSTOM`' with `(FORMATTER='pxfwritable_export')` (write) or `(FORMATTER='pxfwritable_import')` (read).
DISTRIBUTED BY	If you want to load data from an existing Greenplum Database table into the writable external table, consider specifying the same distribution policy or <column_name> on both tables. Doing so will avoid extra motion of data between segments on the load operation.

The PXF `hdfs:parquet` profile supports the following read option. You specify this option in the `CREATE EXTERNAL TABLE LOCATION` clause:

Read Option	Value Description
IGNORE_MISSING_PATH	A Boolean value that specifies the action to take when <path-to-hdfs-file> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.

The PXF `hdfs:parquet` profile supports encoding- and compression-related write options. You specify these write options in the `CREATE WRITABLE EXTERNAL TABLE LOCATION` clause. The `hdfs:parquet` profile supports the following custom write options:

Write Option	Value Description
COMPRESSION_CODEC	The compression codec alias. Supported compression codecs for writing Parquet data include: `snappy`, `gzip`, `lzo`, and `uncompressed` . If this option is not provided, PXF compresses the data using `snappy` compression.
ROWGROUP_SIZE	A Parquet file consists of one or more row groups, a logical partitioning of the data into rows. `ROWGROUP_SIZE` identifies the size (in bytes) of the row group. The default row group size is `8 * 1024 * 1024` bytes.
PAGE_SIZE	A row group consists of column chunks that are divided up into pages. `PAGE_SIZE` is the size (in bytes) of such a page. The default page size is `1024 * 1024` bytes.
DICTIONARY_PAGE_SIZE	Dictionary encoding is enabled by default when PXF writes Parquet files. There is a single dictionary page per column, per row group. `DICTIONARY_PAGE_SIZE` is similar to `PAGE_SIZE`, but for the dictionary. The default dictionary page size is `512 * 1024` bytes.
PARQUET_VERSION	The Parquet version; values `v1` and `v2` are supported. The default Parquet version is `v1`.
SCHEMA	The location of the Parquet schema file on the file system of the specified `SERVER`.

Note: You must explicitly specify `uncompressed` if you do not want PXF to compress the data.

Parquet files that you write to HDFS with PXF have the following naming format: `<file>.<compress_extension>.parquet`, for example `1547061635-0000004417_0.gz.parquet`.

Example

This example utilizes the data schema introduced in Example: Reading Text Data on HDFS.

Column Name	Data Type
location	text
month	text
number_of_orders	int

Column Name	Data Type
total_sales	float8

In this example, you create a Parquet-format writable external table that uses the default PXF server to reference Parquet-format data in HDFS, insert some data into the table, and then create a readable external table to read the data.

1. Use the `hdfs:parquet` profile to create a writable external table. For example:

```
postgres=# CREATE WRITABLE EXTERNAL TABLE pxf_tbl_parquet (location text, month
 text, number_of_orders int, total_sales double precision)
    LOCATION ('pxf://data/pxf_examples/pxf_parquet?PROFILE=hdfs:parquet')
  FORMAT 'CUSTOM' (FORMATTER='pxfwritable_export');
```

2. Write a few records to the `pxf_parquet` HDFS directory by inserting directly into the `pxf_tbl_parquet` table. For example:

```
postgres=# INSERT INTO pxf_tbl_parquet VALUES ( 'Frankfurt', 'Mar', 777, 3956.9
8 );
postgres=# INSERT INTO pxf_tbl_parquet VALUES ( 'Cleveland', 'Oct', 3812, 96645
.37 );
```

3. Recall that Greenplum Database does not support directly querying a writable external table. To read the data in `pxf_parquet`, create a readable external Greenplum Database referencing this HDFS directory:

```
postgres=# CREATE EXTERNAL TABLE read_pxf_parquet(location text, month text, nu
mber_of_orders int, total_sales double precision)
    LOCATION ('pxf://data/pxf_examples/pxf_parquet?PROFILE=hdfs:parquet')
    FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

4. Query the readable external table `read_pxf_parquet`:

```
postgres=# SELECT * FROM read_pxf_parquet ORDER BY total_sales;
```

```
 location  | month | number_of_orders | total_sales
-----------+-------+------------------+-------------
 Frankfurt | Mar   |              777 |     3956.98
 Cleveland | Oct   |             3812 |     96645.4
(2 rows)
```

Reading and Writing HDFS SequenceFile Data

The PXF HDFS connector supports SequenceFile format binary data. This section describes how to use PXF to read and write HDFS SequenceFile data, including how to create, insert, and query data in external tables that reference files in the HDFS data store.

Prerequisites

Ensure that you have met the PXF Hadoop Prerequisites before you attempt to read data from or write data to HDFS.

Creating the External Table

The PXF HDFS connector `hdfs:SequenceFile` profile supports reading and writing HDFS data in SequenceFile binary format. When you insert records into a writable external table, the block(s) of data that you insert are written to one or more files in the directory that you specified.

Note: External tables that you create with a writable profile can only be used for INSERT operations. If you want to query the data that you inserted, you must create a separate readable external table that references the HDFS directory.

Use the following syntax to create a Greenplum Database external table that references an HDFS directory:

```
CREATE [WRITABLE] EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-hdfs-dir>
    ?PROFILE=hdfs:SequenceFile[&SERVER=<server_name>][&<custom-option>=<value>[...]]')
FORMAT 'CUSTOM' (<formatting-properties>)
[DISTRIBUTED BY (<column_name> [, ... ] ) | DISTRIBUTED RANDOMLY];
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-hdfs-dir>	The absolute path to the directory in the HDFS data store.
PROFILE	The `PROFILE` keyword must specify `hdfs:SequenceFile`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
<custom-option>	<custom-option>s are described below.
FORMAT	Use `FORMAT` '`CUSTOM`' with `(FORMATTER='pxfwritable_export')` (write) or `(FORMATTER='pxfwritable_import')` (read).
DISTRIBUTED BY	If you want to load data from an existing Greenplum Database table into the writable external table, consider specifying the same distribution policy or <column_name> on both tables. Doing so will avoid extra motion of data between segments on the load operation.

SequenceFile format data can optionally employ record or block compression. The PXF `hdfs:SequenceFile` profile supports the following compression codecs:

- org.apache.hadoop.io.compress.DefaultCodec
- org.apache.hadoop.io.compress.BZip2Codec

When you use the `hdfs:SequenceFile` profile to write SequenceFile format data, you must provide the name of the Java class to use for serializing/deserializing the binary data. This class must provide read and write methods for each data type referenced in the data schema.

You specify the compression codec and Java serialization class via custom options in the `CREATE EXTERNAL TABLE LOCATION` clause.

The `hdfs:SequenceFile` profile supports the following custom options:

Option	Value Description

COMPRESSION_CODEC	The compression codec Java class name. If this option is not provided, Greenplum Database performs no data compression. Supported compression codecs include: `org.apache.hadoop.io.compress.DefaultCodec` `org.apache.hadoop.io.compress.BZip2Codec` `org.apache.hadoop.io.compress.GzipCodec`
COMPRESSION_TYPE	The compression type to employ; supported values are `RECORD` (the default) or `BLOCK`.
DATA-SCHEMA	The name of the writer serialization/deserialization class. The jar file in which this class resides must be in the PXF classpath. This option is required for the `hdfs:SequenceFile` profile and has no default value.
THREAD-SAFE	Boolean value determining if a table query can run in multi-threaded mode. The default value is `TRUE`. Set this option to `FALSE` to handle all requests in a single thread for operations that are not thread-safe (for example, compression).
IGNORE_MISSING_PATH	A Boolean value that specifies the action to take when <path-to-hdfs-dir> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.

Reading and Writing Binary Data

Use the HDFS connector `hdfs:SequenceFile` profile when you want to read or write SequenceFile format data to HDFS. Files of this type consist of binary key/value pairs. SequenceFile format is a common data transfer format between MapReduce jobs.

Example: Writing Binary Data to HDFS

In this example, you create a Java class named `PxfExample_CustomWritable` that will serialize/deserialize the fields in the sample schema used in previous examples. You will then use this class to access a writable external table that you create with the `hdfs:SequenceFile` profile and that uses the default PXF server.

Perform the following procedure to create the Java class and writable table.

1. Prepare to create the sample Java class:

```
$ mkdir -p pxfex/com/example/pxf/hdfs/writable/dataschema
$ cd pxfex/com/example/pxf/hdfs/writable/dataschema
$ vi PxfExample_CustomWritable.java
```

2. Copy and paste the following text into the `PxfExample_CustomWritable.java` file:

```
package com.example.pxf.hdfs.writable.dataschema;

import org.apache.hadoop.io.*;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import java.lang.reflect.Field;

/**
* PxfExample_CustomWritable class - used to serialize and deserialize data with
* text, int, and float data types
*/
public class PxfExample_CustomWritable implements Writable {
```

```
public String st1, st2;
public int int1;
public float ft;

public PxfExample_CustomWritable() {
    st1 = new String("");
    st2 = new String("");
    int1 = 0;
    ft = 0.f;
}

public PxfExample_CustomWritable(int i1, int i2, int i3) {

    st1 = new String("short_string___" + i1);
    st2 = new String("short_string___" + i1);
    int1 = i2;
    ft = i1 * 10.f * 2.3f;

}

String GetSt1() {
    return st1;
}

String GetSt2() {
    return st2;
}

int GetInt1() {
    return int1;
}

float GetFt() {
    return ft;
}

@Override
public void write(DataOutput out) throws IOException {

    Text txt = new Text();
    txt.set(st1);
    txt.write(out);
    txt.set(st2);
    txt.write(out);

    IntWritable intw = new IntWritable();
    intw.set(int1);
    intw.write(out);

    FloatWritable fw = new FloatWritable();
    fw.set(ft);
    fw.write(out);
}

@Override
public void readFields(DataInput in) throws IOException {

    Text txt = new Text();
    txt.readFields(in);
    st1 = txt.toString();
```

```
    txt.readFields(in);
    st2 = txt.toString();

    IntWritable intw = new IntWritable();
    intw.readFields(in);
    int1 = intw.get();

    FloatWritable fw = new FloatWritable();
    fw.readFields(in);
    ft = fw.get();
}

public void printFieldTypes() {
    Class myClass = this.getClass();
    Field[] fields = myClass.getDeclaredFields();

    for (int i = 0; i < fields.length; i++) {
        System.out.println(fields[i].getType().getName());
    }
}
}
```

3. Compile and create a Java class JAR file for `PxfExample_CustomWritable`. Provide a classpath that includes the `hadoop-common.jar` file for your Hadoop distribution. For example, if you installed the Hortonworks Data Platform Hadoop client:

```
$ javac -classpath /usr/hdp/current/hadoop-client/hadoop-common.jar  PxfExample
_CustomWritable.java
$ cd ../../../../../../
$ jar cf pxfex-customwritable.jar com
$ cp pxfex-customwritable.jar /tmp/
```

   (Your Hadoop library classpath may differ.)

4. Copy the `pxfex-customwritable.jar` file to the Greenplum Database master node. For example:

```
$ scp pxfex-customwritable.jar gpadmin@gpmaster:/home/gpadmin
```

5. Log in to your Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

6. Copy the `pxfex-customwritable.jar` JAR file to the user runtime library directory, and note the location. For example, if `PXF_CONF=/usr/local/greenplum-pxf`:

```
gpadmin@gpmaster$ cp /home/gpadmin/pxfex-customwritable.jar /usr/local/greenplu
m-pxf/lib/pxfex-customwritable.jar
```

7. Synchronize the PXF configuration to the Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

8. Restart PXF on each Greenplum Database segment host as described in Restarting PXF.

9. Use the PXF `hdfs:SequenceFile` profile to create a Greenplum Database writable external

table. Identify the serialization/deserialization Java class you created above in the `DATA-SCHEMA` <custom-option>. Use `BLOCK` mode compression with `BZip2` when you create the writable table.

```
postgres=# CREATE WRITABLE EXTERNAL TABLE pxf_tbl_seqfile (location text, month
 text, number_of_orders integer, total_sales real)
            LOCATION ('pxf://data/pxf_examples/pxf_seqfile?PROFILE=hdfs:Sequenc
eFile&DATA-SCHEMA=com.example.pxf.hdfs.writable.dataschema.PxfExample_CustomWri
table&COMPRESSION_TYPE=BLOCK&COMPRESSION_CODEC=org.apache.hadoop.io.compress.BZ
ip2Codec')
          FORMAT 'CUSTOM' (FORMATTER='pxfwritable_export');
```

Notice that the `'CUSTOM' FORMAT` <formatting-properties> specifies the built-in `pxfwritable_export` formatter.

10. Write a few records to the `pxf_seqfile` HDFS directory by inserting directly into the `pxf_tbl_seqfile` table. For example:

```
postgres=# INSERT INTO pxf_tbl_seqfile VALUES ( 'Frankfurt', 'Mar', 777, 3956.9
8 );
postgres=# INSERT INTO pxf_tbl_seqfile VALUES ( 'Cleveland', 'Oct', 3812, 96645
.37 );
```

11. Recall that Greenplum Database does not support directly querying a writable external table. To read the data in `pxf_seqfile`, create a readable external Greenplum Database referencing this HDFS directory:

```
postgres=# CREATE EXTERNAL TABLE read_pxf_tbl_seqfile (location text, month tex
t, number_of_orders integer, total_sales real)
            LOCATION ('pxf://data/pxf_examples/pxf_seqfile?PROFILE=hdfs:Sequenc
eFile&DATA-SCHEMA=com.example.pxf.hdfs.writable.dataschema.PxfExample_CustomWri
table')
          FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

You must specify the `DATA-SCHEMA` <custom-option> when you read HDFS data via the `hdfs:SequenceFile` profile. You need not provide compression-related options.

12. Query the readable external table `read_pxf_tbl_seqfile`:

```
gpadmin=# SELECT * FROM read_pxf_tbl_seqfile ORDER BY total_sales;
```

```
 location  | month | number_of_orders | total_sales
-----------+-------+------------------+-------------
 Frankfurt | Mar   |              777 |     3956.98
 Cleveland | Oct   |             3812 |     96645.4
(2 rows)
```

Reading the Record Key

When a Greenplum Database external table references SequenceFile or another data format that stores rows in a key-value format, you can access the key values in Greenplum queries by using the `recordkey` keyword as a field name.

The field type of `recordkey` must correspond to the key type, much as the other fields must match

the HDFS data.

You can define `recordkey` to be any of the following Hadoop types:

- BooleanWritable
- ByteWritable
- DoubleWritable
- FloatWritable
- IntWritable
- LongWritable
- Text

If no record key is defined for a row, Greenplum Database returns the id of the segment that processed the row.

Example: Using Record Keys

Create an external readable table to access the record keys from the writable table `pxf_tbl_seqfile` that you created in Example: Writing Binary Data to HDFS. Define the `recordkey` in this example to be of type `int8`.

```
postgres=# CREATE EXTERNAL TABLE read_pxf_tbl_seqfile_recordkey(recordkey int8, locati
on text, month text, number_of_orders integer, total_sales real)
                LOCATION ('pxf://data/pxf_examples/pxf_seqfile?PROFILE=hdfs:SequenceFi
le&DATA-SCHEMA=com.example.pxf.hdfs.writable.dataschema.PxfExample_CustomWritable')
          FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
gpadmin=# SELECT * FROM read_pxf_tbl_seqfile_recordkey;
```

```
 recordkey |  location   | month | number_of_orders | total_sales
-----------+-------------+-------+------------------+-------------
         2 | Frankfurt   | Mar   |              777 |     3956.98
         1 | Cleveland   | Oct   |             3812 |     96645.4
(2 rows)
```

You did not define a record key when you inserted the rows into the writable table, so the `recordkey` identifies the segment on which the row data was processed.

Reading a Multi-Line Text File into a Single Table Row

You can use the PXF HDFS connector to read one or more multi-line text files in HDFS each as a single table row. This may be useful when you want to read multiple files into the same Greenplum Database external table, for example when individual JSON files each contain a separate record.

PXF supports reading only text and JSON files in this manner.

Note: Refer to the Reading JSON Data from HDFS topic if you want to use PXF to read JSON files that include more than one record.

Prerequisites

Ensure that you have met the PXF Hadoop Prerequisites before you attempt to read files from

HDFS.

Reading Multi-Line Text and JSON Files

You can read single- and multi-line files into a single table row, including files with embedded linefeeds. If you are reading multiple JSON files, each file must be a complete record, and each file must contain the same record type.

PXF reads the complete file data into a single row and column. When you create the external table to read multiple files, you must ensure that all of the files that you want to read are of the same (text or JSON) type. You must also specify a single `text` or `json` column, depending upon the file type.

The following syntax creates a Greenplum Database readable external table that references one or more text or JSON files on HDFS:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> text|json | LIKE <other_table> )
  LOCATION ('pxf://<path-to-files>?PROFILE=hdfs:text:multi[&SERVER=<server_name>][&IGN
ORE_MISSING_PATH=<boolean>]&FILE_AS_ROW=true')
FORMAT 'CSV');
```

The keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-files>	The absolute path to the directory or files in the HDFS data store.
PROFILE	The `PROFILE` keyword must specify `hdfs:text:multi`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
FILE_AS_ROW=true	The required option that instructs PXF to read each file into a single table row.
IGNORE_MISSING_PATH=<boolean>	Specify the action to take when <path-to-files> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.
FORMAT	The `FORMAT` must specify `'CSV'`.

Note: The `hdfs:text:multi` profile does not support additional format options when you specify the `FILE_AS_ROW=true` option.

For example, if `/data/pxf_examples/jdir` identifies an HDFS directory that contains a number of JSON files, the following statement creates a Greenplum Database external table that references all of the files in that directory:

```
CREATE EXTERNAL TABLE pxf_readjfiles(j1 json)
  LOCATION ('pxf://data/pxf_examples/jdir?PROFILE=hdfs:text:multi&FILE_AS_ROW=true')
FORMAT 'CSV';
```

When you query the `pxf_readjfiles` table with a `SELECT` statement, PXF returns the contents of each JSON file in `jdir/` as a separate row in the external table.

When you read JSON files, you can use the JSON functions provided in Greenplum Database to access individual data fields in the JSON record. For example, if the `pxf_readjfiles` external table

above reads a JSON file that contains this JSON record:

```
{
  "root":[
    {
      "record_obj":{
        "created_at":"MonSep3004:04:53+00002013",
        "id_str":"384529256681725952",
        "user":{
          "id":31424214,
          "location":"COLUMBUS"
        },
        "coordinates":null
      }
    }
  ]
}
```

You can use the `json_array_elements()` function to extract specific JSON fields from the table row. For example, the following command displays the `user->id` field:

```
SELECT json_array_elements(j1->'root')->'record_obj'->'user'->'id'
  AS userid FROM pxf_readjfiles;

  userid
----------
 31424214
(1 rows)
```

Refer to Working with JSON Data in the Greenplum Documentation for specific information on manipulating JSON data in Greenplum.

Example: Reading an HDFS Text File into a Single Table Row

Perform the following procedure to create 3 sample text files in an HDFS directory, and use the PXF `hdfs:text:multi` profile and the default PXF server to read all of these text files in a single external table query.

1. Create an HDFS directory for the text files. For example:

   ```
   $ hdfs dfs -mkdir -p /data/pxf_examples/tdir
   ```

2. Create a text data file named `file1.txt`:

   ```
   $ echo 'text file with only one line' > /tmp/file1.txt
   ```

3. Create a second text data file named `file2.txt`:

   ```
   $ echo 'Prague,Jan,101,4875.33
   Rome,Mar,87,1557.39
   Bangalore,May,317,8936.99
   Beijing,Jul,411,11600.67' > /tmp/file2.txt
   ```

   This file has multiple lines.

4. Create a third text file named `/tmp/file3.txt`:

```
$ echo '"4627 Star Rd.
San Francisco, CA  94107":Sept:2017
"113 Moon St.
San Diego, CA  92093":Jan:2018
"51 Belt Ct.
Denver, CO  90123":Dec:2016
"93114 Radial Rd.
Chicago, IL  60605":Jul:2017
"7301 Brookview Ave.
Columbus, OH  43213":Dec:2018' > /tmp/file3.txt
```

This file includes embedded line feeds.

5. Save the file and exit the editor.
6. Copy the text files to HDFS:

```
$ hdfs dfs -put /tmp/file1.txt /data/pxf_examples/tdir
$ hdfs dfs -put /tmp/file2.txt /data/pxf_examples/tdir
$ hdfs dfs -put /tmp/file3.txt /data/pxf_examples/tdir
```

7. Log in to a Greenplum Database system and start the `psql` subsystem.
8. Use the `hdfs:text:multi` profile to create an external table that references the `tdir` HDFS directory. For example:

```
CREATE EXTERNAL TABLE pxf_readfileasrow(c1 text)
  LOCATION ('pxf://data/pxf_examples/tdir?PROFILE=hdfs:text:multi&FILE_AS_ROW=t
rue')
FORMAT 'CSV';
```

9. Turn on expanded display and query the `pxf_readfileasrow` table:

```
postgres=# \x on
postgres=# SELECT * FROM pxf_readfileasrow;
```

```
-[ RECORD 1 ]--------------------------
c1 | Prague,Jan,101,4875.33
   | Rome,Mar,87,1557.39
   | Bangalore,May,317,8936.99
   | Beijing,Jul,411,11600.67
-[ RECORD 2 ]--------------------------
c1 | text file with only one line
-[ RECORD 3 ]--------------------------
c1 | "4627 Star Rd.
   | San Francisco, CA  94107":Sept:2017
   | "113 Moon St.
   | San Diego, CA  92093":Jan:2018
   | "51 Belt Ct.
   | Denver, CO  90123":Dec:2016
   | "93114 Radial Rd.
   | Chicago, IL  60605":Jul:2017
   | "7301 Brookview Ave.
   | Columbus, OH  43213":Dec:2018
```

Reading Hive Table Data

Apache Hive is a distributed data warehousing infrastructure. Hive facilitates managing large data sets supporting multiple data formats, including comma-separated value (.csv) TextFile, RCFile, ORC, and Parquet.

The PXF Hive connector reads data stored in a Hive table. This section describes how to use the PXF Hive connector.

Prerequisites

Before working with Hive table data using PXF, ensure that you have met the PXF Hadoop Prerequisites.

If you plan to use PXF filter pushdown with Hive integral types, ensure that the configuration parameter `hive.metastore.integral.jdo.pushdown` exists and is set to `true` in the `hive-site.xml` file in both your Hadoop cluster **and** `$PXF_CONF/servers/default/hive-site.xml`. Refer to About Updating Hadoop Configuration for more information.

Hive Data Formats

The PXF Hive connector supports several data formats, and has defined the following profiles for accessing these formats:

File Format	Description	Profile
TextFile	Flat file with data in comma-, tab-, or space-separated value format or JSON notation.	Hive, HiveText
SequenceFile	Flat file consisting of binary key/value pairs.	Hive
RCFile	Record columnar data consisting of binary key/value pairs; high row compression rate.	Hive, HiveRC
ORC	Optimized row columnar data with stripe, footer, and postscript sections; reduces data size.	Hive, HiveORC, HiveVectorizedORC
Parquet	Compressed columnar data representation.	Hive

Note: The `Hive` profile supports all file storage formats. It will use the optimal `Hive*` profile for the underlying file format type.

Data Type Mapping

The PXF Hive connector supports primitive and complex data types.

Primitive Data Types

To represent Hive data in Greenplum Database, map data values that use a primitive data type to Greenplum Database columns of the same type.

The following table summarizes external mapping rules for Hive primitive types.

Hive Data Type	Greenplum Data Type
boolean	bool

Hive Data Type	Greenplum Data Type
int	int4
smallint	int2
tinyint	int2
bigint	int8
float	float4
double	float8
string	text
binary	bytea
timestamp	timestamp

Note: The `HiveVectorizedORC` profile does not support the timestamp data type.

Complex Data Types

Hive supports complex data types including array, struct, map, and union. PXF maps each of these complex types to `text`. You can create Greenplum Database functions or application code to extract subcomponents of these complex data types.

Examples using complex data types with the `Hive` and `HiveORC` profiles are provided later in this topic.

Note: The `HiveVectorizedORC` profile does not support complex types.

Sample Data Set

Examples presented in this topic operate on a common data set. This simple data set models a retail sales operation and includes fields with the following names and data types:

Column Name	Data Type
location	text
month	text
number_of_orders	integer
total_sales	double

Prepare the sample data set for use:

1. First, create a text file:

   ```
   $ vi /tmp/pxf_hive_datafile.txt
   ```

2. Add the following data to `pxf_hive_datafile.txt`; notice the use of the comma `,` to separate the four field values:

   ```
   Prague,Jan,101,4875.33
   Rome,Mar,87,1557.39
   ```

```
Bangalore,May,317,8936.99
Beijing,Jul,411,11600.67
San Francisco,Sept,156,6846.34
Paris,Nov,159,7134.56
San Francisco,Jan,113,5397.89
Prague,Dec,333,9894.77
Bangalore,Jul,271,8320.55
Beijing,Dec,100,4248.41
```

Make note of the path to `pxf_hive_datafile.txt`; you will use it in later exercises.

Hive Command Line

The Hive command line is a subsystem similar to that of `psql`. To start the Hive command line:

```
$ HADOOP_USER_NAME=hdfs hive
```

The default Hive database is named `default`.

Example: Creating a Hive Table

Create a Hive table to expose the sample data set.

1. Create a Hive table named `sales_info` in the `default` database:

   ```
   hive> CREATE TABLE sales_info (location string, month string,
           number_of_orders int, total_sales double)
           ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
           STORED AS textfile;
   ```

   Notice that:

   - The `STORED AS textfile` subclause instructs Hive to create the table in Textfile (the default) format. Hive Textfile format supports comma-, tab-, and space-separated values, as well as data specified in JSON notation.
   - The `DELIMITED FIELDS TERMINATED BY` subclause identifies the field delimiter within a data record (line). The `sales_info` table field delimiter is a comma (`,`).

2. Load the `pxf_hive_datafile.txt` sample data file into the `sales_info` table that you just created:

   ```
   hive> LOAD DATA LOCAL INPATH '/tmp/pxf_hive_datafile.txt'
           INTO TABLE sales_info;
   ```

   In examples later in this section, you will access the `sales_info` Hive table directly via PXF. You will also insert `sales_info` data into tables of other Hive file format types, and use PXF to access those directly as well.

3. Perform a query on `sales_info` to verify that you loaded the data successfully:

   ```
   hive> SELECT * FROM sales_info;
   ```

Determining the HDFS Location of a Hive Table

Should you need to identify the HDFS file location of a Hive managed table, reference it using its HDFS file path. You can determine a Hive table's location in HDFS using the `DESCRIBE` command. For example:

```
hive> DESCRIBE EXTENDED sales_info;
Detailed Table Information
...
location:hdfs://<namenode>:<port>/apps/hive/warehouse/sales_info
...
```

Querying External Hive Data

You can create a Greenplum Database external table to access Hive table data. As described previously, the PXF Hive connector defines specific profiles to support different file formats. These profiles are named `Hive`, `HiveText`, and `HiveRC`, `HiveORC`, and `HiveVectorizedORC`.

The `HiveText` and `HiveRC` profiles are specifically optimized for text and RCFile formats, respectively. The `HiveORC` and `HiveVectorizedORC` profiles are optimized for ORC file formats. The `Hive` profile is optimized for all file storage types; you can use the `Hive` profile when the underlying Hive table is composed of multiple partitions with differing file formats.

PXF uses column projection to increase query performance when you access a Hive table using the `Hive`, `HiveRC`, `HiveORC`, or `HiveVectorizedORC` profiles.

Use the following syntax to create a Greenplum Database external table that references a Hive table:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<hive-db-name>.<hive-table-name>
    ?PROFILE=Hive|HiveText|HiveRC|HiveORC|HiveVectorizedORC[&SERVER=<server_name>]'])
FORMAT 'CUSTOM|TEXT' (FORMATTER='pxfwritable_import' | delimiter='<delim>')
```

Hive connector-specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE call are described below.

Keyword	Value
<hive-db-name>	The name of the Hive database. If omitted, defaults to the Hive database named `default`.
<hive-table-name>	The name of the Hive table.
PROFILE	The `PROFILE` keyword must specify one of the values `Hive`, `HiveText`, `HiveRC`, `HiveORC`, or `HiveVectorizedORC`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
FORMAT (`Hive`, `HiveORC`, and `HiveVectorizedORC` profiles)	The `FORMAT` clause must specify `'CUSTOM'`. The `CUSTOM` format requires the built-in `pxfwritable_import formatter`.
FORMAT (`HiveText` and `HiveRC` profiles)	The `FORMAT` clause must specify `TEXT`. Specify the single ascii character field delimiter in the `delimiter='<delim>'` formatting option.

Because Hive tables can be backed by one or more files and each file can have a unique layout or schema, PXF requires that the column names that you specify when you create the external table match the column names defined for the Hive table. This allows you to:

- Create the PXF external table with columns in a different order than the Hive table.
- Create a PXF external table that reads a subset of the columns in the Hive table.
- Read a Hive table where the files backing the table have a different number of columns.

Accessing TextFile-Format Hive Tables

You can use the `Hive` and `HiveText` profiles to access Hive table data stored in TextFile format.

Example: Using the Hive Profile

Use the `Hive` profile to create a readable Greenplum Database external table that references the Hive `sales_info` textfile format table that you created earlier.

1. Create the external table:

```
postgres=# CREATE EXTERNAL TABLE salesinfo_hiveprofile(location text, month tex
t, number_of_orders int, total_sales float8)
            LOCATION ('pxf://default.sales_info?PROFILE=Hive')
          FORMAT 'custom' (FORMATTER='pxfwritable_import');
```

2. Query the table:

```
postgres=# SELECT * FROM salesinfo_hiveprofile;
```

```
   location    | month | number_of_orders | total_sales
---------------+-------+------------------+-------------
 Prague        | Jan   |              101 |     4875.33
 Rome          | Mar   |               87 |     1557.39
 Bangalore     | May   |              317 |     8936.99
 ...
```

Example: Using the HiveText Profile

Use the PXF `HiveText` profile to create a readable Greenplum Database external table from the Hive `sales_info` textfile format table that you created earlier.

1. Create the external table:

```
postgres=# CREATE EXTERNAL TABLE salesinfo_hivetextprofile(location text, month
 text, number_of_orders int, total_sales float8)
            LOCATION ('pxf://default.sales_info?PROFILE=HiveText')
          FORMAT 'TEXT' (delimiter=E',');
```

   Notice that the `FORMAT` subclause `delimiter` value is specified as the single ascii comma character `','`. `E` escapes the character.

2. Query the external table:

```
postgres=# SELECT * FROM salesinfo_hivetextprofile WHERE location='Beijing';
```

```
 location | month | number_of_orders | total_sales
----------+-------+------------------+-------------
 Beijing  | Jul   |              411 |    11600.67
```

```
 Beijing   | Dec    |              100 |     4248.41
(2 rows)
```

Accessing RCFile-Format Hive Tables

The RCFile Hive table format is used for row columnar formatted data. The PXF `HiveRC` profile provides access to RCFile data.

Example: Using the HiveRC Profile

Use the `HiveRC` profile to query RCFile-formatted data in a Hive table.

1. Start the `hive` command line and create a Hive table stored in RCFile format:

   ```
   $ HADOOP_USER_NAME=hdfs hive
   ```

   ```
   hive> CREATE TABLE sales_info_rcfile (location string, month string,
           number_of_orders int, total_sales double)
         ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
         STORED AS rcfile;
   ```

2. Insert the data from the `sales_info` table into `sales_info_rcfile`:

   ```
   hive> INSERT INTO TABLE sales_info_rcfile SELECT * FROM sales_info;
   ```

   A copy of the sample data set is now stored in RCFile format in the Hive `sales_info_rcfile` table.

3. Query the `sales_info_rcfile` Hive table to verify that the data was loaded correctly:

   ```
   hive> SELECT * FROM sales_info_rcfile;
   ```

4. Use the PXF `HiveRC` profile to create a readable Greenplum Database external table that references the Hive `sales_info_rcfile` table that you created in the previous steps. For example:

   ```
   postgres=# CREATE EXTERNAL TABLE salesinfo_hivercprofile(location text, month t
   ext, number_of_orders int, total_sales float8)
               LOCATION ('pxf://default.sales_info_rcfile?PROFILE=HiveRC')
             FORMAT 'TEXT' (delimiter=E',');
   ```

5. Query the external table:

   ```
   postgres=# SELECT location, total_sales FROM salesinfo_hivercprofile;
   ```

   ```
      location    | total_sales
   ---------------+-------------
    Prague        |     4875.33
    Rome          |     1557.39
    Bangalore     |     8936.99
    Beijing       |    11600.67
    ...
   ```

Accessing ORC-Format Hive Tables

The Optimized Row Columnar (ORC) file format is a columnar file format that provides a highly efficient way to both store and access HDFS data. ORC format offers improvements over text and RCFile formats in terms of both compression and performance. PXF supports ORC version 1.2.1.

ORC is type-aware and specifically designed for Hadoop workloads. ORC files store both the type of and encoding information for the data in the file. All columns within a single group of row data (also known as stripe) are stored together on disk in ORC format files. The columnar nature of the ORC format type enables read projection, helping avoid accessing unnecessary columns during a query.

ORC also supports predicate pushdown with built-in indexes at the file, stripe, and row levels, moving the filter operation to the data loading phase.

Refer to the Apache orc and the Apache Hive LanguageManual ORC websites for detailed information about the ORC file format.

Profiles Supporting the ORC File Format

When choosing an ORC-supporting profile, consider the following:

- The `HiveORC` profile:
    - Reads a single row of data at a time.
    - Supports column projection.
    - Supports complex types. You can access Hive tables composed of array, map, struct, and union data types. PXF serializes each of these complex types to `text`.
- The `HiveVectorizedORC` profile:
    - Reads up to 1024 rows of data at once.
    - Does not support column projection.
    - Does not support complex types or the timestamp data type.

Example: Using the HiveORC Profile

In the following example, you will create a Hive table stored in ORC format and use the `HiveORC` profile to query this Hive table.

1. Create a Hive table with ORC file format:

    ```
    $ HADOOP_USER_NAME=hdfs hive
    ```

    ```
    hive> CREATE TABLE sales_info_ORC (location string, month string,
            number_of_orders int, total_sales double)
          STORED AS ORC;
    ```

2. Insert the data from the `sales_info` table into `sales_info_ORC`:

    ```
    hive> INSERT INTO TABLE sales_info_ORC SELECT * FROM sales_info;
    ```

    A copy of the sample data set is now stored in ORC format in `sales_info_ORC`.

3. Perform a Hive query on `sales_info_ORC` to verify that the data was loaded successfully:

```
hive> SELECT * FROM sales_info_ORC;
```

4. Start the `psql` subsystem and turn on timing:

```
$ psql -d postgres
```

```
postgres=> \timing
Timing is on.
```

5. Use the PXF `HiveORC` profile to create a Greenplum Database external table that references the Hive table named `sales_info_ORC` you created in Step 1. The `FORMAT` clause must specify `'CUSTOM'`. The `HiveORC CUSTOM` format supports only the built-in `'pxfwritable_import' formatter`.

```
postgres=> CREATE EXTERNAL TABLE salesinfo_hiveORCprofile(location text, month
text, number_of_orders int, total_sales float8)
             LOCATION ('pxf://default.sales_info_ORC?PROFILE=HiveORC')
             FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

6. Query the external table:

```
postgres=> SELECT * FROM salesinfo_hiveORCprofile;
```

```
   location    | month | number_of_orders | total_sales
---------------+-------+------------------+-------------
 Prague        | Jan   |              101 |     4875.33
 Rome          | Mar   |               87 |     1557.39
 Bangalore     | May   |              317 |     8936.99
 ...

Time: 425.416 ms
```

Example: Using the HiveVectorizedORC Profile

In the following example, you will use the `HiveVectorizedORC` profile to query the `sales_info_ORC` Hive table that you created in the previous example.

1. Start the `psql` subsystem:

```
$ psql -d postgres
```

2. Use the PXF `HiveVectorizedORC` profile to create a readable Greenplum Database external table that references the Hive table named `sales_info_ORC` that you created in Step 1 of the previous example. The `FORMAT` clause must specify `'CUSTOM'`. The `HiveVectorizedORC CUSTOM` format supports only the built-in `'pxfwritable_import' formatter`.

```
postgres=> CREATE EXTERNAL TABLE salesinfo_hiveVectORC(location text, month tex
t, number_of_orders int, total_sales float8)
             LOCATION ('pxf://default.sales_info_ORC?PROFILE=HiveVectorizedORC'
)
             FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

3. Query the external table:

```
postgres=> SELECT * FROM salesinfo_hiveVectORC;
```

```
   location    | month | number_of_orders | total_sales
---------------+-------+------------------+-------------
 Prague        | Jan   |              101 |     4875.33
 Rome          | Mar   |               87 |     1557.39
 Bangalore     | May   |              317 |     8936.99
 ...

Time: 425.416 ms
```

Accessing Parquet-Format Hive Tables

The PXF `Hive` profile supports both non-partitioned and partitioned Hive tables that use the Parquet storage format. Map the table columns using equivalent Greenplum Database data types. For example, if a Hive table is created in the `default` schema using:

```
hive> CREATE TABLE hive_parquet_table (location string, month string,
            number_of_orders int, total_sales double)
        STORED AS parquet;
```

Define the Greenplum Database external table:

```
postgres=# CREATE EXTERNAL TABLE pxf_parquet_table (location text, month text, number_
of_orders int, total_sales double precision)
    LOCATION ('pxf://default.hive_parquet_table?profile=Hive')
    FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

And query the table:

```
postgres=# SELECT month, number_of_orders FROM pxf_parquet_table;
```

PXF does not support reading a Hive table stored as Parquet if the table is backed by more than one Parquet file and the columns are in a different order. This limitation applies even when the Parquet files have the same column names in their schema.

Working with Complex Data Types

Example: Using the Hive Profile with Complex Data Types

This example employs the `Hive` profile and the array and map complex types, specifically an array of integers and a string key/value pair map.

The data schema for this example includes fields with the following names and data types:

Column Name	Data Type
index	int
name	string

Column Name	Data Type
intarray	array of integers
propmap	map of string key and value pairs

When you specify an array field in a Hive table, you must identify the terminator for each item in the collection. Similarly, you must also specify the map key termination character.

1. Create a text file from which you will load the data set:

   ```
   $ vi /tmp/pxf_hive_complex.txt
   ```

2. Add the following text to `pxf_hive_complex.txt`. This data uses a comma `,` to separate field values, the percent symbol `%` to separate collection items, and a `:` to terminate map key values:

   ```
   3,Prague,1%2%3,zone:euro%status:up
   89,Rome,4%5%6,zone:euro
   400,Bangalore,7%8%9,zone:apac%status:pending
   183,Beijing,0%1%2,zone:apac
   94,Sacramento,3%4%5,zone:noam%status:down
   101,Paris,6%7%8,zone:euro%status:up
   56,Frankfurt,9%0%1,zone:euro
   202,Jakarta,2%3%4,zone:apac%status:up
   313,Sydney,5%6%7,zone:apac%status:pending
   76,Atlanta,8%9%0,zone:noam%status:down
   ```

3. Create a Hive table to represent this data:

   ```
   $ HADOOP_USER_NAME=hdfs hive
   ```

   ```
   hive> CREATE TABLE table_complextypes( index int, name string, intarray ARRAY<i
   nt>, propmap MAP<string, string>)
            ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
            COLLECTION ITEMS TERMINATED BY '%'
            MAP KEYS TERMINATED BY ':'
            STORED AS TEXTFILE;
   ```

   Notice that:

   - `FIELDS TERMINATED BY` identifies a comma as the field terminator.
   - The `COLLECTION ITEMS TERMINATED BY` subclause specifies the percent sign as the collection items (array item, map key/value pair) terminator.
   - `MAP KEYS TERMINATED BY` identifies a colon as the terminator for map keys.

4. Load the `pxf_hive_complex.txt` sample data file into the `table_complextypes` table that you just created:

   ```
   hive> LOAD DATA LOCAL INPATH '/tmp/pxf_hive_complex.txt' INTO TABLE table_compl
   extypes;
   ```

5. Perform a query on Hive table `table_complextypes` to verify that the data was loaded successfully:

```
hive> SELECT * FROM table_complextypes;
```

```
3   Prague   [1,2,3] {"zone":"euro","status":"up"}
89  Rome     [4,5,6] {"zone":"euro"}
400 Bangalore   [7,8,9] {"zone":"apac","status":"pending"}
...
```

6. Use the PXF `Hive` profile to create a readable Greenplum Database external table that references the Hive table named `table_complextypes`:

```
postgres=# CREATE EXTERNAL TABLE complextypes_hiveprofile(index int, name text,
 intarray text, propmap text)
             LOCATION ('pxf://table_complextypes?PROFILE=Hive')
           FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

   Notice that the integer array and map complex types are mapped to Greenplum Database data type text.

7. Query the external table:

```
postgres=# SELECT * FROM complextypes_hiveprofile;
```

```
 index |    name    | intarray |              propmap
-------+------------+----------+-----------------------------------
     3 | Prague     | [1,2,3]  | {"zone":"euro","status":"up"}
    89 | Rome       | [4,5,6]  | {"zone":"euro"}
   400 | Bangalore  | [7,8,9]  | {"zone":"apac","status":"pending"}
   183 | Beijing    | [0,1,2]  | {"zone":"apac"}
    94 | Sacramento | [3,4,5]  | {"zone":"noam","status":"down"}
   101 | Paris      | [6,7,8]  | {"zone":"euro","status":"up"}
    56 | Frankfurt  | [9,0,1]  | {"zone":"euro"}
   202 | Jakarta    | [2,3,4]  | {"zone":"apac","status":"up"}
   313 | Sydney     | [5,6,7]  | {"zone":"apac","status":"pending"}
    76 | Atlanta    | [8,9,0]  | {"zone":"noam","status":"down"}
(10 rows)
```

   `intarray` and `propmap` are each serialized as text strings.

Example: Using the HiveORC Profile with Complex Data Types

In the following example, you will create and populate a Hive table stored in ORC format. You will use the `HiveORC` profile to query the complex types in this Hive table.

1. Create a Hive table with ORC storage format:

```
$ HADOOP_USER_NAME=hdfs hive
```

```
hive> CREATE TABLE table_complextypes_ORC( index int, name string, intarray ARR
AY<int>, propmap MAP<string, string>)
        ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
        COLLECTION ITEMS TERMINATED BY '%'
        MAP KEYS TERMINATED BY ':'
      STORED AS ORC;
```

2. Insert the data from the `table_complextypes` table that you created in the previous example

into `table_complextypes_ORC`:

```
hive> INSERT INTO TABLE table_complextypes_ORC SELECT * FROM table_complextypes
;
```

A copy of the sample data set is now stored in ORC format in `table_complextypes_ORC`.

3. Perform a Hive query on `table_complextypes_ORC` to verify that the data was loaded successfully:

```
hive> SELECT * FROM table_complextypes_ORC;
```

```
OK
3       Prague        [1,2,3]     {"zone":"euro","status":"up"}
89      Rome          [4,5,6]     {"zone":"euro"}
400     Bangalore     [7,8,9]     {"zone":"apac","status":"pending"}
...
```

4. Start the `psql` subsystem:

```
$ psql -d postgres
```

5. Use the PXF `HiveORC` profile to create a readable Greenplum Database external table from the Hive table named `table_complextypes_ORC` you created in Step 1. The `FORMAT` clause must specify `'CUSTOM'`. The `HiveORC CUSTOM` format supports only the built-in `'pxfwritable_import' formatter`.

```
postgres=> CREATE EXTERNAL TABLE complextypes_hiveorc(index int, name text, int
array text, propmap text)
           LOCATION ('pxf://default.table_complextypes_ORC?PROFILE=HiveORC')
             FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

Notice that the integer array and map complex types are again mapped to Greenplum Database data type text.

6. Query the external table:

```
postgres=> SELECT * FROM complextypes_hiveorc;
```

```
 index |    name    | intarray |              propmap
-------+------------+----------+-------------------------------------
     3 | Prague     | [1,2,3]  | {"zone":"euro","status":"up"}
    89 | Rome       | [4,5,6]  | {"zone":"euro"}
   400 | Bangalore  | [7,8,9]  | {"zone":"apac","status":"pending"}
 ...
```

`intarray` and `propmap` are again serialized as text strings.

Partition Filter Pushdown

The PXF Hive connector supports Hive partitioning pruning and the Hive partition directory structure. This enables partition exclusion on selected HDFS files comprising a Hive table. To use the

partition filtering feature to reduce network traffic and I/O, run a query on a PXF external table using a `WHERE` clause that refers to a specific partition column in a partitioned Hive table.

The PXF Hive Connector partition filtering support for Hive string and integral types is described below:

- The relational operators `=`, `<`, `<=`, `>`, `>=`, and `<>` are supported on string types.
- The relational operators `=` and `<>` are supported on integral types (To use partition filtering with Hive integral types, you must update the Hive configuration as described in the Prerequisites).
- The logical operators `AND` and `OR` are supported when used with the relational operators mentioned above.
- The `LIKE` string operator is not supported.

To take advantage of PXF partition filtering pushdown, the Hive and PXF partition field names must be the same. Otherwise, PXF ignores partition filtering and the filtering is performed on the Greenplum Database side, impacting performance.

The PXF Hive connector filters only on partition columns, not on other table attributes. Additionally, filter pushdown is supported only for those data types and operators identified above.

PXF filter pushdown is enabled by default. You configure PXF filter pushdown as described in About Filter Pushdown.

Example: Using the Hive Profile to Access Partitioned Homogenous Data

In this example, you use the `Hive` profile to query a Hive table named `sales_part` that you partition on the `delivery_state` and `delivery_city` fields. You then create a Greenplum Database external table to query `sales_part`. The procedure includes specific examples that illustrate filter pushdown.

1. Create a Hive table named `sales_part` with two partition columns, `delivery_state` and `delivery_city`:

```
hive> CREATE TABLE sales_part (cname string, itype string, supplier_key int, pr
ice double)
        PARTITIONED BY (delivery_state string, delivery_city string)
        ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

2. Load data into this Hive table and add some partitions:

```
hive> INSERT INTO TABLE sales_part
        PARTITION(delivery_state = 'CALIFORNIA', delivery_city = 'Fresno')
        VALUES ('block', 'widget', 33, 15.17);
hive> INSERT INTO TABLE sales_part
        PARTITION(delivery_state = 'CALIFORNIA', delivery_city = 'Sacramento')
        VALUES ('cube', 'widget', 11, 1.17);
hive> INSERT INTO TABLE sales_part
        PARTITION(delivery_state = 'NEVADA', delivery_city = 'Reno')
        VALUES ('dowel', 'widget', 51, 31.82);
hive> INSERT INTO TABLE sales_part
        PARTITION(delivery_state = 'NEVADA', delivery_city = 'Las Vegas')
        VALUES ('px49', 'pipe', 52, 99.82);
```

3. Query the `sales_part` table:

```
hive> SELECT * FROM sales_part;
```

A `SELECT *` statement on a Hive partitioned table shows the partition fields at the end of the record.

4. Examine the Hive/HDFS directory structure for the `sales_part` table:

```
$ sudo -u hdfs hdfs dfs -ls -R /apps/hive/warehouse/sales_part
/apps/hive/warehouse/sales_part/delivery_state=CALIFORNIA/delivery_city=Fresno/
/apps/hive/warehouse/sales_part/delivery_state=CALIFORNIA/delivery_city=Sacrame
nto/
/apps/hive/warehouse/sales_part/delivery_state=NEVADA/delivery_city=Reno/
/apps/hive/warehouse/sales_part/delivery_state=NEVADA/delivery_city=Las Vegas/
```

5. Create a PXF external table to read the partitioned `sales_part` Hive table. To take advantage of partition filter push-down, define fields corresponding to the Hive partition fields at the end of the `CREATE EXTERNAL TABLE` attribute list.

```
$ psql -d postgres
```

```
postgres=# CREATE EXTERNAL TABLE pxf_sales_part(
             cname TEXT, itype TEXT,
             supplier_key INTEGER, price DOUBLE PRECISION,
             delivery_state TEXT, delivery_city TEXT)
           LOCATION ('pxf://sales_part?Profile=Hive')
           FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

6. Query the table:

```
postgres=# SELECT * FROM pxf_sales_part;
```

7. Perform another query (no pushdown) on `pxf_sales_part` to return records where the `delivery_city` is `Sacramento` and `cname` is `cube`:

```
postgres=# SELECT * FROM pxf_sales_part WHERE delivery_city = 'Sacramento' AND
cname = 'cube';
```

The query filters the `delivery_city` partition `Sacramento`. The filter on `cname` is not pushed down, since it is not a partition column. It is performed on the Greenplum Database side after all the data in the `Sacramento` partition is transferred for processing.

8. Query (with pushdown) for all records where `delivery_state` is `CALIFORNIA`:

```
postgres=# SET gp_external_enable_filter_pushdown=on;
postgres=# SELECT * FROM pxf_sales_part WHERE delivery_state = 'CALIFORNIA';
```

This query reads all of the data in the `CALIFORNIA delivery_state` partition, regardless of the city.

Example: Using the Hive Profile to Access Partitioned Heterogeneous Data

You can use the PXF `Hive` profile with any Hive file storage types. With the `Hive` profile, you can access heterogeneous format data in a single Hive table where the partitions may be stored in different file formats.

In this example, you create a partitioned Hive external table. The table is composed of the HDFS data files associated with the `sales_info` (text format) and `sales_info_rcfile` (RC format) Hive tables that you created in previous exercises. You will partition the data by year, assigning the data from `sales_info` to the year 2013, and the data from `sales_info_rcfile` to the year 2016. (Ignore at the moment the fact that the tables contain the same data.) You will then use the PXF `Hive` profile to query this partitioned Hive external table.

1. Create a Hive external table named `hive_multiformpart` that is partitioned by a string field named `year`:

   ```
   $ HADOOP_USER_NAME=hdfs hive
   ```

   ```
   hive> CREATE EXTERNAL TABLE hive_multiformpart( location string, month string,
   number_of_orders int, total_sales double)
           PARTITIONED BY( year string )
           ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
   ```

2. Describe the `sales_info` and `sales_info_rcfile` tables, noting the HDFS file `location` for each table:

   ```
   hive> DESCRIBE EXTENDED sales_info;
   hive> DESCRIBE EXTENDED sales_info_rcfile;
   ```

3. Create partitions in the `hive_multiformpart` table for the HDFS file locations associated with each of the `sales_info` and `sales_info_rcfile` tables:

   ```
   hive> ALTER TABLE hive_multiformpart ADD PARTITION (year = '2013') LOCATION 'hd
   fs://namenode:8020/apps/hive/warehouse/sales_info';
   hive> ALTER TABLE hive_multiformpart ADD PARTITION (year = '2016') LOCATION 'hd
   fs://namenode:8020/apps/hive/warehouse/sales_info_rcfile';
   ```

4. Explicitly identify the file format of the partition associated with the `sales_info_rcfile` table:

   ```
   hive> ALTER TABLE hive_multiformpart PARTITION (year='2016') SET FILEFORMAT RCF
   ILE;
   ```

   You need not specify the file format of the partition associated with the `sales_info` table, as `TEXTFILE` format is the default.

5. Query the `hive_multiformpart` table:

   ```
   hive> SELECT * from hive_multiformpart;
   ...
   Bangalore   Jul 271 8320.55 2016
   Beijing Dec 100 4248.41 2016
   Prague  Jan 101 4875.33 2013
   Rome    Mar 87  1557.39 2013
   ...
   hive> SELECT * from hive_multiformpart WHERE year='2013';
   hive> SELECT * from hive_multiformpart WHERE year='2016';
   ```

6. Show the partitions defined for the `hive_multiformpart` table and exit `hive`:

```
hive> SHOW PARTITIONS hive_multiformpart;
year=2013
year=2016
hive> quit;
```

7. Start the `psql` subsystem:

```
$ psql -d postgres
```

8. Use the PXF `Hive` profile to create a readable Greenplum Database external table that references the Hive `hive_multiformpart` external table that you created in the previous steps:

```
postgres=# CREATE EXTERNAL TABLE pxf_multiformpart(location text, month text, n
umber_of_orders int, total_sales float8, year text)
            LOCATION ('pxf://default.hive_multiformpart?PROFILE=Hive')
          FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

9. Query the PXF external table:

```
postgres=# SELECT * FROM pxf_multiformpart;
```

```
   location    | month | number_of_orders | total_sales | year
---------------+-------+------------------+-------------+--------
 ....
 Prague        | Dec   |              333 |     9894.77 | 2013
 Bangalore     | Jul   |              271 |     8320.55 | 2013
 Beijing       | Dec   |              100 |     4248.41 | 2013
 Prague        | Jan   |              101 |     4875.33 | 2016
 Rome          | Mar   |               87 |     1557.39 | 2016
 Bangalore     | May   |              317 |     8936.99 | 2016
 ....
```

10. Perform a second query to calculate the total number of orders for the year 2013:

```
postgres=# SELECT sum(number_of_orders) FROM pxf_multiformpart WHERE month='Dec
' AND year='2013';
 sum
-----
 433
```

Using PXF with Hive Default Partitions

This topic describes a difference in query results between Hive and PXF queries when Hive tables use a default partition. When dynamic partitioning is enabled in Hive, a partitioned table may store data in a default partition. Hive creates a default partition when the value of a partitioning column does not match the defined type of the column (for example, when a NULL value is used for any partitioning column). In Hive, any query that includes a filter on a partition column *excludes* any data that is stored in the table's default partition.

Similar to Hive, PXF represents a table's partitioning columns as columns that are appended to the

end of the table. However, PXF translates any column value in a default partition to a NULL value. This means that a Greenplum Database query that includes an `IS NULL` filter on a partitioning column can return different results than the same Hive query.

Consider a Hive partitioned table that is created with the statement:

```
hive> CREATE TABLE sales (order_id bigint, order_amount float) PARTITIONED BY (xdate d
ate);
```

The table is loaded with five rows that contain the following data:

```
1.0    1900-01-01
2.2    1994-04-14
3.3    2011-03-31
4.5    NULL
5.0    2013-12-06
```

Inserting row 4 creates a Hive default partition, because the partition column `xdate` contains a null value.

In Hive, any query that filters on the partition column omits data in the default partition. For example, the following query returns no rows:

```
hive> SELECT * FROM sales WHERE xdate IS null;
```

However, if you map this Hive table to a PXF external table in Greenplum Database, all default partition values are translated into actual NULL values. In Greenplum Database, executing the same query against the PXF external table returns row 4 as the result, because the filter matches the NULL value.

Keep this behavior in mind when you execute `IS NULL` queries on Hive partitioned tables.

Reading HBase Table Data

Apache HBase is a distributed, versioned, non-relational database on Hadoop.

The PXF HBase connector reads data stored in an HBase table. The HBase connector supports filter pushdown.

This section describes how to use the PXF HBase connector.

Prerequisites

Before working with HBase table data, ensure that you have:

- Copied `$PXF_HOME/lib/pxf-hbase-*.jar` to each node in your HBase cluster, and that the location of this PXF JAR file is in the `$HBASE_CLASSPATH`. This configuration is required for the PXF HBase connector to support filter pushdown.
- Met the PXF Hadoop Prerequisites.

HBase Primer

This topic assumes that you have a basic understanding of the following HBase concepts:

- An HBase column includes two components: a column family and a column qualifier. These components are delimited by a colon `:` character, <column-family>:<column-qualifier>.
- An HBase row consists of a row key and one or more column values. A row key is a unique identifier for the table row.
- An HBase table is a multi-dimensional map comprised of one or more columns and rows of data. You specify the complete set of column families when you create an HBase table.
- An HBase cell is comprised of a row (column family, column qualifier, column value) and a timestamp. The column value and timestamp in a given cell represent a version of the value.

For detailed information about HBase, refer to the Apache HBase Reference Guide.

HBase Shell

The HBase shell is a subsystem similar to that of `psql`. To start the HBase shell:

```
$ hbase shell
<hbase output>
hbase(main):001:0>
```

The default HBase namespace is named `default`.

Example: Creating an HBase Table

Create a sample HBase table.

1. Create an HBase table named `order_info` in the `default` namespace. `order_info` has two column families: `product` and `shipping_info`:

   ```
   hbase(main):> create 'order_info', 'product', 'shipping_info'
   ```

2. The `order_info` `product` column family has qualifiers named `name` and `location`. The `shipping_info` column family has qualifiers named `state` and `zipcode`. Add some data to the `order_info` table:

   ```
   put 'order_info', '1', 'product:name', 'tennis racquet'
   put 'order_info', '1', 'product:location', 'out of stock'
   put 'order_info', '1', 'shipping_info:state', 'CA'
   put 'order_info', '1', 'shipping_info:zipcode', '12345'
   put 'order_info', '2', 'product:name', 'soccer ball'
   put 'order_info', '2', 'product:location', 'on floor'
   put 'order_info', '2', 'shipping_info:state', 'CO'
   put 'order_info', '2', 'shipping_info:zipcode', '56789'
   put 'order_info', '3', 'product:name', 'snorkel set'
   put 'order_info', '3', 'product:location', 'warehouse'
   put 'order_info', '3', 'shipping_info:state', 'OH'
   put 'order_info', '3', 'shipping_info:zipcode', '34567'
   ```

   You will access the `orders_info` HBase table directly via PXF in examples later in this topic.

3. Display the contents of the `order_info` table:

   ```
   hbase(main):> scan 'order_info'
   ROW     COLUMN+CELL
   ```

```
 1      column=product:location, timestamp=1499074825516, value=out of stock

 1      column=product:name, timestamp=1499074825491, value=tennis racquet

 1      column=shipping_info:state, timestamp=1499074825531, value=CA

 1      column=shipping_info:zipcode, timestamp=1499074825548, value=12345

 2      column=product:location, timestamp=1499074825573, value=on floor
 ...
3 row(s) in 0.0400 seconds
```

Querying External HBase Data

The PXF HBase connector supports a single profile named `HBase`.

Use the following syntax to create a Greenplum Database external table that references an HBase table:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<hbase-table-name>?PROFILE=HBase[&SERVER=<server_name>]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

HBase connector-specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE call are described below.

Keyword	Value
<hbase-table-name>	The name of the HBase table.
PROFILE	The `PROFILE` keyword must specify `HBase`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
FORMAT	The `FORMAT` clause must specify `'CUSTOM' (FORMATTER='pxfwritable_import')`.

Data Type Mapping

HBase is byte-based; it stores all data types as an array of bytes. To represent HBase data in Greenplum Database, select a data type for your Greenplum Database column that matches the underlying content of the HBase column qualifier values.

Note: PXF does not support complex HBase objects.

Column Mapping

You can create a Greenplum Database external table that references all, or a subset of, the column qualifiers defined in an HBase table. PXF supports direct or indirect mapping between a Greenplum Database table column and an HBase table column qualifier.

Direct Mapping

When you use direct mapping to map Greenplum Database external table column names to HBase

qualifiers, you specify column-family-qualified HBase qualifier names as quoted values. The PXF HBase connector passes these column names as-is to HBase as it reads the table data.

For example, to create a Greenplum Database external table accessing the following data:

- qualifier `name` in the column family named `product`
- qualifier `zipcode` in the column family named `shipping_info`

from the `order_info` HBase table that you created in Example: Creating an HBase Table, use this `CREATE EXTERNAL TABLE` syntax:

```
CREATE EXTERNAL TABLE orderinfo_hbase ("product:name" varchar, "shipping_info:zipcode"
 int)
    LOCATION ('pxf://order_info?PROFILE=HBase')
    FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

Indirect Mapping via Lookup Table

When you use indirect mapping to map Greenplum Database external table column names to HBase qualifiers, you specify the mapping in a lookup table that you create in HBase. The lookup table maps a <column-family>:<column-qualifier> to a column name alias that you specify when you create the Greenplum Database external table.

You must name the HBase PXF lookup table `pxflookup`. And you must define this table with a single column family named `mapping`. For example:

```
hbase(main):> create 'pxflookup', 'mapping'
```

While the direct mapping method is fast and intuitive, using indirect mapping allows you to create a shorter, character-based alias for the HBase <column-family>:<column-qualifier> name. This better reconciles HBase column qualifier names with Greenplum Database due to the following:

- HBase qualifier names can be very long. Greenplum Database has a 63 character limit on the size of the column name.
- HBase qualifier names can include binary or non-printable characters. Greenplum Database column names are character-based.

When populating the `pxflookup` HBase table, add rows to the table such that the:

- row key specifies the HBase table name
- `mapping` column family qualifier identifies the Greenplum Database column name, and the value identifies the HBase `<column-family>:<column-qualifier>` for which you are creating the alias.

For example, to use indirect mapping with the `order_info` table, add these entries to the `pxflookup` table:

```
hbase(main):> put 'pxflookup', 'order_info', 'mapping:pname', 'product:name'
hbase(main):> put 'pxflookup', 'order_info', 'mapping:zip', 'shipping_info:zipcode'
```

Then create a Greenplum Database external table using the following `CREATE EXTERNAL TABLE` syntax:

```
CREATE EXTERNAL TABLE orderinfo_map (pname varchar, zip int)
    LOCATION ('pxf://order_info?PROFILE=HBase')
    FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

Row Key

The HBase table row key is a unique identifier for the table row. PXF handles the row key in a special way.

To use the row key in the Greenplum Database external table query, define the external table using the PXF reserved column named `recordkey`. The `recordkey` column name instructs PXF to return the HBase table record key for each row.

Define the `recordkey` using the Greenplum Database data type `bytea`.

For example:

```
CREATE EXTERNAL TABLE <table_name> (recordkey bytea, ... )
    LOCATION ('pxf://<hbase_table_name>?PROFILE=HBase')
    FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

After you have created the external table, you can use the `recordkey` in a `WHERE` clause to filter the HBase table on a range of row key values.

Note: To enable filter pushdown on the `recordkey`, define the field as `text`.

Accessing Azure, Google Cloud Storage, Minio, and S3 Object Stores with PXF

PXF is installed with connectors to Azure Blob Storage, Azure Data Lake, Google Cloud Storage, Minio, and S3 object stores.

Prerequisites

Before working with object store data using PXF, ensure that:

- You have configured and initialized PXF, and PXF is running on each Greenplum Database segment host. See Configuring PXF for additional information.
- You have configured the PXF Object Store Connectors that you plan to use. Refer to Configuring Connectors to Azure and Google Cloud Storage Object Stores and Configuring Connectors to Minio and S3 Object Stores for instructions.
- Time is synchronized between the Greenplum Database segment hosts and the external object store systems.

Connectors, Data Formats, and Profiles

The PXF object store connectors provide built-in profiles to support the following data formats:

- Text
- Avro
- JSON
- Parquet
- AvroSequenceFile
- SequenceFile

The PXF connectors to Azure expose the following profiles to read, and in many cases write, these supported data formats:

Data Format	Azure Blob Storage	Azure Data Lake
delimited single line plain text	wasbs:text	adl:text
delimited text with quoted linefeeds	wasbs:text:multi	adl:text:multi
Avro	wasbs:avro	adl:avro
JSON	wasbs:json	adl:json
Parquet	wasbs:parquet	adl:parquet

Data Format	Azure Blob Storage	Azure Data Lake
AvroSequenceFile	wasbs:AvroSequenceFile	adl:AvroSequenceFile
SequenceFile	wasbs:SequenceFile	adl:SequenceFile

Similarly, the PXF connectors to Google Cloud Storage, Minio, and S3 expose these profiles:

Data Format	Google Cloud Storage	S3 or Minio
delimited single line plain text	gs:text	s3:text
delimited text with quoted linefeeds	gs:text:multi	s3:text:multi
Avro	gs:avro	s3:avro
JSON	gs:json	s3:json
Parquet	gs:parquet	s3:parquet
AvroSequenceFile	gs:AvroSequenceFile	s3:AvroSequenceFile
SequenceFile	gs:SequenceFile	s3:SequenceFile

You provide the profile name when you specify the `pxf` protocol on a `CREATE EXTERNAL TABLE` command to create a Greenplum Database external table that references a file or directory in the specific object store.

Sample CREATE EXTERNAL TABLE Commands

When you create an external table that references a file or directory in an object store, you must specify a `SERVER` in the `LOCATION` URI.

The following command creates an external table that references a text file on S3. It specifies the profile named `s3:text` and the server configuration named `s3srvcfg`:

```
CREATE EXTERNAL TABLE pxf_s3_text(location text, month text, num_orders int, total_sal
es float8)
  LOCATION ('pxf://S3_BUCKET/pxf_examples/pxf_s3_simple.txt?PROFILE=s3:text&SERVER=s3srv
cfg')
FORMAT 'TEXT' (delimiter=E',');
```

The following command creates an external table that references a text file on Azure Blob Storage. It specifies the profile named `wasbs:text` and the server configuration named `wasbssrvcfg`. You would provide the Azure Blob Storage container identifier and your Azure Blob Storage account name.

```
CREATE EXTERNAL TABLE pxf_wasbs_text(location text, month text, num_orders int, total_
sales float8)
  LOCATION ('pxf://AZURE_CONTAINER@YOUR_AZURE_BLOB_STORAGE_ACCOUNT_NAME.blob.co
re.windows.net/path/to/blob/file?PROFILE=wasbs:text&SERVER=wasbssrvcfg')
FORMAT 'TEXT';
```

The following command creates an external table that references a text file on Azure Data Lake. It specifies the profile named `adl:text` and the server configuration named `adlsrvcfg`. You would provide your Azure Data Lake account name.

```
CREATE EXTERNAL TABLE pxf_adl_text(location text, month text, num_orders int, total_sa
```

```
les float8)
  LOCATION ('pxf://YOUR_ADL_ACCOUNT_NAME.azuredatalakestore.net/path/to/file?PROFIL
E=adl:text&SERVER=adlsrvcfg')
FORMAT 'TEXT';
```

The following command creates an external table that references a JSON file on Google Cloud Storage. It specifies the profile named `gs:json` and the server configuration named `gcssrvcfg`:

```
CREATE EXTERNAL TABLE pxf_gsc_json(location text, month text, num_orders int, total_sa
les float8)
  LOCATION ('pxf://dir/subdir/file.json?PROFILE=gs:json&SERVER=gcssrvcfg')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

About Accessing the S3 Object Store

PXF is installed with a connector to the S3 object store. PXF supports the following additional runtime features with this connector:

- Overriding the S3 credentials specified in the server configuration by providing them in the `CREATE EXTERNAL TABLE` command DDL.
- Using the Amazon S3 Select service to read certain CSV and Parquet data from S3.

Overriding the S3 Server Configuration with DDL

If you are accessing an S3-compatible object store, you can override the credentials in an S3 server configuration by directly specifying the S3 access ID and secret key via these custom options in the `CREATE EXTERNAL TABLE LOCATION` clause:

Custom Option	Value Description
accesskey	The AWS account access key ID.
secretkey	The secret key associated with the AWS access key ID.

For example:

```
CREATE EXTERNAL TABLE pxf_ext_tbl(name text, orders int)
  LOCATION ('pxf://S3_BUCKET/dir/file.txt?PROFILE=s3:text&SERVER=s3srvcfg&accesskey=YOU
RKEY&secretkey=YOURSECRET')
FORMAT 'TEXT' (delimiter=E',');
```

Credentials that you provide in this manner are visible as part of the external table definition. Do not use this method of passing credentials in a production environment.

PXF does not support overriding Azure, Google Cloud Storage, and Minio server credentials in this manner at this time.

Refer to Configuration Property Precedence for detailed information about the precedence rules that PXF uses to obtain configuration property settings for a Greenplum Database user.

Using the Amazon S3 Select Service

Refer to Reading CSV and Parquet Data from S3 Using S3 Select for specific information on how

PXF can use the Amazon S3 Select service to read CSV and Parquet files stored on S3.

Reading and Writing Text Data in an Object Store

The PXF object store connectors support plain delimited and comma-separated value format text data. This section describes how to use PXF to access text data in an object store, including how to create, query, and insert data into an external table that references files in the object store.

Note: Accessing text data from an object store is very similar to accessing text data in HDFS.

Prerequisites

Ensure that you have met the PXF Object Store Prerequisites before you attempt to read data from or write data to an object store.

Reading Text Data

Use the `<objstore>:text` profile when you read plain text delimited or .csv data from an object store where each row is a single record. PXF supports the following `<objstore>` profile prefixes:

Object Store	Profile Prefix
Azure Blob Storage	wasbs
Azure Data Lake	adl
Google Cloud Storage	gs
Minio	s3
S3	s3

The following syntax creates a Greenplum Database readable external table that references a simple text file in an object store:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-file>?PROFILE=<objstore>:text&SERVER=<server_name>[&IGNORE_M
ISSING_PATH=<boolean>][&<custom-option>=<value>[...]]')
FORMAT '[TEXT|CSV]' (delimiter[=|<space>][E]'<delim_value>');
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-file>	The absolute path to the directory or file in the S3 object store.
PROFILE=<objstore>:text	The `PROFILE` keyword must identify the specific object store. For example, `s3:text`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data.
IGNORE_MISSING_PATH=<boolean>	Specify the action to take when <path-to-file> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.

Keyword	Value
FORMAT	Use `FORMAT 'TEXT'` when <path-to-file> references plain text delimited data. Use `FORMAT 'CSV'` when <path-to-file> references comma-separated value data.
delimiter	The delimiter character in the data. For `FORMAT 'CSV'`, the default <delim_value> is a comma `,`. Preface the <delim_value> with an `E` when the value is an escape sequence. Examples: `(delimiter=E'\t')`, `(delimiter ':')`.

If you are accessing an S3 object store:

- You can provide S3 credentials via custom options in the `CREATE EXTERNAL TABLE` command as described in Overriding the S3 Server Configuration with DDL.
- If you are reading CSV-format data from S3, you can direct PXF to use the S3 Select Amazon service to retrieve the data. Refer to Using the Amazon S3 Select Service for more information about the PXF custom option used for this purpose.

Example: Reading Text Data from S3

Perform the following procedure to create a sample text file, copy the file to S3, and use the `s3:text` profile to create two PXF external tables to query the data.

To run this example, you must:

- Have the AWS CLI tools installed on your system
- Know your AWS access ID and secret key
- Have write permission to an S3 bucket

1. Create a directory in S3 for PXF example data files. For example, if you have write access to an S3 bucket named `BUCKET`:

   ```
   $ aws s3 mb s3://BUCKET/pxf_examples
   ```

2. Locally create a delimited plain text data file named `pxf_s3_simple.txt`:

   ```
   $ echo 'Prague,Jan,101,4875.33
   Rome,Mar,87,1557.39
   Bangalore,May,317,8936.99
   Beijing,Jul,411,11600.67' > /tmp/pxf_s3_simple.txt
   ```

   Note the use of the comma `,` to separate the four data fields.

3. Copy the data file to the S3 directory you created in Step 1:

   ```
   $ aws s3 cp /tmp/pxf_s3_simple.txt s3://BUCKET/pxf_examples/
   ```

4. Verify that the file now resides in S3:

   ```
   $ aws s3 ls s3://BUCKET/pxf_examples/pxf_s3_simple.txt
   ```

5. Start the `psql` subsystem:

   ```
   $ psql -d postgres
   ```

6. Use the PXF `s3:text` profile to create a Greenplum Database external table that references the `pxf_s3_simple.txt` file that you just created and added to S3. For example, if your server name is `s3srvcfg`:

```
postgres=# CREATE EXTERNAL TABLE pxf_s3_textsimple(location text, month text, n
um_orders int, total_sales float8)
            LOCATION ('pxf://BUCKET/pxf_examples/pxf_s3_simple.txt?PROFILE=s3:t
ext&SERVER=s3srvcfg')
          FORMAT 'TEXT' (delimiter=E',');
```

7. Query the external table:

```
postgres=# SELECT * FROM pxf_s3_textsimple;
```

```
   location    | month | num_orders | total_sales
---------------+-------+------------+-------------
 Prague        | Jan   |        101 |     4875.33
 Rome          | Mar   |         87 |     1557.39
 Bangalore     | May   |        317 |     8936.99
 Beijing       | Jul   |        411 |    11600.67
(4 rows)
```

8. Create a second external table that references `pxf_s3_simple.txt`, this time specifying the `CSV FORMAT`:

```
postgres=# CREATE EXTERNAL TABLE pxf_s3_textsimple_csv(location text, month tex
t, num_orders int, total_sales float8)
            LOCATION ('pxf://BUCKET/pxf_examples/pxf_s3_simple.txt?PROFILE=s3:t
ext&SERVER=s3srvcfg')
          FORMAT 'CSV';
postgres=# SELECT * FROM pxf_s3_textsimple_csv;
```

When you specify `FORMAT 'CSV'` for comma-separated value data, no `delimiter` formatter option is required because comma is the default delimiter value.

Reading Text Data with Quoted Linefeeds

Use the `<objstore>:text:multi` profile to read plain text data with delimited single- or multi- line records that include embedded (quoted) linefeed characters. The following syntax creates a Greenplum Database readable external table that references such a text file in an object store:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-file>?PROFILE=<objstore>:text:multi&SERVER=<server_name>[&IG
NORE_MISSING_PATH=<boolean>][&<custom-option>=<value>[...]]')
FORMAT '[TEXT|CSV]' (delimiter[=|<space>][E]'<delim_value>');
```

The specific keywords and values used in the CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-file>	The absolute path to the directory or file in the S3 data store.

Keyword	Value
PROFILE=<objstore>:text:multi	The `PROFILE` keyword must identify the specific object store. For example, `s3:text:multi`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data.
IGNORE_MISSING_PATH=<boolean>	Specify the action to take when <path-to-file> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.
FORMAT	Use `FORMAT 'TEXT'` when <path-to-file> references plain text delimited data. Use `FORMAT 'CSV'` when <path-to-file> references comma-separated value data.
delimiter	The delimiter character in the data. For `FORMAT 'CSV'`, the default <delim_value> is a comma `,`. Preface the <delim_value> with an `E` when the value is an escape sequence. Examples: `(delimiter=E'\t')`, `(delimiter ':')`.

If you are accessing an S3 object store, you can provide S3 credentials via custom options in the `CREATE EXTERNAL TABLE` command as described in Overriding the S3 Server Configuration with DDL.

Example: Reading Multi-Line Text Data from S3

Perform the following steps to create a sample text file, copy the file to S3, and use the PXF `s3:text:multi` profile to create a Greenplum Database readable external table to query the data.

To run this example, you must:

- Have the AWS CLI tools installed on your system
- Know your AWS access ID and secret key
- Have write permission to an S3 bucket

1. Create a second delimited plain text file:

   ```
   $ vi /tmp/pxf_s3_multi.txt
   ```

2. Copy/paste the following data into `pxf_s3_multi.txt`:

   ```
   "4627 Star Rd.
   San Francisco, CA  94107":Sept:2017
   "113 Moon St.
   San Diego, CA  92093":Jan:2018
   "51 Belt Ct.
   Denver, CO  90123":Dec:2016
   "93114 Radial Rd.
   Chicago, IL  60605":Jul:2017
   "7301 Brookview Ave.
   Columbus, OH  43213":Dec:2018
   ```

   Notice the use of the colon `:` to separate the three fields. Also notice the quotes around the first (address) field. This field includes an embedded line feed separating the street address from the city and state.

3. Copy the text file to S3:

   ```
   $ aws s3 cp /tmp/pxf_s3_multi.txt s3://BUCKET/pxf_examples/
   ```

4. Use the `s3:text:multi` profile to create an external table that references the `pxf_s3_multi.txt` S3 file, making sure to identify the `:` (colon) as the field separator. For example, if your server name is `s3srvcfg`:

```
postgres=# CREATE EXTERNAL TABLE pxf_s3_textmulti(address text, month text, yea
r int)
            LOCATION ('pxf://BUCKET/pxf_examples/pxf_s3_multi.txt?PROFILE=s3:te
xt:multi&SERVER=s3srvcfg')
          FORMAT 'CSV' (delimiter ':');
```

   Notice the alternate syntax for specifying the `delimiter`.

5. Query the `pxf_s3_textmulti` table:

```
postgres=# SELECT * FROM pxf_s3_textmulti;
```

```
         address          | month | year
--------------------------+-------+------
 4627 Star Rd.            | Sept  | 2017
 San Francisco, CA  94107
 113 Moon St.             | Jan   | 2018
 San Diego, CA  92093
 51 Belt Ct.              | Dec   | 2016
 Denver, CO  90123
 93114 Radial Rd.         | Jul   | 2017
 Chicago, IL  60605
 7301 Brookview Ave.      | Dec   | 2018
 Columbus, OH  43213
(5 rows)
```

Writing Text Data

The “<objstore>:text” profiles support writing single line plain text data to an object store. When you create a writable external table with PXF, you specify the name of a directory. When you insert records into a writable external table, the block(s) of data that you insert are written to one or more files in the directory that you specified.

Note: External tables that you create with a writable profile can only be used for `INSERT` operations. If you want to query the data that you inserted, you must create a separate readable external table that references the directory.

Use the following syntax to create a Greenplum Database writable external table that references an object store directory:

```
CREATE WRITABLE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-dir>
    ?PROFILE=<objstore>:text&SERVER=<server_name>[&<custom-option>=<value>[...]]')
FORMAT '[TEXT|CSV]' (delimiter[=|<space>][E]'<delim_value>');
[DISTRIBUTED BY (<column_name> [, ... ] ) | DISTRIBUTED RANDOMLY];
```

The specific keywords and values used in the CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-dir>	The absolute path to the directory in the S3 data store.
PROFILE=<objstore>:text	The `PROFILE` keyword must identify the specific object store. For example, `s3:text`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data.
<custom-option>=<value>	<custom-option>s are described below.
FORMAT	Use `FORMAT 'TEXT'` to write plain, delimited text to <path-to-dir>. Use `FORMAT 'CSV'` to write comma-separated value text to <path-to-dir>.
delimiter	The delimiter character in the data. For `FORMAT 'CSV'`, the default <delim_value> is a comma `,`. Preface the <delim_value> with an `E` when the value is an escape sequence. Examples: `(delimiter=E'\t')`, `(delimiter ':')`.
DISTRIBUTED BY	If you want to load data from an existing Greenplum Database table into the writable external table, consider specifying the same distribution policy or <column_name> on both tables. Doing so will avoid extra motion of data between segments on the load operation.

Writable external tables that you create using an `<objstore>:text` profile can optionally use record or block compression. The PXF `<objstore>:text` profiles support the following compression codecs:

- org.apache.hadoop.io.compress.DefaultCodec
- org.apache.hadoop.io.compress.GzipCodec
- org.apache.hadoop.io.compress.BZip2Codec

You specify the compression codec via custom options in the `CREATE EXTERNAL TABLE LOCATION` clause. The `<objstore>:text` profile support the following custom write options:

Option	Value Description

COMPRESSION_CODEC	The compression codec Java class name. If this option is not provided, Greenplum Database performs no data compression. Supported compression codecs include:
	`org.apache.hadoop.io.compress.DefaultCodec`
	`org.apache.hadoop.io.compress.BZip2Codec`
	`org.apache.hadoop.io.compress.GzipCodec`

COMPRESSION_TYPE	The compression type to employ; supported values are `RECORD` (the default) or `BLOCK`.
THREAD-SAFE	Boolean value determining if a table query can run in multi-threaded mode. The default value is `TRUE`. Set this option to `FALSE` to handle all requests in a single thread for operations that are not thread-safe (for example, compression).

If you are accessing an S3 object store, you can provide S3 credentials via custom options in the `CREATE EXTERNAL TABLE` command as described in Overriding the S3 Server Configuration with DDL.

Example: Writing Text Data to S3

This example utilizes the data schema introduced in Example: Reading Text Data from S3.

Column Name	Data Type
location	text

Column Name	Data Type
month	text
number_of_orders	int
total_sales	float8

This example also optionally uses the Greenplum Database external table named `pxf_s3_textsimple` that you created in that exercise.

Procedure

Perform the following procedure to create Greenplum Database writable external tables utilizing the same data schema as described above, one of which will employ compression. You will use the PXF `s3:text` profile to write data to S3. You will also create a separate, readable external table to read the data that you wrote to S3.

1. Create a Greenplum Database writable external table utilizing the data schema described above. Write to the S3 directory `BUCKET/pxf_examples/pxfwrite_s3_textsimple1`. Create the table specifying a comma `,` as the delimiter. For example, if your server name is `s3srvcfg`:

   ```
   postgres=# CREATE WRITABLE EXTERNAL TABLE pxf_s3_writetbl_1(location text, mont
   h text, num_orders int, total_sales float8)
               LOCATION ('pxf://BUCKET/pxf_examples/pxfwrite_s3_textsimple1?PROFIL
   E=s3:text&SERVER=s3srvcfg')
             FORMAT 'TEXT' (delimiter=',');
   ```

   You specify the `FORMAT` subclause `delimiter` value as the single ascii comma character `,`.

2. Write a few individual records to the `pxfwrite_s3_textsimple1` S3 directory by invoking the SQL `INSERT` command on `pxf_s3_writetbl_1`:

   ```
   postgres=# INSERT INTO pxf_s3_writetbl_1 VALUES ( 'Frankfurt', 'Mar', 777, 3956
   .98 );
   postgres=# INSERT INTO pxf_s3_writetbl_1 VALUES ( 'Cleveland', 'Oct', 3812, 966
   45.37 );
   ```

3. (Optional) Insert the data from the `pxf_s3_textsimple` table that you created in Example: Reading Text Data from S3 into `pxf_s3_writetbl_1`:

   ```
   postgres=# INSERT INTO pxf_s3_writetbl_1 SELECT * FROM pxf_s3_textsimple;
   ```

4. Greenplum Database does not support directly querying a writable external table. To query the data that you just added to S3, you must create a readable external Greenplum Database table that references the S3 directory:

   ```
   postgres=# CREATE EXTERNAL TABLE pxf_s3_textsimple_r1(location text, month text
   , num_orders int, total_sales float8)
               LOCATION ('pxf://BUCKET/pxf_examples/pxfwrite_s3_textsimple1?PROFIL
   E=s3:text&SERVER=s3srvcfg')
               FORMAT 'CSV';
   ```

   You specify the `'CSV' FORMAT` when you create the readable external table because you

created the writable table with a comma `,` as the delimiter character, the default delimiter for `'CSV' FORMAT`.

5. Query the readable external table:

```
postgres=# SELECT * FROM pxf_s3_textsimple_r1 ORDER BY total_sales;
```

```
 location  | month | num_orders | total_sales
-----------+-------+------------+-------------
 Rome      | Mar   |         87 |     1557.39
 Frankfurt | Mar   |        777 |     3956.98
 Prague    | Jan   |        101 |     4875.33
 Bangalore | May   |        317 |     8936.99
 Beijing   | Jul   |        411 |    11600.67
 Cleveland | Oct   |       3812 |    96645.37
(6 rows)
```

   The `pxf_s3_textsimple_r1` table includes the records you individually inserted, as well as the full contents of the `pxf_s3_textsimple` table if you performed the optional step.

6. Create a second Greenplum Database writable external table, this time using Gzip compression and employing a colon `:` as the delimiter:

```
postgres=# CREATE WRITABLE EXTERNAL TABLE pxf_s3_writetbl_2 (location text, mon
th text, num_orders int, total_sales float8)
            LOCATION ('pxf://BUCKET/pxf_examples/pxfwrite_s3_textsimple2?PROFIL
E=s3:text&SERVER=s3srvcfg&COMPRESSION_CODEC=org.apache.hadoop.io.compress.GzipC
odec')
          FORMAT 'TEXT' (delimiter=':');
```

7. Write a few records to the `pxfwrite_s3_textsimple2` S3 directory by inserting directly into the `pxf_s3_writetbl_2` table:

```
gpadmin=# INSERT INTO pxf_s3_writetbl_2 VALUES ( 'Frankfurt', 'Mar', 777, 3956.
98 );
gpadmin=# INSERT INTO pxf_s3_writetbl_2 VALUES ( 'Cleveland', 'Oct', 3812, 9664
5.37 );
```

8. To query data from the newly-created S3 directory named `pxfwrite_s3_textsimple2`, you can create a readable external Greenplum Database table as described above that references this S3 directory and specifies `FORMAT 'CSV' (delimiter=':')`.

Reading and Writing Avro Data in an Object Store

The PXF object store connectors support reading Avro-format data. This section describes how to use PXF to read and write Avro data in an object store, including how to create, query, and insert into an external table that references an Avro file in the store.

Note: Accessing Avro-format data from an object store is very similar to accessing Avro-format data in HDFS. This topic identifies object store-specific information required to read Avro data, and links to the PXF HDFS Avro documentation where appropriate for common information.

Prerequisites

Ensure that you have met the PXF Object Store Prerequisites before you attempt to read data from an object store.

Working with Avro Data

Refer to Working with Avro Data in the PXF HDFS Avro documentation for a description of the Apache Avro data serialization framework.

When you read or write Avro data in an object store:

- If the Avro schema file resides in the object store:
    - You must include the bucket in the schema file path. This bucket need not specify the same bucket as the Avro data file.
    - The secrets that you specify in the `SERVER` configuration must provide access to both the data file and schema file buckets.
- The schema file path must not include spaces.

Creating the External Table

Use the `<objstore>:avro` profiles to read and write Avro-format files in an object store. PXF supports the following `<objstore>` profile prefixes:

Object Store	Profile Prefix
Azure Blob Storage	wasbs
Azure Data Lake	adl
Google Cloud Storage	gs
Minio	s3
S3	s3

The following syntax creates a Greenplum Database external table that references an Avro-format file:

```
CREATE [WRITABLE] EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-file>?PROFILE=<objstore>:avro&SERVER=<server_name>[&<custom-option>=<value>[...]]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import'|'pxfwritable_export');
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-file>	The absolute path to the directory or file in the object store.
PROFILE=<objstore>:avro	The `PROFILE` keyword must identify the specific object store. For example, `s3:avro`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data.

Keyword	Value
<custom-option>=<value>	Avro-specific custom options are described in the PXF HDFS Avro documentation.
FORMAT 'CUSTOM'	Use `FORMAT 'CUSTOM'` with `(FORMATTER='pxfwritable_export')` (write) or `(FORMATTER='pxfwritable_import')` (read).

If you are accessing an S3 object store, you can provide S3 credentials via custom options in the `CREATE EXTERNAL TABLE` command as described in Overriding the S3 Server Configuration with DDL.

Example

Refer to Example: Reading Avro Data in the PXF HDFS Avro documentation for an Avro example. Modifications that you must make to run the example with an object store include:

- Copying the file to the object store instead of HDFS. For example, to copy the file to S3:

```
$ aws s3 cp /tmp/pxf_avro.avro s3://BUCKET/pxf_examples/
```

- Using the `CREATE EXTERNAL TABLE` syntax and `LOCATION` keywords and settings described above. For example, if your server name is `s3srvcfg`:

```
CREATE EXTERNAL TABLE pxf_s3_avro(id bigint, username text, followers text, fma
p text, relationship text, address text)
  LOCATION ('pxf://BUCKET/pxf_examples/pxf_avro.avro?PROFILE=s3:avro&SERVER=s3s
rvcfg&COLLECTION_DELIM=,&MAPKEY_DELIM=:&RECORDKEY_DELIM=:')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

You make similar modifications to follow the steps in Example: Writing Avro Data.

Reading JSON Data from an Object Store

The PXF object store connectors support reading JSON-format data. This section describes how to use PXF to access JSON data in an object store, including how to create and query an external table that references a JSON file in the store.

Note: Accessing JSON-format data from an object store is very similar to accessing JSON-format data in HDFS. This topic identifies object store-specific information required to read JSON data, and links to the PXF HDFS JSON documentation where appropriate for common information.

Prerequisites

Ensure that you have met the PXF Object Store Prerequisites before you attempt to read data from an object store.

Working with Avro Data

Refer to Working with JSON Data in the PXF HDFS JSON documentation for a description of the JSON text-based data-interchange format.

Creating the External Table

Use the `<objstore>:json` profile to read JSON-format files from an object store. PXF supports the following `<objstore>` profile prefixes:

Object Store	Profile Prefix
Azure Blob Storage	wasbs
Azure Data Lake	adl
Google Cloud Storage	gs
Minio	s3
S3	s3

The following syntax creates a Greenplum Database readable external table that references a JSON-format file:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-file>?PROFILE=<objstore>:json&SERVER=<server_name>[&<custom-option>=<value>[...]]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-file>	The absolute path to the directory or file in the object store.
PROFILE=<objstore>:json	The `PROFILE` keyword must identify the specific object store. For example, `s3:json`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data.
<custom-option>=<value>	JSON supports the custom option named `IDENTIFIER` as described in the PXF HDFS JSON documentation.
FORMAT 'CUSTOM'	Use `FORMAT 'CUSTOM'` with the `<objstore>:json` profile. The `CUSTOM FORMAT` requires that you specify `(FORMATTER='pxfwritable_import')`.

If you are accessing an S3 object store, you can provide S3 credentials via custom options in the `CREATE EXTERNAL TABLE` command as described in Overriding the S3 Server Configuration with DDL.

Example

Refer to Loading the Sample JSON Data to HDFS and Example: Reading a JSON File with Single Line Records in the PXF HDFS JSON documentation for a JSON example. Modifications that you must make to run the example with an object store include:

- Copying the file to the object store instead of HDFS. For example, to copy the file to S3:

  ```
  $ aws s3 cp /tmp/singleline.json s3://BUCKET/pxf_examples/
  $ aws s3 cp /tmp/multiline.json s3://BUCKET/pxf_examples/
  ```

- Using the `CREATE EXTERNAL TABLE` syntax and `LOCATION` keywords and settings described

above. For example, if your server name is `s3srvcfg`:

```
CREATE EXTERNAL TABLE singleline_json_s3(
  created_at TEXT,
  id_str TEXT,
  "user.id" INTEGER,
  "user.location" TEXT,
  "coordinates.values[0]" INTEGER,
  "coordinates.values[1]" INTEGER
)
  LOCATION('pxf://BUCKET/pxf_examples/singleline.json?PROFILE=s3:json&SERVER=s3
srvcfg')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

Reading and Writing Parquet Data in an Object Store

The PXF object store connectors support reading and writing Parquet-format data. This section describes how to use PXF to access Parquet-format data in an object store, including how to create and query external tables that references a Parquet file in the store.

Note: Accessing Parquet-format data from an object store is very similar to accessing Parquet-format data in HDFS. This topic identifies object store-specific information required to read and write Parquet data, and links to the PXF HDFS Parquet documentation where appropriate for common information.

Prerequisites

Ensure that you have met the PXF Object Store Prerequisites before you attempt to read data from or write data to an object store.

Data Type Mapping

Refer to Data Type Mapping in the PXF HDFS Parquet documentation for a description of the mapping between Greenplum Database and Parquet data types.

Creating the External Table

The PXF `<objstore>:parquet` profiles support reading and writing data in Parquet-format. PXF supports the following `<objstore>` profile prefixes:

Object Store	Profile Prefix
Azure Blob Storage	wasbs
Azure Data Lake	adl
Google Cloud Storage	gs
Minio	s3
S3	s3

Use the following syntax to create a Greenplum Database external table that references an HDFS directory. When you insert records into a writable external table, the block(s) of data that you insert

are written to one or more files in the directory that you specified.

```
CREATE [WRITABLE] EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-dir>
    ?PROFILE=<objstore>:parquet&SERVER=<server_name>[&<custom-option>=<value>[...]]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import'|'pxfwritable_export');
[DISTRIBUTED BY (<column_name> [, ... ] ) | DISTRIBUTED RANDOMLY];
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-dir>	The absolute path to the directory in the object store.
PROFILE=<objstore>:parquet	The `PROFILE` keyword must identify the specific object store. For example, `s3:parquet`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data.
<custom-option>=<value>	Parquet-specific custom options are described in the PXF HDFS Parquet documentation.
FORMAT 'CUSTOM'	Use `FORMAT 'CUSTOM'` with `(FORMATTER='pxfwritable_export')` (write) or `(FORMATTER='pxfwritable_import')` (read).
DISTRIBUTED BY	If you want to load data from an existing Greenplum Database table into the writable external table, consider specifying the same distribution policy or <column_name> on both tables. Doing so will avoid extra motion of data between segments on the load operation.

If you are accessing an S3 object store:

- You can provide S3 credentials via custom options in the `CREATE EXTERNAL TABLE` command as described in Overriding the S3 Server Configuration with DDL.
- If you are reading Parquet data from S3, you can direct PXF to use the S3 Select Amazon service to retrieve the data. Refer to Using the Amazon S3 Select Service for more information about the PXF custom option used for this purpose.

Example

Refer to the Example in the PXF HDFS Parquet documentation for a Parquet write/read example. Modifications that you must make to run the example with an object store include:

- Using the `CREATE WRITABLE EXTERNAL TABLE` syntax and `LOCATION` keywords and settings described above for the writable external table. For example, if your server name is `s3srvcfg`:

  ```
  CREATE WRITABLE EXTERNAL TABLE pxf_tbl_parquet_s3 (location text, month text, n
  umber_of_orders int, total_sales double precision)
    LOCATION ('pxf://BUCKET/pxf_examples/pxf_parquet?PROFILE=s3:parquet&SERVER=s3
  srvcfg')
  FORMAT 'CUSTOM' (FORMATTER='pxfwritable_export');
  ```

- Using the `CREATE EXTERNAL TABLE` syntax and `LOCATION` keywords and settings described above for the readable external table. For example, if your server name is `s3srvcfg`:

```
CREATE EXTERNAL TABLE read_pxf_parquet_s3(location text, month text, number_of_
orders int, total_sales double precision)
  LOCATION ('pxf://BUCKET/pxf_examples/pxf_parquet?PROFILE=s3:parquet&SERVER=s3
srvcfg')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

Reading and Writing SequenceFile Data in an Object Store

The PXF object store connectors support SequenceFile format binary data. This section describes how to use PXF to read and write SequenceFile data, including how to create, insert, and query data in external tables that reference files in an object store.

Note: Accessing SequenceFile-format data from an object store is very similar to accessing SequenceFile-format data in HDFS. This topic identifies object store-specific information required to read and write SequenceFile data, and links to the PXF HDFS SequenceFile documentation where appropriate for common information.

Prerequisites

Ensure that you have met the PXF Object Store Prerequisites before you attempt to read data from or write data to an object store.

Creating the External Table

The PXF `<objstore>:SequenceFile` profiles support reading and writing binary data in SequenceFile-format. PXF supports the following `<objstore>` profile prefixes:

Object Store	Profile Prefix
Azure Blob Storage	wasbs
Azure Data Lake	adl
Google Cloud Storage	gs
Minio	s3
S3	s3

Use the following syntax to create a Greenplum Database external table that references an HDFS directory. When you insert records into a writable external table, the block(s) of data that you insert are written to one or more files in the directory that you specified.

```
CREATE [WRITABLE] EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-dir>
    ?PROFILE=<objstore>:SequenceFile&SERVER=<server_name>[&<custom-option>=<value>[...
]]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import'|'pxfwritable_export')
[DISTRIBUTED BY (<column_name> [, ... ] ) | DISTRIBUTED RANDOMLY];
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-dir>	The absolute path to the directory in the object store.
PROFILE=<objstore>:SequenceFile	The `PROFILE` keyword must identify the specific object store. For example, `s3:SequenceFile`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data.
<custom-option>=<value>	SequenceFile-specific custom options are described in the PXF HDFS SequenceFile documentation.
FORMAT 'CUSTOM'	Use `FORMAT 'CUSTOM'` with `(FORMATTER='pxfwritable_export')` (write) or `(FORMATTER='pxfwritable_import')` (read).
DISTRIBUTED BY	If you want to load data from an existing Greenplum Database table into the writable external table, consider specifying the same distribution policy or <column_name> on both tables. Doing so will avoid extra motion of data between segments on the load operation.

If you are accessing an S3 object store, you can provide S3 credentials via custom options in the `CREATE EXTERNAL TABLE` command as described in Overriding the S3 Server Configuration with DDL.

Example

Refer to Example: Writing Binary Data to HDFS in the PXF HDFS SequenceFile documentation for a write/read example. Modifications that you must make to run the example with an object store include:

- Using the `CREATE EXTERNAL TABLE` syntax and `LOCATION` keywords and settings described above for the writable external table. For example, if your server name is `s3srvcfg`:

```
CREATE WRITABLE EXTERNAL TABLE pxf_tbl_seqfile_s3(location text, month text, nu
mber_of_orders integer, total_sales real)
  LOCATION ('pxf://BUCKET/pxf_examples/pxf_seqfile?PROFILE=s3:SequenceFile&DATA
-SCHEMA=com.example.pxf.hdfs.writable.dataschema.PxfExample_CustomWritable&COMP
RESSION_TYPE=BLOCK&COMPRESSION_CODEC=org.apache.hadoop.io.compress.BZip2Codec&S
ERVER=s3srvcfg')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_export');
```

- Using the `CREATE EXTERNAL TABLE` syntax and `LOCATION` keywords and settings described above for the readable external table. For example, if your server name is `s3srvcfg`:

```
CREATE EXTERNAL TABLE read_pxf_tbl_seqfile_s3(location text, month text, number
_of_orders integer, total_sales real)
  LOCATION ('pxf://BUCKET/pxf_examples/pxf_seqfile?PROFILE=s3:SequenceFile&DATA
-SCHEMA=com.example.pxf.hdfs.writable.dataschema.PxfExample_CustomWritable&SERV
ER=s3srvcfg')
 FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

Reading a Multi-Line Text File into a Single Table Row

The PXF object store connectors support reading a multi-line text file as a single table row. This section describes how to use PXF to read multi-line text and JSON data files in an object store, including how to create an external table that references multiple files in the store.

PXF supports reading only text and JSON files in this manner.

Note: Accessing multi-line files from an object store is very similar to accessing multi-line files in HDFS. This topic identifies the object store-specific information required to read these files. Refer to the PXF HDFS documentation for more information.

Prerequisites

Ensure that you have met the PXF Object Store Prerequisites before you attempt to read data from multiple files residing in an object store.

Creating the External Table

Use the `<objstore>:hdfs:multi` profile to read multiple files in an object store each into a single table row. PXF supports the following `<objstore>` profile prefixes:

Object Store	Profile Prefix
Azure Blob Storage	wasbs
Azure Data Lake	adl
Google Cloud Storage	gs
Minio	s3
S3	s3

The following syntax creates a Greenplum Database readable external table that references one or more text files in an object store:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> text|json | LIKE <other_table> )
  LOCATION ('pxf://<path-to-files>?PROFILE=<objstore>:text:multi&SERVER=<server_name>[
&IGNORE_MISSING_PATH=<boolean>]&FILE_AS_ROW=true')
  FORMAT 'CSV');
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<path-to-files>	The absolute path to the directory or files in the object store.
PROFILE=<objstore>:text:multi	The `PROFILE` keyword must identify the specific object store. For example, `s3:text:multi`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data.
IGNORE_MISSING_PATH=<boolean>	Specify the action to take when <path-to-files> is missing or invalid. The default value is `false`, PXF returns an error in this situation. When the value is `true`, PXF ignores missing path errors and returns an empty fragment.
FILE_AS_ROW=true	The required option that instructs PXF to read each file into a single table row.
FORMAT	The `FORMAT` must specify `'CSV'`.

If you are accessing an S3 object store, you can provide S3 credentials via custom options in the

`CREATE EXTERNAL TABLE` command as described in Overriding the S3 Server Configuration with DDL.

Example

Refer to Example: Reading an HDFS Text File into a Single Table Row in the PXF HDFS documentation for an example. Modifications that you must make to run the example with an object store include:

- Copying the file to the object store instead of HDFS. For example, to copy the file to S3:

```
$ aws s3 cp /tmp/file1.txt s3://BUCKET/pxf_examples/tdir
$ aws s3 cp /tmp/file2.txt s3://BUCKET/pxf_examples/tdir
$ aws s3 cp /tmp/file3.txt s3://BUCKET/pxf_examples/tdir
```

- Using the `CREATE EXTERNAL TABLE` syntax and `LOCATION` keywords and settings described above. For example, if your server name is `s3srvcfg`:

```
CREATE EXTERNAL TABLE pxf_readfileasrow_s3( c1 text )
  LOCATION('pxf://BUCKET/pxf_examples/tdir?PROFILE=s3:text:multi&SERVER=s3srvcf
g&FILE_AS_ROW=true')
FORMAT 'CSV'
```

Reading CSV and Parquet Data from S3 Using S3 Select

The PXF S3 connector supports reading certain CSV- and Parquet-format data from S3 using the Amazon S3 Select service. S3 Select provides direct query-in-place features on data stored in Amazon S3.

When you enable it, PXF uses S3 Select to filter the contents of S3 objects to retrieve the subset of data that you request. This typically reduces both the amount of data transferred to Greenplum Database and the query time.

You can use the PXF S3 Connector with S3 Select to read:

- `gzip`- or `bzip2`-compressed `CSV` files
- `Parquet` files with `gzip`- or `snappy`-compressed columns

The data must be `UTF-8`-encoded, and may be server-side encrypted.

PXF supports column projection as well as predicate pushdown for `AND`, `OR`, and `NOT` operators when using S3 Select.

Using the Amazon S3 Select service may increase the cost of data access and retrieval. Be sure to consider the associated costs before you enable PXF to use the S3 Select service.

Enabling PXF to Use S3 Select

The `S3_SELECT` external table custom option governs PXF's use of S3 Select when accessing the S3 object store. You can provide the following values when you set the `S3_SELECT` option:

S3-SELECT Value	Description
OFF	PXF does not use S3 Select; the default.

S3-SELECT Value	Description
ON	PXF always uses S3 Select.
AUTO	PXF uses S3 Select when it will benefit access or performance.

By default, PXF does not use S3 Select (`S3_SELECT=OFF`). You can enable PXF to always use S3 Select, or to use S3 Select only when PXF determines that it could be beneficial for performance. For example, when `S3_SELECT=AUTO`, PXF automatically uses S3 Select when a query on the external table utilizes column projection or predicate pushdown, or when the referenced CSV file has a header row.

Note: The `IGNORE_MISSING_PATH` custom option is not available when you use a PXF external table to read CSV text and Parquet data from S3 using S3 Select.

Reading Parquet Data with S3 Select

PXF supports reading Parquet data from S3 as described in Reading and Writing Parquet Data in an Object Store. If you want PXF to use S3 Select when reading the Parquet data, you add the `S3_SELECT` custom option and value to the `CREATE EXTERNAL TABLE LOCATION` URI.

Specifying the Parquet Column Compression Type

If columns in the Parquet file are `gzip`- or `snappy`-compressed, use the `COMPRESSION_CODEC` custom option in the `LOCATION` URI to identify the compression codec alias. For example:

```
&COMPRESSION_CODEC=gzip
```

Or,

```
&COMPRESSION_CODEC=snappy
```

Creating the External Table

Use the following syntax to create a Greenplum Database external table that references a Parquet file on S3 that you want PXF to access with the S3 Select service:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
  LOCATION ('pxf://<path-to-file>?PROFILE=s3:parquet&SERVER=<server_name>&S3_SELECT=ON
|AUTO[&<other-custom-option>=<value>[...]]')
FORMAT 'CSV';
```

You *must* specify `FORMAT 'CSV'` when you enable PXF to use S3 Select on an external table that accesses a Parquet file on S3.

For example, use the following command to have PXF use S3 Select to access a Parquet file on S3 when optimal:

```
CREATE EXTERNAL TABLE parquet_on_s3 ( LIKE table1 )
  LOCATION ('pxf://bucket/file.parquet?PROFILE=s3:parquet&SERVER=s3srvcfg&S3_SELECT=AU
TO')
FORMAT 'CSV';
```

Reading CSV files with S3 Select

PXF supports reading CSV data from S3 as described in Reading and Writing Text Data in an Object Store. If you want PXF to use S3 Select when reading the CSV data, you add the `S3_SELECT` custom option and value to the `CREATE EXTERNAL TABLE LOCATION` URI. You may also specify the delimiter formatter option and the file header and compression custom options.

Handling the CSV File Header

PXF can read a CSV file with a header row *only* when the S3 Connector uses the Amazon S3 Select service to access the file on S3. PXF does not support reading a CSV file that includes a header row from any other external data store.

CSV files may include a header line. When you enable PXF to use S3 Select to access a CSV-format file, you use the `FILE_HEADER` custom option in the `LOCATION` URI to identify whether or not the CSV file has a header row and, if so, how you want PXF to handle the header. PXF never returns the header row.

Note: You *must* specify `S3_SELECT=ON` or `S3_SELECT=AUTO` when the CSV file has a header row. Do not specify `S3_SELECT=OFF` in this case.

The `FILE_HEADER` option takes the following values:

FILE_HEADER Value	Description
NONE	The file has no header row; the default.
IGNORE	The file has a header row; ignore the header. Use when the order of the columns in the external table and the CSV file are the same. (When the column order is the same, the column names and the CSV header names may be different.)
USE	The file has a header row; read the header. Use when the external table column names and the CSV header names are the same, but are in a different order.

If both the order and the names of the external table columns and the CSV header are the same, you can specify either `FILE_HEADER=IGNORE` or `FILE_HEADER=USE`.

PXF cannot match the CSV data with the external table definition when both the order and the names of the external table columns are different from the CSV header columns. Any query on an external table with these conditions fails with the error `Some headers in the query are missing from the file`.

For example, if the order of the columns in the CSV file header and the external table are the same, add the following to the `CREATE EXTERNAL TABLE LOCATION` URI to have PXF ignore the CSV header:

```
&FILE_HEADER=IGNORE
```

Specifying the CSV File Compression Type

If the CSV file is `gzip`- or `bzip2`-compressed, use the `COMPRESSION_CODEC` custom option in the `LOCATION` URI to identify the compression codec alias. For example:

```
&COMPRESSION_CODEC=gzip
```

Or,

```
&COMPRESSION_CODEC=bzip2
```

Creating the External Table

Use the following syntax to create a Greenplum Database external table that references a CSV file on S3 that you want PXF to access with the S3 Select service:

```
CREATE EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<path-to-file>
    ?PROFILE=s3:text&SERVER=<server_name>&S3_SELECT=ON|AUTO[&FILE_HEADER=IGNORE|USE][&
COMPRESSION_CODEC=gzip|bzip2][&<other-custom-option>=<value>[...]]')
FORMAT 'CSV' [(delimiter '<delim_char>')];
```

Note: Do not use the `(HEADER)` formatter option in the `CREATE EXTERNAL TABLE` command.

For example, use the following command to have PXF always use S3 Select to access a `gzip`-compressed file on S3, where the field delimiter is a pipe (’|’) character and the external table and CSV header columns are in the same order.

```
CREATE EXTERNAL TABLE gzippedcsv_on_s3 ( LIKE table2 )
  LOCATION ('pxf://bucket/file.csv.gz?PROFILE=s3:text&SERVER=s3srvcfg&S3_SELECT=ON&FIL
E_HEADER=USE')
FORMAT 'CSV' (delimiter '|');
```

Accessing an SQL Database with PXF (JDBC)

Some of your data may already reside in an external SQL database. PXF provides access to this data via the PXF JDBC connector. The JDBC connector is a JDBC client. It can read data from and write data to SQL databases including MySQL, ORACLE, Microsoft SQL Server, DB2, PostgreSQL, Hive, and Apache Ignite.

This section describes how to use the PXF JDBC connector to access data in an external SQL database, including how to create and query or insert data into a PXF external table that references a table in an external database.

The JDBC connector does not guarantee consistency when writing to an external SQL database. Be aware that if an `INSERT` operation fails, some data may be written to the external database table. If you require consistency for writes, consider writing to a staging table in the external database, and loading to the target table only after verifying the write operation.

Prerequisites

Before you access an external SQL database using the PXF JDBC connector, ensure that:

- You have configured and initialized PXF, and PXF is running on each Greenplum Database segment host. See Configuring PXF for additional information.
- You can identify the PXF user configuration directory (`$PXF_CONF`).
- Connectivity exists between all Greenplum Database segment hosts and the external SQL database.
- You have configured your external SQL database for user access from all Greenplum Database segment hosts.
- You have registered any JDBC driver JAR dependencies.
- (Recommended) You have created one or more named PXF JDBC connector server configurations as described in Configuring the PXF JDBC Connector.

Data Types Supported

The PXF JDBC connector supports the following data types:

- INTEGER, BIGINT, SMALLINT
- REAL, FLOAT8
- NUMERIC
- BOOLEAN

- VARCHAR, BPCHAR, TEXT
- DATE
- TIMESTAMP
- BYTEA

Any data type not listed above is not supported by the PXF JDBC connector.

Note: The JDBC connector does not support reading or writing Hive data stored as a byte array (`byte[]`).

Accessing an External SQL Database

The PXF JDBC connector supports a single profile named `Jdbc`. You can both read data from and write data to an external SQL database table with this profile. You can also use the connector to run a static, named query in external SQL database and read the results.

To access data in a remote SQL database, you create a readable or writable Greenplum Database external table that references the remote database table. The Greenplum Database external table and the remote database table or query result tuple must have the same definition; the column names and types must match.

Use the following syntax to create a Greenplum Database external table that references a remote SQL database table or a query result from the remote database:

```
CREATE [READABLE | WRITABLE] EXTERNAL TABLE <table_name>
    ( <column_name> <data_type> [, ...] | LIKE <other_table> )
LOCATION ('pxf://<external-table-name>|query:<query_name>?PROFILE=Jdbc[&SERVER=<server_name>][&<custom-option>=<value>[...]]')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import'|'pxfwritable_export');
```

The specific keywords and values used in the Greenplum Database CREATE EXTERNAL TABLE command are described in the table below.

Keyword	Value
<external-table-name>	The full name of the external table. Depends on the external SQL database, may include a schema name and a table name.
query:<query_name>	The name of the query to execute in the remote SQL database.
PROFILE	The `PROFILE` keyword value must specify `Jdbc`.
SERVER=<server_name>	The named server configuration that PXF uses to access the data. Optional; PXF uses the `default` server if not specified.
<custom-option>=<value>	<custom-option> is profile-specific. `Jdbc` profile-specific options are discussed in the next section.
FORMAT 'CUSTOM'	The JDBC `CUSTOM FORMAT` supports the built-in `'pxfwritable_import' FORMATTER` function for read operations and the built-in `'pxfwritable_export'` function for write operations.

Note: You cannot use the `HEADER` option in your `FORMAT` specification when you create a PXF external table.

JDBC Custom Options

You include JDBC connector custom options in the `LOCATION` URI, prefacing each option with an ampersand `&`. `CREATE EXTERNAL TABLE` <custom-option>s supported by the `Jdbc` profile include:

Option Name	Operation	Description
BATCH_SIZE	Write	Integer that identifies the number of `INSERT` operations to batch to the external SQL database. Write batching is enabled by default; the default value is 100.
FETCH_SIZE	Read	Integer that identifies the number of rows to buffer when reading from an external SQL database. Read row batching is enabled by default; the default read fetch size is 1000.
QUERY_TIMEOUT	Read/Write	Integer that identifies the amount of time (in seconds) that the JDBC driver waits for a statement to execute. The default wait time is infinite.
POOL_SIZE	Write	Enable thread pooling on `INSERT` operations and identify the number of threads in the pool. Thread pooling is disabled by default.
PARTITION_BY	Read	Enables read partitioning. The partition column, <column-name>:<column-type>. You may specify only one partition column. The JDBC connector supports `date`, `int`, and `enum` <column-type> values, where `int` represents any JDBC integral type. If you do not identify a `PARTITION_BY` column, a single PXF instance services the read request.
RANGE	Read	Required when `PARTITION_BY` is specified. The query range; used as a hint to aid the creation of partitions. The `RANGE` format is dependent upon the data type of the partition column. When the partition column is an `enum` type, `RANGE` must specify a list of values, <value>:<value>[:<value>[…]], each of which forms its own fragment. If the partition column is an `int` or `date` type, `RANGE` must specify <start-value>:<end-value> and represents the interval from <start-value> through <end-value>, inclusive. The `RANGE` for an `int` partition column may span any 64-bit signed integer values. If the partition column is a `date` type, use the `yyyy-MM-dd` date format.
INTERVAL	Read	Required when `PARTITION_BY` is specified and of the `int`, `bigint`, or `date` type. The interval, <interval-value>[:<interval-unit>], of one fragment. Used with `RANGE` as a hint to aid the creation of partitions. Specify the size of the fragment in <interval-value>. If the partition column is a `date` type, use the <interval-unit> to specify `year`, `month`, or `day`. PXF ignores `INTERVAL` when the `PARTITION_BY` column is of the `enum` type.
QUOTE_COLUMNS	Read	Controls whether PXF should quote column names when constructing an SQL query to the external database. Specify `true` to force PXF to quote all column names; PXF does not quote column names if any other value is provided. If `QUOTE_COLUMNS` is not specified (the default), PXF automatically quotes *all* column names in the query when *any* column name: - includes special characters, or - is mixed case and the external database does not support unquoted mixed case identifiers.

Batching Insert Operations (Write)

When the JDBC driver of the external SQL database supports it, batching of `INSERT` operations may significantly increase performance.

Write batching is enabled by default, and the default batch size is 100. To disable batching or to modify the default batch size value, create the PXF external table with a `BATCH_SIZE` setting:

- `BATCH_SIZE=0` or `BATCH_SIZE=1` - disables batching

- `BATCH_SIZE=(n>1)` - sets the `BATCH_SIZE` to `n`

When the external database JDBC driver does not support batching, the behaviour of the PXF JDBC connector depends on the `BATCH_SIZE` setting as follows:

- `BATCH_SIZE` omitted - The JDBC connector inserts without batching.
- `BATCH_SIZE=(n>1)` - The `INSERT` operation fails and the connector returns an error.

Batching on Read Operations

By default, the PXF JDBC connector automatically batches the rows it fetches from an external database table. The default row fetch size is 1000. To modify the default fetch size value, specify a `FETCH_SIZE` when you create the PXF external table. For example:

```
FETCH_SIZE=5000
```

If the external database JDBC driver does not support batching on read, you must explicitly disable read row batching by setting `FETCH_SIZE=0`.

Thread Pooling (Write)

The PXF JDBC connector can further increase write performance by processing `INSERT` operations in multiple threads when threading is supported by the JDBC driver of the external SQL database.

Consider using batching together with a thread pool. When used together, each thread receives and processes one complete batch of data. If you use a thread pool without batching, each thread in the pool receives exactly one tuple.

The JDBC connector returns an error when any thread in the thread pool fails. Be aware that if an `INSERT` operation fails, some data may be written to the external database table.

To disable or enable a thread pool and set the pool size, create the PXF external table with a `POOL_SIZE` setting as follows:

- `POOL_SIZE=(n<1)` - thread pool size is the number of CPUs in the system
- `POOL_SIZE=1` - disable thread pooling
- `POOL_SIZE=(n>1)` - set the `POOL_SIZE` to `n`

Partitioning (Read)

The PXF JDBC connector supports simultaneous read access from PXF instances running on multiple segment hosts to an external SQL table. This feature is referred to as partitioning. Read partitioning is not enabled by default. To enable read partitioning, set the `PARTITION_BY`, `RANGE`, and `INTERVAL` custom options when you create the PXF external table.

PXF uses the `RANGE` and `INTERVAL` values and the `PARTITON_BY` column that you specify to assign specific data rows in the external table to PXF instances running on the Greenplum Database segment hosts. This column selection is specific to PXF processing, and has no relationship to a partition column that you may have specified for the table in the external SQL database.

Example JDBC <custom-option> substrings that identify partitioning parameters:

```
&PARTITION_BY=id:int&RANGE=1:100&INTERVAL=5
&PARTITION_BY=year:int&RANGE=2011:2013&INTERVAL=1
&PARTITION_BY=createdate:date&RANGE=2013-01-01:2016-01-01&INTERVAL=1:month
&PARTITION_BY=color:enum&RANGE=red:yellow:blue
```

When you enable partitioning, the PXF JDBC connector splits a `SELECT` query into multiple subqueries that retrieve a subset of the data, each of which is called a fragment. The JDBC connector automatically adds extra query constraints (`WHERE` expressions) to each fragment to guarantee that every tuple of data is retrieved from the external database exactly once.

For example, when a user queries a PXF external table created with a `LOCATION` clause that specifies `&PARTITION_BY=id:int&RANGE=1:5&INTERVAL=2`, PXF generates 5 fragments: two according to the partition settings and up to three implicitly generated fragments. The constraints associated with each fragment are as follows:

- Fragment 1: WHERE (id < 1) - implicitly-generated fragment for RANGE start-bounded interval
- Fragment 2: WHERE (id >= 1) AND (id < 3) - fragment specified by partition settings
- Fragment 3: WHERE (id >= 3) AND (id < 5) - fragment specified by partition settings
- Fragment 4: WHERE (id >= 5) - implicitly-generated fragment for RANGE end-bounded interval
- Fragment 5: WHERE (id IS NULL) - implicitly-generated fragment

PXF distributes the fragments among Greenplum Database segments. A PXF instance running on a segment host spawns a thread for each segment on that host that services a fragment. If the number of fragments is less than or equal to the number of Greenplum segments configured on a segment host, a single PXF instance may service all of the fragments. Each PXF instance sends its results back to Greenplum Database, where they are collected and returned to the user.

When you specify the `PARTITION_BY` option, tune the `INTERVAL` value and unit based upon the optimal number of JDBC connections to the target database and the optimal distribution of external data across Greenplum Database segments. The `INTERVAL` low boundary is driven by the number of Greenplum Database segments while the high boundary is driven by the acceptable number of JDBC connections to the target database. The `INTERVAL` setting influences the number of fragments, and should ideally not be set too high nor too low. Testing with multiple values may help you select the optimal settings.

Example: Reading From and Writing to a PostgreSQL Table

In this example, you:

- Create a PostgreSQL database and table, and insert data into the table
- Create a PostgreSQL user and assign all privileges on the table to the user
- Configure the PXF JDBC connector to access the PostgreSQL database
- Create a PXF readable external table that references the PostgreSQL table
- Read the data in the PostgreSQL table
- Create a PXF writable external table that references the PostgreSQL table
- Write data to the PostgreSQL table

- Read the data in the PostgreSQL table again

Create a PostgreSQL Table

Perform the following steps to create a PostgreSQL table named `forpxf_table1` in the `public` schema of a database named `pgtestdb`, and grant a user named `pxfuser1` all privileges on this table:

1. Identify the host name and port of your PostgreSQL server.
2. Connect to the default PostgreSQL database as the `postgres` user. For example, if your PostgreSQL server is running on the default port on the host named `pserver`:

   ```
   $ psql -U postgres -h pserver
   ```

3. Create a PostgreSQL database named `pgtestdb` and connect to this database:

   ```
   =# CREATE DATABASE pgtestdb;
   =# \connect pgtestdb;
   ```

4. Create a table named `forpxf_table1` and insert some data into this table:

   ```
   =# CREATE TABLE forpxf_table1(id int);
   =# INSERT INTO forpxf_table1 VALUES (1);
   =# INSERT INTO forpxf_table1 VALUES (2);
   =# INSERT INTO forpxf_table1 VALUES (3);
   ```

5. Create a PostgreSQL user named `pxfuser1`:

   ```
   =# CREATE USER pxfuser1 WITH PASSWORD 'changeme';
   ```

6. Assign user `pxfuser1` all privileges on table `forpxf_table1`, and exit the `psql` subsystem:

   ```
   =# GRANT ALL ON forpxf_table1 TO pxfuser1;
   =# \q
   ```

   With these privileges, `pxfuser1` can read from and write to the `forpxf_table1` table.

7. Update the PostgreSQL configuration to allow user `pxfuser1` to access `pgtestdb` from each Greenplum Database segment host. This configuration is specific to your PostgreSQL environment. You will update the `/var/lib/pgsql/pg_hba.conf` file and then restart the PostgreSQL server.

Configure the JDBC Connector

You must create a JDBC server configuration for PostgreSQL, download the PostgreSQL driver JAR file to your system, copy the JAR file to the PXF user configuration directory, synchronize the PXF configuration, and then restart PXF.

This procedure will typically be performed by the Greenplum Database administrator.

1. Log in to the Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Create a JDBC server configuration for PostgreSQL as described in Example Configuration Procedure, naming the server/directory `pgsrvcfg`. The `jdbc-site.xml` file contents should look similar to the following (substitute your PostgreSQL host system for `pgserverhost`):

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<property>
    <name>jdbc.driver</name>
    <value>org.postgresql.Driver</value>
</property>
<property>
    <name>jdbc.url</name>
    <value>jdbc:postgresql://pgserverhost:5432/pgtestdb</value>
</property>
<property>
    <name>jdbc.user</name>
    <value>pxfuser1</value>
</property>
<property>
    <name>jdbc.password</name>
    <value>changeme</value>
</property>
</configuration>
```

3. Synchronize the PXF server configuration to the Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

Read from the PostgreSQL Table

Perform the following procedure to create a PXF external table that references the `forpxf_table1` PostgreSQL table that you created in the previous section, and read the data in the table:

1. Create the PXF external table specifying the `Jdbc` profile. For example:

```
gpadmin=# CREATE EXTERNAL TABLE pxf_tblfrompg(id int)
            LOCATION ('pxf://public.forpxf_table1?PROFILE=Jdbc&SERVER=pgsrvcfg'
)
            FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

2. Display all rows of the `pxf_tblfrompg` table:

```
gpadmin=# SELECT * FROM pxf_tblfrompg;
 id
----
  1
  2
  3
(3 rows)
```

Write to the PostgreSQL Table

Perform the following procedure to insert some data into the `forpxf_table1` Postgres table and then read from the table. You must create a new external table for the write operation.

1. Create a writable PXF external table specifying the `Jdbc` profile. For example:

```
gpadmin=# CREATE WRITABLE EXTERNAL TABLE pxf_writeto_postgres(id int)
            LOCATION ('pxf://public.forpxf_table1?PROFILE=Jdbc&SERVER=pgsrvcfg'
)
          FORMAT 'CUSTOM' (FORMATTER='pxfwritable_export');
```

2. Insert some data into the `pxf_writeto_postgres` table. For example:

```
=# INSERT INTO pxf_writeto_postgres VALUES (111);
=# INSERT INTO pxf_writeto_postgres VALUES (222);
=# INSERT INTO pxf_writeto_postgres VALUES (333);
```

3. Use the `pxf_tblfrompg` readable external table that you created in the previous section to view the new data in the `forpxf_table1` PostgreSQL table:

```
gpadmin=# SELECT * FROM pxf_tblfrompg ORDER BY id DESC;
 id
-----
 333
 222
 111
   3
   2
   1
(6 rows)
```

About Using Named Queries

The PXF JDBC Connector allows you to specify a statically-defined query to run against the remote SQL database. Consider using a *named query* when:

- You need to join several tables that all reside in the same external database.
- You want to perform complex aggregation closer to the data source.
- You would use, but are not allowed to create, a `VIEW` in the external database.
- You would rather consume computational resources in the external system to minimize utilization of Greenplum Database resources.
- You want to run a HIVE query and control resource utilization via YARN.

The Greenplum Database administrator defines a query and provides you with the query name to use when you create the external table. Instead of a table name, you specify `query:<query_name>` in the `CREATE EXTERNAL TABLE LOCATION` clause to instruct the PXF JDBC connector to run the static query named `<query_name>` in the remote SQL database.

PXF supports named queries only with readable external tables. You must create a unique Greenplum Database readable external table for each query that you want to run.

The names and types of the external table columns must exactly match the names, types, and order of the columns return by the query result. If the query returns the results of an aggregation or other function, be sure to use the `AS` qualifier to specify a specific column name.

For example, suppose that you are working with PostgreSQL tables that have the following definitions:

```
CREATE TABLE customers(id int, name text, city text, state text);
CREATE TABLE orders(customer_id int, amount int, month int, year int);
```

And this PostgreSQL query that the administrator named `order_rpt`:

```
SELECT c.name, sum(o.amount) AS total, o.month
  FROM customers c JOIN orders o ON c.id = o.customer_id
  WHERE c.state = 'CO'
GROUP BY c.name, o.month
```

This query returns tuples of type `(name text, total int, month int)`. If the `order_rpt` query is defined for the PXF JDBC server named `pgserver`, you could create a Greenplum Database external table to read these query results as follows:

```
CREATE EXTERNAL TABLE orderrpt_frompg(name text, total int, month int)
  LOCATION ('pxf://query:order_rpt?PROFILE=Jdbc&SERVER=pgserver&PARTITION_BY=month:int
&RANGE=1:13&INTERVAL=3')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

This command references a query named `order_rpt` defined in the `pgserver` server configuration. It also specifies JDBC read partitioning options that provide PXF with the information that it uses to split/partition the query result data across its servers/segments.

The PXF JDBC connector automatically applies column projection and filter pushdown to external tables that reference named queries.

Example: Reading the Results of a PostgreSQL Query

In this example, you:

- Use the PostgreSQL database `pgtestdb`, user `pxfuser1`, and PXF JDBC connector server configuration `pgsrvcfg` that you created in Example: Reading From and Writing to a PostgreSQL Database.
- Create two PostgreSQL tables and insert data into the tables.
- Assign all privileges on the tables to `pxfuser1`.
- Define a named query that performs a complex SQL statement on the two PostgreSQL tables, and add the query to the `pgsrvcfg` JDBC server configuration.
- Create a PXF readable external table definition that matches the query result tuple and also specifies read partitioning options.
- Read the query results, making use of PXF column projection and filter pushdown.

Create the PostgreSQL Tables and Assign Permissions

Perform the following procedure to create PostgreSQL tables named `customers` and `orders` in the `public` schema of the database named `pgtestdb`, and grant the user named `pxfuser1` all privileges on these tables:

1. Identify the host name and port of your PostgreSQL server.
2. Connect to the `pgtestdb` PostgreSQL database as the `postgres` user. For example, if your

PostgreSQL server is running on the default port on the host named `pserver`:

```
$ psql -U postgres -h pserver -d pgtestdb
```

3. Create a table named `customers` and insert some data into this table:

```
CREATE TABLE customers(id int, name text, city text, state text);
INSERT INTO customers VALUES (111, 'Bill', 'Helena', 'MT');
INSERT INTO customers VALUES (222, 'Mary', 'Athens', 'OH');
INSERT INTO customers VALUES (333, 'Tom', 'Denver', 'CO');
INSERT INTO customers VALUES (444, 'Kate', 'Helena', 'MT');
INSERT INTO customers VALUES (555, 'Harry', 'Columbus', 'OH');
INSERT INTO customers VALUES (666, 'Kim', 'Denver', 'CO');
INSERT INTO customers VALUES (777, 'Erik', 'Missoula', 'MT');
INSERT INTO customers VALUES (888, 'Laura', 'Athens', 'OH');
INSERT INTO customers VALUES (999, 'Matt', 'Aurora', 'CO');
```

4. Create a table named `orders` and insert some data into this table:

```
CREATE TABLE orders(customer_id int, amount int, month int, year int);
INSERT INTO orders VALUES (111, 12, 12, 2018);
INSERT INTO orders VALUES (222, 234, 11, 2018);
INSERT INTO orders VALUES (333, 34, 7, 2018);
INSERT INTO orders VALUES (444, 456, 111, 2018);
INSERT INTO orders VALUES (555, 56, 11, 2018);
INSERT INTO orders VALUES (666, 678, 12, 2018);
INSERT INTO orders VALUES (777, 12, 9, 2018);
INSERT INTO orders VALUES (888, 120, 10, 2018);
INSERT INTO orders VALUES (999, 120, 11, 2018);
```

5. Assign user `pxfuser1` all privileges on tables `customers` and `orders`, and then exit the `psql` subsystem:

```
GRANT ALL ON customers TO pxfuser1;
GRANT ALL ON orders TO pxfuser1;
\q
```

Configure the Named Query

In this procedure you create a named query text file, add it to the `pgsrvcfg` JDBC server configuration, and synchronize the PXF configuration to the Greenplum Database cluster.

This procedure will typically be performed by the Greenplum Database administrator.

1. Log in to the Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. Navigate to the JDBC server configuration directory `pgsrvcfg`. For example:

```
gpadmin@gpmaster$ cd $PXF_CONF/servers/pgsrvcfg
```

3. Open a query text file named `pg_order_report.sql` in a text editor and copy/paste the following query into the file:

```
SELECT c.name, c.city, sum(o.amount) AS total, o.month
  FROM customers c JOIN orders o ON c.id = o.customer_id
  WHERE c.state = 'CO'
GROUP BY c.name, c.city, o.month
```

4. Save the file and exit the editor.
5. Synchronize these changes to the PXF configuration to the Greenplum Database cluster:

```
gpadmin@gpmaster$ pxf cluster sync
```

Read the Query Results

Perform the following procedure on your Greenplum Database cluster to create a PXF external table that references the query file that you created in the previous section, and then reads the query result data:

1. Create the PXF external table specifying the `Jdbc` profile. For example:

```
CREATE EXTERNAL TABLE pxf_queryres_frompg(name text, city text, total int, mont
h int)
  LOCATION ('pxf://query:pg_order_report?PROFILE=Jdbc&SERVER=pgsrvcfg&PARTITION
_BY=month:int&RANGE=1:13&INTERVAL=3')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

   With this partitioning scheme, PXF will issue 4 queries to the remote SQL database, one query per quarter. Each query will return customer names and the total amount of all of their orders in a given month, aggregated per customer, per month, for each month of the target quarter. Greenplum Database will then combine the data into a single result set for you when you query the external table.

2. Display all rows of the query result:

```
SELECT * FROM pxf_queryres_frompg ORDER BY city, total;

 name |  city  | total | month
------+--------+-------+-------
 Matt | Aurora |   120 |    11
 Tom  | Denver |    34 |     7
 Kim  | Denver |   678 |    12
(3 rows)
```

3. Use column projection to display the order total per city:

```
SELECT city, sum(total) FROM pxf_queryres_frompg GROUP BY city;

  city  | sum
--------+-----
 Aurora | 120
 Denver | 712
(2 rows)
```

   When you execute this query, PXF requests and retrieves query results for only the `city` and `total` columns, reducing the amount of data sent back to Greenplum Database.

4. Provide additional filters and aggregations to filter the `total` in PostgreSQL:

```
SELECT city, sum(total) FROM pxf_queryres_frompg
            WHERE total > 100
            GROUP BY city;

  city   | sum
--------+-----
 Denver | 678
 Aurora | 120
(2 rows)
```

In this example, PXF will add the `WHERE` filter to the subquery. This filter is pushed to and executed on the remote database system, reducing the amount of data that PXF sends back to Greenplum Database. The `GROUP BY` aggregation, however, is not pushed to the remote and is performed by Greenplum.

Overriding the JDBC Server Configuration with DDL

You can override certain properties in a JDBC server configuration for a specific external database table by directly specifying the custom option in the `CREATE EXTERNAL TABLE LOCATION` clause:

Custom Option Name	jdbc-site.xml Property Name
JDBC_DRIVER	jdbc.driver
DB_URL	jdbc.url
USER	jdbc.user
PASS	jdbc.password
BATCH_SIZE	jdbc.statement.batchSize
FETCH_SIZE	jdbc.statement.fetchSize
QUERY_TIMEOUT	jdbc.statement.queryTimeout

Example JDBC connection strings specified via custom options:

```
&JDBC_DRIVER=org.postgresql.Driver&DB_URL=jdbc:postgresql://pgserverhost:5432/pgtestdb
&USER=pguser1&PASS=changeme
&JDBC_DRIVER=com.mysql.jdbc.Driver&DB_URL=jdbc:mysql://mysqlhost:3306/testdb&USER=user
1&PASS=changeme
```

For example:

```
CREATE EXTERNAL TABLE pxf_pgtbl(name text, orders int)
  LOCATION ('pxf://public.forpxf_table1?PROFILE=Jdbc&JDBC_DRIVER=org.postgresql.Driver&DB
_URL=jdbc:postgresql://pgserverhost:5432/pgtestdb&USER=pxfuser1&PASS=changeme')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_export');
```

Credentials that you provide in this manner are visible as part of the external table definition. Do not use this method of passing credentials in a production environment.

Refer to Configuration Property Precedence for detailed information about the precedence rules that PXF uses to obtain configuration property settings for a Greenplum Database user.

Troubleshooting PXF

PXF Errors

The following table describes some errors you may encounter while using PXF:

Error Message	Discussion
Protocol "pxf" does not exist	**Cause**: The `pxf` extension was not registered. **Solution**: Create (enable) the PXF extension for the database as described in the PXF Enable Procedure.
Invalid URI pxf://<path-to-data>: missing options section	**Cause**: The `LOCATION` URI does not include the profile or other required options. **Solution**: Provide the profile and required options in the URI.
org.apache.hadoop.mapred.InvalidInputException: Input path does not exist: hdfs://<namenode>:8020/<path-to-file>	**Cause**: The HDFS file that you specified in <path-to-file> does not exist. **Solution**: Provide the path to an existing HDFS file.
NoSuchObjectException(message:<schema>.<hivetable> table not found)	**Cause**: The Hive table that you specified with <schema>.<hivetable> does not exist. **Solution**: Provide the name of an existing Hive table.
Failed to connect to <segment-host> port 5888: Connection refused (libchurl.c:944) (<segment-id> slice<N> <segment-host>:40000 pid=<process-id>) ...	**Cause**: PXF is not running on <segment-host>. **Solution**: Restart PXF on <segment-host>.
ERROR: failed to acquire resources on one or more segments *DETAIL*: could not connect to server: Connection refused Is the server running on host "<segment-host>" and accepting TCP/IP connections on port 40000?(seg<N> <segment-host>:40000)	**Cause**: The Greenplum Database segment host <segment-host> is down.
org.apache.hadoop.security.AccessControlException: Permission denied: user=, access=READ, inode="":::-rw——-	**Cause**: The Greenplum Database user that executed the PXF operation does not have permission to access the underlying Hadoop service (HDFS or Hive). See Configuring the Hadoop User, User Impersonation, and Proxying.

PXF Logging

Enabling more verbose logging may aid PXF troubleshooting efforts. PXF provides two categories of message logging: service-level and client-level.

Service-Level Logging

PXF utilizes `log4j` for service-level logging. PXF-service-related log messages are captured in a log file specified by PXF's `log4j` properties file, `$PXF_CONF/conf/pxf-log4j.properties`. The default PXF logging configuration will write `INFO` and more severe level logs to `$PXF_CONF/logs/pxf-service.log`. You can configure the logging level and the log file location.

PXF provides more detailed logging when the `DEBUG` level is enabled. To configure PXF `DEBUG` logging and examine the output:

1. Log in to your Greenplum Database master node:

    ```
    $ ssh gpadmin@<gpmaster>
    ```

2. Open `$PXF_CONF/conf/pxf-log4j.properties` in an editor, uncomment the following line, save the file, and exit the editor:

    ```
    #log4j.logger.org.greenplum.pxf=DEBUG
    ```

3. Use the `pxf cluster sync` command to copy the updated `pxf-log4j.properties` file to the Greenplum Database cluster. For example:

    ```
    gpadmin@gpmaster$ pxf cluster sync
    ```

4. Restart PXF on each Greenplum Database segment host as described in Restarting PXF.

5. With `DEBUG` level logging now enabled, you can perform your PXF operations. Be sure to make note of the time; this will direct you to the relevant log messages in `$PXF_CONF/logs/pxf-service.log`.

    ```
    $ date
    Wed Oct  4 09:30:06 MDT 2017
    $ psql -d <dbname>
    ```

6. Create and query an external table. For example:

    ```
    dbname=> CREATE EXTERNAL TABLE hdfstest(id int, newid int)
        LOCATION ('pxf://data/dir/hdfsfile?PROFILE=hdfs:text')
        FORMAT 'TEXT' (delimiter='E',');
    dbname=> SELECT * FROM hdfstest;
    <select output>
    ```

7. Finally, examine/collect the log messages from `pxf-service.log`.

Note: `DEBUG` logging is quite verbose and has a performance impact. Remember to turn off PXF service `DEBUG` logging after you have collected the desired information.

Client-Level Logging

Database-level client logging may provide insight into internal PXF service operations.

Enable Greenplum Database and PXF debug message logging during operations on PXF external tables by setting the `client_min_messages` server configuration parameter to `DEBUG2` in your `psql` session.

```
$ psql -d <dbname>
```

```
dbname=# SET client_min_messages=DEBUG2;
dbname=# SELECT * FROM hdfstest;
...
DEBUG2:  churl http header: cell #19: X-GP-URL-HOST: seghost1  (seg0 slice1 127.0.0.1:
40000 pid=3981)
CONTEXT:  External table hdfstest
DEBUG2:  churl http header: cell #20: X-GP-URL-PORT: 5888  (seg0 slice1 127.0.0.1:4000
0 pid=3981)
CONTEXT:  External table hdfstest
DEBUG2:  churl http header: cell #21: X-GP-DATA-DIR: data/dir/hdfsfile  (seg0 slice1 1
27.0.0.1:40000 pid=3981)
CONTEXT:  External table hdfstest
DEBUG2:  churl http header: cell #22: X-GP-OPTIONS-PROFILE: hdfs:text  (seg0 slice1 12
7.0.0.1:40000 pid=3981)
CONTEXT:  External table hdfstest
...
```

Examine/collect the log messages from `stdout`.

Note: `DEBUG2` database session logging has a performance impact. Remember to turn off `DEBUG2` logging after you have collected the desired information.

```
dbname=# SET client_min_messages=NOTICE;
```

Addressing PXF Memory Issues

Because a single PXF agent (JVM) serves multiple segments on a segment host, the PXF heap size can be a limiting runtime factor. This will be more evident under concurrent workloads and/or queries against large files. You may run into situations where a query will hang or fail due to insufficient memory or the Java garbage collector impacting response times. To avert or remedy these situations, first try increasing the Java maximum heap size or decreasing the Tomcat maximum number of threads, depending upon what works best for your system configuration. You may also choose to configure PXF to perform specific actions when it detects an out of memory condition.

Note: The configuration changes described in this topic require modifying config files on *each* node in your Greenplum Database cluster. After you perform the updates on the master, be sure to synchronize the PXF configuration to the Greenplum Database cluster.

Configuring Out of Memory Condition Actions

In an out of memory (OOM) situation, PXF returns the following error in response to a query:

```
java.lang.OutOfMemoryError: Java heap space
```

You can configure the PXF JVM to enable/disable the following actions when it detects an OOM condition:

- Auto-kill the PXF server (enabled by default).
- Dump the Java heap (disabled by default).

Auto-Killing the PXF Server

By default, PXF is configured such that when the PXF JVM detects an out of memory condition on a segment host, it automatically runs a script that kills the PXF server running on the host. The `PXF_OOM_KILL` configuration property governs this auto-kill behavior.

When auto-kill is enabled and the PXF JVM detects an OOM condition and kills the PXF server on the segment host:

- PXF logs the following messages to `$PXF_CONF/logs/catalina.out` on the segment host:

  ```
  =====> <date> PXF Out of memory detected <======
  =====> <date> PXF shutdown scheduled <======
  ```

- Any query that you run on a PXF external table will fail with the following error until you restart the PXF server on the segment host:

  ```
  ... Failed to connect to <host> port 5888: Connection refused
  ```

When the PXF server on a segment host is shut down in this manner, you must explicitly restart the PXF server on the host. See the pxf reference page for more information on the `pxf start` command.

Refer to the configuration procedure below for the instructions to disable/enable this PXF configuration property.

Dumping the Java Heap

In an out of memory situation, it may be useful to capture the Java heap dump to help determine what factors contributed to the resource exhaustion. You can use the `PXF_OOM_DUMP_PATH` property to configure PXF to write the heap dump to a file when it detects an OOM condition. By default, PXF does not dump the Java heap on OOM.

If you choose to enable the heap dump on OOM, you must set `PXF_OOM_DUMP_PATH` to the absolute path to a file or directory:

- If you specify a directory, the PXF JVM writes the heap dump to the file `<directory>/java_pid<pid>.hprof`, where `<pid>` identifies the process ID of the PXF server instance. The PXF JVM writes a new file to the directory every time the JVM goes OOM.
- If you specify a file and the file does not exist, the PXF JVM writes the heap dump to the file when it detects an OOM. If the file already exists, the JVM will not dump the heap.

Ensure that the `gpadmin` user has write access to the dump file or directory.

Note: Heap dump files are often rather large. If you enable heap dump on OOM for PXF and specify a directory for `PXF_OOM_DUMP_PATH`, multiple OOMs will generate multiple files in the directory and could potentially consume a large amount of disk space. If you specify a file for `PXF_OOM_DUMP_PATH`, disk usage is constant when the file name does not change. You must rename the dump file or configure a different `PXF_OOM_DUMP_PATH` to generate subsequent heap dumps.

Refer to the configuration procedure below for the instructions to enable/disable this PXF configuration property.

Procedure

Auto-kill of the PXF server on OOM is enabled by default. Heap dump generation on OOM is disabled by default. To configure one or both of these properties, perform the following procedure:

1. Log in to your Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Edit the `$PXF_CONF/conf/pxf-env.sh` file. For example:

   ```
   gpadmin@gpmaster$ vi $PXF_CONF/conf/pxf-env.sh
   ```

3. If you want to configure (i.e. turn off, or turn back on) auto-kill of the PXF server on OOM, locate the `PXF_OOM_KILL` property in the `pxf-env.sh` file. If the setting is commented out, uncomment it, and then update the value. For example, to turn off this behavior, set the value to `false`:

   ```
   export PXF_OOM_KILL=false
   ```

4. If you want to configure (i.e. turn on, or turn back off) automatic heap dumping when the PXF server hits an OOM condition, locate the `PXF_OOM_DUMP_PATH` setting in the `pxf-env.sh` file.

   1. To turn this behavior on, set the `PXF_OOM_DUMP_PATH` property value to the file system location to which you want the PXF JVM to dump the Java heap. For example, to dump to a file named `/home/gpadmin/pxfoom_segh1`:

      ```
      export PXF_OOM_DUMP_PATH=/home/pxfoom_segh1
      ```

   2. To turn off heap dumping after you have turned it on, comment out the `PXF_OOM_DUMP_PATH` property setting:

      ```
      #export PXF_OOM_DUMP_PATH=/home/pxfoom_segh1
      ```

5. Save the `pxf-enf.sh` file and exit the editor.

6. Use the `pxf cluster sync` command to copy the updated `pxf-env.sh` file to the Greenplum Database cluster. For example:

   ```
   gpadmin@gpmaster$ pxf cluster sync
   ```

7. Restart PXF on each Greenplum Database segment host as described in Restarting PXF.

Increasing the JVM Memory for PXF

Each PXF agent running on a segment host is configured with a default maximum Java heap size of 2GB and an initial heap size of 1GB. If the segment hosts in your Greenplum Database cluster have an ample amount of memory, try increasing the maximum heap size to a value between 3-4GB. Set the initial and maximum heap size to the same value if possible.

Perform the following procedure to increase the heap size for the PXF agent running on each segment host in your Greenplum Database cluster.

1. Log in to your Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. Edit the `$PXF_CONF/conf/pxf-env.sh` file. For example:

```
gpadmin@gpmaster$ vi $PXF_CONF/conf/pxf-env.sh
```

3. Locate the `PXF_JVM_OPTS` setting in the `pxf-env.sh` file, and update the `-Xmx` and/or `-Xms` options to the desired value. For example:

```
PXF_JVM_OPTS="-Xmx3g -Xms3g"
```

4. Save the file and exit the editor.
5. Use the `pxf cluster sync` command to copy the updated `pxf-env.sh` file to the Greenplum Database cluster. For example:

```
gpadmin@gpmaster$ pxf cluster sync
```

6. Restart PXF on each Greenplum Database segment host as described in Restarting PXF.

Another Option for Resource-Constrained PXF Segment Hosts

If increasing the maximum heap size is not suitable for your Greenplum Database deployment, try decreasing the number of concurrent working threads configured for PXF's underlying Tomcat web application. A decrease in the number of running threads will prevent any PXF node from exhausting its memory, while ensuring that current queries run to completion (albeit a bit slower). Tomcat's default behavior is to queue requests until a thread is free, or the queue is exhausted.

The default maximum number of Tomcat threads for PXF is 200. The `PXF_MAX_THREADS` configuration property controls this setting.

PXF thread capacity is determined by the profile and whether or not the data is compressed. If you plan to run large workloads on a large number of files in an external Hive data store, or you are reading compressed ORC or Parquet data, consider specifying a lower `PXF_MAX_THREADS` value.

Note: Keep in mind that an increase in the thread count correlates with an increase in memory consumption when the thread count is exhausted.

Perform the following procedure to set the maximum number of Tomcat threads for the PXF agent running on each segment host in your Greenplum Database deployment.

1. Log in to your Greenplum Database master node:

```
$ ssh gpadmin@<gpmaster>
```

2. Edit the `$PXF_CONF/conf/pxf-env.sh` file. For example:

```
gpadmin@gpmaster$ vi $PXF_CONF/conf/pxf-env.sh
```

3. Locate the `PXF_MAX_THREADS` setting in the `pxf-env.sh` file. Uncomment the setting and update it to the desired value. For example, to set the maximum number of Tomcat threads to 100:

```
export PXF_MAX_THREADS=100
```

4. Save the file and exit the editor.
5. Use the `pxf cluster sync` command to copy the updated `pxf-env.sh` file to the Greenplum Database cluster. For example:

   ```
   gpadmin@gpmaster$ pxf cluster sync
   ```

6. Restart PXF on each Greenplum Database segment host as described in Restarting PXF.

Addressing PXF JDBC Connector Time Zone Errors

You use the PXF JDBC connector to access data stored in an external SQL database. Depending upon the JDBC driver, the driver may return an error if there is a mismatch between the default time zone set for the PXF server and the time zone set for the external SQL database.

For example, if you use the PXF JDBC connector to access an Oracle database with a conflicting time zone, PXF logs an error similar to the following:

```
SEVERE: Servlet.service() for servlet [PXF REST Service] in context with path [/pxf] t
hrew exception
java.io.IOException: ORA-00604: error occurred at recursive SQL level 1
ORA-01882: timezone region not found
```

Should you encounter this error, you can set default time zone option(s) for the PXF server in the `$PXF_CONF/conf/pxf-env.sh` configuration file, `PXF_JVM_OPTS` property setting. For example, to set the time zone:

```
export PXF_JVM_OPTS="<current_settings> -Duser.timezone=America/Chicago"
```

You can use the `PXF_JVM_OPTS` property to set other Java options as well.

As described in previous sections, you must synchronize the updated PXF configuration to the Greenplum Database cluster and restart the PXF server on each segment host.

PXF Fragment Metadata Caching

A PXF connector *Fragmenter* uses metadata from the external data source to split data into a list of fragments (blocks, files, etc.) that can be read in parallel. PXF caches the fragment metadata on a per-query basis: the first thread to access a fragment's metadata stores the information in a cache, and other threads reuse this cached metadata. Caching of this nature reduces query memory requirements for external data sources with a large number of fragments.

PXF fragment metadata caching is enabled by default. To turn off fragment metadata caching, or to re-enable it after turning it off, perform the following procedure:

1. Log in to your Greenplum Database master node:

   ```
   $ ssh gpadmin@<gpmaster>
   ```

2. Edit the `$PXF_CONF/conf/pxf-env.sh` file. For example:

   ```
   gpadmin@gpmaster$ vi $PXF_CONF/conf/pxf-env.sh
   ```

3. Locate the `PXF_FRAGMENTER_CACHE` setting in the `pxf-env.sh` file. If the setting is commented out, uncomment it, and then update the value. For example, to turn off fragment metadata caching, set the value to `false`:

   ```
   export PXF_FRAGMENTER_CACHE=false
   ```

4. Save the file and exit the editor.
5. Use the `pxf cluster sync` command to copy the updated `pxf-env.sh` file to the Greenplum Database cluster. For example:

   ```
   gpadmin@gpmaster$ pxf cluster sync
   ```

6. Restart PXF on each Greenplum Database segment host as described in Restarting PXF.

About PXF External Table Child Partitions

Greenplum Database supports partitioned tables, and permits exchanging a leaf child partition with a PXF external table.

When you read from a partitioned Greenplum table where one or more partitions is a PXF external table and there is no data backing the external table path, PXF returns an error and the query fails. This default PXF behavior is not optimal in the partitioned table case; an empty child partition is valid and should not cause a query on the parent table to fail.

The `IGNORE_MISSING_PATH` PXF custom option is a boolean that specifies the action to take when the external table path is missing or invalid. The default value is `false`, PXF returns an error when it encounters a missing path. If the external table is a child partition of a Greenplum table, you want PXF to ignore a missing path error, so set this option to `true`.

For example, PXF ignores missing path errors generated from the following external table:

```
CREATE EXTERNAL TABLE ext_part_87 (id int, some_date date)
  LOCATION ('pxf://bucket/path/?PROFILE=s3:parquet&SERVER=s3&IGNORE_MISSING_PATH=true'
)
FORMAT 'CUSTOM' (formatter = 'pxfwritable_import');
```

The `IGNORE_MISSING_PATH` custom option applies only to file-based profiles, including `*:text`, `*:parquet`, `*:avro`, `*:json`, `*:AvroSequenceFile`, and `*:SequenceFile`. This option is *not available* when the external table specifies the `HBase`, `Hive*`, or `Jdbc` profiles, or when reading from S3 using S3-Select.

PXF Utility Reference

The Greenplum Platform Extension Framework (PXF) includes the following utility reference pages:

- pxf cluster
- pxf

pxf cluster

Manage the PXF configuration and the PXF service instance on all Greenplum Database hosts.

Synopsis

```
pxf cluster <command> [<option>]
```

where `<command>` is:

```
help
init
register
reset
restart
start
status
stop
sync
```

Description

The `pxf cluster` utility command manages PXF on the master, standby master, and on all Greenplum Database segment hosts. You can use the utility to:

- Initialize PXF configuration on all hosts in the Greenplum Database cluster.
- Reset the PXF service instance on all hosts to its uninitialized state.
- Start, stop, and restart the PXF service instance on all segment hosts.
- Display the status of the PXF service instance on all segment hosts.
- Synchronize the PXF configuration from the Greenplum Database master host to the standby master and to all segment hosts.
- Copy the extension files from the PXF installation on each host to the Greenplum installation on the host after a Greenplum upgrade.

`pxf cluster` requires a running Greenplum Database cluster. You must run the utility on the

Greenplum Database master host.

(If you want to manage the PXF service instance on a specific segment host, use the `pxf` utility. See `pxf`.)

Commands

help
: Display the `pxf cluster` help message and then exit.

init
: Initialize the PXF service instance on the master, standby master, and on all segment hosts. When you initialize PXF across your Greenplum Database cluster, you must identify the PXF user configuration directory via an environment variable named `$PXF_CONF`. If you do not set `$PXF_CONF` prior to initializing PXF, PXF returns an error.

register
: Copy the PXF extension files from the PXF installation on each host to the Greenplum installation on the host. This command requires that `$GPHOME` be set, and is typically run after you upgrade your Greenplum Database installation.

reset
: Reset the PXF service instance on the master, standby master, and on all segment hosts. Resetting removes PXF runtime files and directories, and returns PXF to an uninitialized state. You must stop the PXF service instance running on each segment host before you reset PXF in your Greenplum Database cluster.

restart
: Stop, and then start, the PXF service instance on all segment hosts.

start
: Start the PXF service instance on all segment hosts.

status
: Display the status of the PXF service instance on all segment hosts.

stop
: Stop the PXF service instance on all segment hosts.

sync
: Synchronize the PXF configuration (`$PXF_CONF`) from the master to the standby master and to all Greenplum Database segment hosts. By default, this command updates files on and copies files to the remote. You can instruct PXF to also delete files during the synchronization; see Options.
  If you have updated the PXF user configuration or added JAR files, you must also restart PXF after you synchronize the PXF configuration.

Options

The `pxf cluster sync` command takes the following option:

–d | ––delete
: Delete any files in the PXF user configuration on the standby master and segment hosts that are not also present on the master host.

Examples

Stop the PXF service instance on all segment hosts:

```
$ pxf cluster stop
```

Synchronize the PXF configuration to the standby and all segment hosts, deleting files that do not exist on the master host:

```
$ pxf cluster sync --delete
```

See Also

`pxf`

pxf

Manage the PXF configuration and the PXF service instance on the local Greenplum Database host.

Synopsis

```
pxf <command> [<option>]
```

where <command> is:

```
cluster
help
init
register
reset
restart
start
status
stop
sync
version
```

Description

The `pxf` utility manages the PXF configuration and the PXF service instance on the local Greenplum Database host.

You can initialize or reset PXF on the master, master standby, or a specific segment host. You can also synchronize the PXF configuration from the master to these hosts.

You can start, stop, or restart the PXF service instance on a specific segment host, or display the status of the PXF service instance running on a segment host.

You can copy the extension files from a PXF installation on the host to the Greenplum installation on the host after a Greenplum upgrade.

(Use the `pxf cluster` command to copy extension files to, initialize, or reset PXF on all hosts,

synchronize the PXF configuration to the Greenplum Database cluster, or to start, stop, or display the status of the PXF service instance on all segment hosts in the cluster.)

Commands

cluster
: Manage the PXF configuration and the PXF service instance on all Greenplum Database hosts. See `pxf cluster`.

help
: Display the `pxf` management utility help message and then exit.

init
: Initialize the PXF service instance on the host. When you initialize PXF, you must identify the PXF user configuration directory via an environment variable named `$PXF_CONF`. If you do not set `$PXF_CONF` prior to initializing PXF, PXF prompts you to accept or decline the default user configuration directory, `$HOME/pxf`, during the initialization process. See Options.

register
: Copy the PXF extension files from the PXF installation on the host to the Greenplum installation on the host. This command requires that `$GPHOME` be set, and is typically run after you upgrade your Greenplum Database installation.

reset
: Reset the PXF service instance running on the host. Resetting removes PXF runtime files and directories, and returns PXF to an uninitialized state. You must stop the PXF service instance running on a segment host before you reset PXF on the host.

restart
: Restart the PXF service instance running on the segment host.

start
: Start the PXF service instance on the segment host.

status
: Display the status of the PXF service instance running on the segment host.

stop
: Stop the PXF service instance running on the segment host.

sync
: Synchronize the PXF configuration (`$PXF_CONF`) from the master to a specific Greenplum Database standby master or segment host. You must run `pxf sync` on the master host. By default, this command updates files on and copies files to the remote. You can instruct PXF to also delete files during the synchronization; see Options.

version
: Display the PXF version and then exit.

Options

The `pxf init` command takes the following option:

–y

Do not prompt, use the default `$PXF_CONF` directory location if the environment variable is not set.

The `pxf reset` command takes the following option:

–f | ––force
: Do not prompt before resetting the PXF service instance; reset without user interaction.

The `pxf sync` command, which you must run on the Greenplum Database master host, takes the following option and argument:

–d | ––delete
: Delete any files in the PXF user configuration on `<gphost>` that are not also present on the master host. If you specify this option, you must provide it on the command line before `<gphost>`.

<gphost>
: The Greenplum Database host to which to synchronize the PXF configuration. Required. `<gphost>` must identify the standy master host or a segment host.

Examples

Start the PXF service instance on the local segment host:

```
$ pxf start
```

See Also

`pxf cluster`