PXF for WarehousePG

Version 6

Built at 2026-08-12T13:11:59

1 PXF for WarehousePG

The WarehousePG Platform Extension Framework (PXF) gives WarehousePG (WHPG) access to data that lives outside the cluster, in systems like Hadoop, object stores, and external SQL databases. You interact with that external data through external tables, using a readable external table to query it and a writable external table to insert into it, the same way you'd work with any other WHPG table.

A `pxf` extension and a PXF service run on the coordinator, standby coordinator, and every segment host. When a query touches a PXF external table, each segment's PXF service connects to the external source independently and reads or writes its share of the data in parallel, so a PXF query scales with your cluster rather than routing through the coordinator. See Architecture for how these pieces fit together.

Key features

- **Broad connector support.** Reach Hadoop, object stores such as Amazon S3, MinIO, Azure Blob Storage, and Google Cloud Storage, SQL databases over JDBC, and network file systems.
- **Multiple data formats.** Read and write text, CSV, Avro, JSON, RCFile, Parquet, SequenceFile, and ORC, depending on the connector. See Compatibility for supported sources, formats, and versions.
- **Parallel, segment-based execution.** Each segment's PXF service handles its own portion of the data independently, so performance scales with the number of segments in your cluster.
- **Full read and write support.** Query external data or insert into it using PXF external tables.
- **Cluster-wide management.** The `pxf cluster` command line tool syncs configuration and starts or stops the PXF service across every host from one place. See Managing the PXF cluster for details.

2 PXF release notes

The PXF documentation describes the latest version of PXF for WarehousePG, including minor releases and patches.

Version	Release date
6.10.2	26 Feb 2026

2.1 PXF 6.10.2 release notes

PXF 6.10.2

Released: 26 Feb 2026

Enhancements

- Added `TIMESTAMPTZ` support to the JDBC profile.
- Replaced the `adl` profile with `abfss`, for Azure Data Lake Storage Gen2 support.

Bug fixes

- Fixed the `pxf.control` and `pxf_fdw.control` extension files installing to the PXF package directory instead of `$GPHOME/share/postgresql/extension`, where WHPG expects them. The package now installs both files to the correct location directly, so `pxf cluster register` is no longer required after a default installation.

3 Overview of PXF for WarehousePG

Data managed by your organization often lives outside your WarehousePG (WHPG) cluster, in systems like Hadoop, object stores, or other SQL databases. The WarehousePG Platform Extension Framework (PXF) gives you access to that external data through built-in connectors that map an external data source to a WHPG table definition. You read and write external data through PXF the same way you query or insert into a WHPG external table, using standard SQL.

Supported connectors and formats

PXF connects to external data through profiles, which pair a connector, such as an object store or Hadoop, with a data format. Supported connectors include:

- Object stores compatible with the S3 API, including Amazon S3 and MinIO
- Hadoop (HDFS)
- Hive
- HBase
- SQL databases, through the Java Database Connectivity (JDBC) connector

Supported data formats include text, CSV, Avro, JSON, RCFile, Parquet, SequenceFile, and ORC, depending on the connector.

PXF supports both reading external data into WHPG and writing WHPG data out to external storage, through readable and writable external tables respectively.

Explore PXF

- Architecture: Understand the PXF components and how a query reaches external data.
- Compatibility: Check supported WHPG versions, platforms, and PXF versions.
- Known issues: Review current limitations and workarounds.

3.1 PXF for WarehousePG architecture

The WarehousePG Platform Extension Framework (PXF) architecture consists of an extension registered inside WarehousePG (WHPG), and a Java service that runs alongside WHPG on the coordinator, standby coordinator, and every segment host. You interact with PXF through external tables, using a readable external table to query external data and a writable external table to insert into it.

Components

The PXF architecture consists of three main components: the `pxf` extension inside WHPG and the PXF service, both running on every host, and the `pxf cluster` CLI, which you run from a shell to manage the other two.

The pxf extension

You create the `pxf` extension inside each database that needs external table access. It implements the `pxf://` protocol, and its role differs by host when a query scans a PXF external table. On the coordinator (cdw), it only takes part in planning, working out which filters and columns to push down and estimating cost for the query plan, without contacting the PXF service or the external source. On each segment (sdw), the extension executes its share of that plan, reading the table's connector, server, and path details and forwarding the request to the local PXF service.

PXF service

The PXF service is a long-running Java (JVM) process that runs on the coordinator, standby coordinator, and every WHPG segment host. It listens on a local port (5888 by default) for requests from the `pxf` extension. Each PXF service instance reads its own local server configuration, which must be kept in sync across the cluster.

The pxf cluster CLI

The `pxf` and `pxf cluster` commands are a separate command-line tool for managing the PXF service across every host. Running one of its commands connects to WHPG to look up the cluster's hosts, then reaches each one over SSH to copy configuration or start and stop its local PXF service. This CLI tool doesn't take part in query execution, and it doesn't depend on the PXF service running on the host you invoke it from, so it works even if that host's PXF service is down.

Key concepts

You connect to external data through a PXF external table, which brings together a few pieces of configuration to tell PXF what to read and how to reach it.

- **External table**: The WHPG table you use to read or write external data through PXF. Create a readable or writable external table with a `pxf://` location, the same way as any other WHPG external table.
- **Connector**: PXF's implementation for reading from and writing to a specific kind of external data store, such as Hadoop, an S3-compatible object store, or a JDBC-accessible database. See Object stores, Hadoop, Connecting to SQL databases over JDBC, and Network file system for the connectors PXF supports.
- **Server**: A named configuration for a connector, telling PXF how to reach a specific external source, including details such as the endpoint URL and credentials. A table's `SERVER` setting determines which connection settings its PXF requests use. See Understanding PXF servers.
- **Profile**: A named mapping of a connector to a data format, for example `s3:parquet`. A table's `PROFILE` setting determines which connector and format code handles the request. See Understanding PXF profiles.

An external table's `LOCATION` clause specifies both a profile and a server:

```
CREATE EXTERNAL TABLE sales (id int, name text, amount
numeric)
    LOCATION ('pxf://data/sales.csv?
PROFILE=s3:text&SERVER=example')
    FORMAT 'CSV'
(delimiter=',');
```

Query execution flow

A query against a PXF external table follows the same path on every segment, from the `pxf` extension out to the external source and back.

1. A query against a PXF external table reaches the WHPG segments as part of the query's overall execution plan.
2. On each segment, the `pxf` extension forwards the relevant portion of the request to the local PXF service, including the profile, server name, and any parameters from the `LOCATION` clause.
3. Each PXF service independently connects to the external source, such as an object store, Hadoop cluster, or external database, using the server configuration, and reads or writes its portion of the data.
4. For reads, results stream back through the `pxf` extension into the query's execution plan on that segment. For writes, the extension confirms the PXF service accepted the data before the operation completes on that segment.

Because each segment's PXF service handles its own portion of the data in parallel, a PXF query scales with the number of segments in your cluster rather than routing through a single coordinator process.

Filter pushdown

When you query a PXF external table with a `WHERE` clause, PXF can push part of that clause down to the external data source instead of transferring every row to WHPG and filtering there. WHPG controls pushdown through the `gp_external_enable_filter_pushdown` server configuration parameter. See Compatibility to check which operators and data types support it. PXF falls back to filtering in WHPG for any part of the clause the connector or profile doesn't push down.

Column projection

PXF also applies column projection automatically. With column projection, PXF asks the external data source for only the columns your query selects, rather than transferring every column and discarding the rest in WHPG. See Compatibility to check which connectors and profiles support it.

3.2 PXF for WarehousePG compatibility

RHEL 7, RHEL 8, or RHEL 9.

System requirements

- Java 8 or Java 11 on every host in the cluster.

External system compatibility

PXF bundles a client library for each connector. These versions determine which external system versions PXF can reach.

Connector	Compatible version
HDFS	Hadoop 2.10.x
Hive	Hive 2.3.x metastore
HBase	HBase 1.3.x
Amazon S3 and other S3-compatible object stores, including MinIO	S3 API, through the AWS SDK for Java 1.12.x
Azure Blob Storage and Azure Data Lake Storage Gen2	Bundled with PXF's Hadoop 2.10.x client
Google Cloud Storage	GCS connector 1.9.x
SQL databases, through JDBC	Any JDBC 4.x-compliant driver you provide. PXF bundles the PostgreSQL driver, currently 42.7.x, by default.

Operations by connector

Connector	Read	Write	Supported formats
HDFS	Yes	Yes	Delimited text, CSV, fixed-width, Parquet, ORC, Avro, JSON, SequenceFile, and Avro in a SequenceFile
Object stores (S3, Azure, Google Cloud Storage)	Yes	Yes	Same formats as HDFS
Network file system	Yes	Yes	Delimited text, CSV, fixed-width, Parquet, ORC, Avro, and JSON
Hive	Yes	No	Whatever format the Hive table itself uses, delimited text, SequenceFile, RCFile, ORC, Parquet, or Avro
HBase	Yes	No	Not applicable, HBase's own column-family model
SQL databases, through JDBC	Yes	Yes	Not applicable, the external database's native column types

Filter pushdown support

Profile	Comparisons (`<` , `>` , `<=` , `>=` , `=` , `<>`)	`LIKE`	`IS [NOT] NULL`	`IN`	`AND` / `OR`	`NOT`
`jdbc`	Yes	Yes[4]	Yes	No	Yes	Yes
`*:parquet`	Yes[1]	No	Yes[1]	Yes[1]	Yes[1]	Yes[1]

Profile	Comparisons (`<` , `>` , `<=` , `>=` , `=` , `<>`)	`LIKE`	`IS [NOT] NULL`	`IN`	`AND` / `OR`	`NOT`
`*:orc` , except `hive:orc`	Yes[1] [3]	No	Yes[1] [3]	Yes[1] [3]	Yes[1] [3]	Yes[1] [3]
`s3:parquet` and `s3:text` with S3 Select	Yes	No	Yes	Yes	Yes	Yes
`hbase`	Yes	No	Yes	No	Yes	No
`hive:text` , `hive` on a text-format table	Yes[2]	No	No	No	Yes[2]	No
`hive:rc` , `hive` on an RCFile table	Yes[2]	No	Yes	Yes	Yes[2]	Yes
`hive:orc` , `hive` on an ORC table	Yes[2]	No	Yes	Yes	Yes[2]	Yes
`hive` on a Parquet table	Yes[2]	No	No	Yes	Yes[2]	Yes
`hive:orc` with `VECTORIZE=true`	Yes[2]	No	No	No	Yes[2]	No

- [1] PXF applies the predicate itself instead of the remote system. Doing so doesn't reduce the data transferred over the network, but it does reduce how much of that data PXF holds in memory at once while processing the result.
- [2] PXF supports partition pruning based on partition keys.
- [3] PXF bases filtering on file-level, stripe-level, and row-level ORC statistics.
- [4] The `jdbc` profile supports `LIKE` only for `TEXT` columns.

PXF doesn't support filter pushdown for any other profile, including `*:avro` , `*:AvroSequenceFile` , `*:SequenceFile` , `*:json` , `*:text` , `*:csv` , `*:fixedwidth` , and `*:text:multi` .

Supported data types for pushdown include `INT2` , `INT4` , `INT8` , `CHAR` , `TEXT` , `VARCHAR` , `FLOAT` , `BOOL` , and `NUMERIC` , except with `hive` on a Parquet table. `DATE` and `TIMESTAMP` push down only with the JDBC connector, `hive:rc` , `hive:orc` , and `hive` on an RCFile or ORC table.

Column projection support

Data source	Profiles
SQL database, through JDBC	`jdbc`
Hive	`hive` on a text, Parquet, RCFile, or ORC table, `hive:rc` , `hive:orc`
HDFS	`hdfs:orc` , `hdfs:parquet`
Network file system	`file:orc` , `file:parquet`
Object stores (S3, Azure, Google Cloud Storage)	`s3:orc` , `s3:parquet` , `wasbs:orc` , `wasbs:parquet` , `abfss:orc` , `abfss:parquet` , `gs:orc` , `gs:parquet`

PXF may still turn projection off for a specific query if it can't serialize the query's filter, for example when a `WHERE` clause resolves to a `boolean` column rather than a comparison.

3.3 Known issues

This release includes the following known issues and limitations. Where applicable, workarounds are included to help mitigate the impact. These issues are actively tracked and are planned for resolution in a future release.

- If `fs.s3a.endpoint` in a server's site XML file doesn't include the URI scheme (`http://` or `https://`), PXF fails with a connection error such as `Unsupported or unrecognized SSL message`, and the PXF logs don't show additional detail. Always include the scheme in the endpoint value. See S3-compatible object stores.

- Third-party JDBC driver JARs placed in the global `$PXF_BASE/lib` directory can cause classloader conflicts with libraries PXF already ships. Place them in the server-specific directory (`$PXF_BASE/servers/<server_name>/`) instead, so PXF loads them in an isolated classloader.

4 Installing PXF for WarehousePG

Install PXF on every host in your WarehousePG (WHPG) cluster, including the coordinator, standby coordinator, and all segment hosts.

Prerequisites

- WarehousePG installed on every host in the cluster, running on RHEL 7 (WHPG 6.x only), 8, or 9.
- Java 8 or 11 installed on every host in the cluster.

Installing PXF

Install the PXF package on the coordinator, standby coordinator, and all segment hosts. Run these steps as `gpadmin` on the coordinator.

1. On the coordinator, download the package from the EDB repository:

   ```
   export EDB_SUBSCRIPTION_TOKEN=<your-token>
   curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/gpsupp/setup.rpm.sh" | sudo
   -E bash
   sudo dnf download edb-whpg<whpg_major_version>-
   pxf-<version>
   ```

   Where `<whpg_major_version>` is your WHPG version, `6` or `7`.

2. Create a file `all_hosts` listing the coordinator, standby coordinator, and all segment hosts:

   ```
   cdw
   scdw
   sdw1
   sdw2
   sdw3
   ```

3. If you haven't already, install Java on every host. For example, to install Java 8:

   ```
   gpssh -f all_hosts -u gpadmin -e "sudo dnf -y install java-1.8.0-openjdk-
   devel"
   ```

4. Copy the package to all hosts in the WHPG cluster, replacing `<el_version>` with your OS version:

```
```bash
gpsync -f all_hosts edb-whpg7-pxf-<version>.<el_version>.x86_64.rpm =:/tmp/
```
```

```
```bash
gpscp -f all_hosts edb-whpg6-pxf-<version>.<el_version>.x86_64.rpm =:/tmp/
```
```

1. Install the package on all hosts:

   ```
   gpssh -f all_hosts -u gpadmin -e "sudo dnf install -y /tmp/edb-whpg<whpg_major_version>-pxf-
   <version>.<el_version>.x86_64.rpm"
   ```

   The installation creates `/usr/local/edb-whpg<whpg_major_version>-pxf`. You'll set this variable as the `$PXF_HOME` environment variable when Configuring and starting PXF.

2. Set ownership of `$PXF_HOME` on every host, so `gpadmin` can run PXF:

   ```
   gpssh -f all_hosts -u gpadmin -e "sudo chown -R gpadmin:gpadmin /usr/local/edb-
   whpg<whpg_major_version>-pxf"
   ```

Next steps

See Configuring and starting PXF to set up and start the PXF service across your cluster. If you plan to query through `pxf_fdw` instead of `pxf://` external tables, see Using the foreign data wrapper for its own package requirements.

5 Configuring and starting PXF for WarehousePG

After you install PXF on every host in your WarehousePG (WHPG) cluster, set up its environment, initialize and start the service, and create the `pxf` extension.

Setting environment variables

Set `$PXF_HOME` to the directory created during installation, so PXF and your shell can find the PXF service and its default configuration templates. PXF also needs a separate, writable runtime directory, `$PXF_BASE`, where your per-cluster configuration, server definitions, and logs live. Keeping `$PXF_BASE` separate from `$PXF_HOME` means a PXF upgrade doesn't overwrite your configuration.

On the coordinator, set `PXF_HOME`, `PXF_BASE`, and add the PXF `bin` directory to your `PATH`:

```
export PXF_HOME=/usr/local/pxf
export PXF_BASE=$HOME/pxf-base
export PATH="$PXF_HOME/bin:$PATH"
```

Add these lines to `~/.bashrc` on the coordinator so they persist across sessions.

Initializing and starting PXF

Set up PXF's runtime configuration across the cluster and start the service, so PXF is ready to handle queries. Run the following commands from the coordinator host. Each `pxf cluster` command applies the action to every host in the cluster.

1. Create the runtime configuration directory on every host:

   ```
   pxf cluster prepare
   ```

   This command creates `$PXF_BASE` on every host and copies the default configuration templates from `$PXF_HOME/conf` into it. If `$PXF_BASE` doesn't already exist, `pxf cluster prepare` creates it for you.

2. Set `JAVA_HOME` in `$PXF_BASE/conf/pxf-env.sh` on the coordinator:

   ```
   echo "export JAVA_HOME=$(readlink -f $(which java) | sed 's:/bin/java$::')" >> $PXF_BASE/conf/pxf-env.sh
   ```

3. Sync the change to every host:

   ```
   pxf cluster sync
   ```

4. Start the PXF Java service on every host, listening on port 5888 by default:

   ```
   pxf cluster start
   ```

5. Confirm PXF is running on every host:

```
pxf cluster
status
```

```
output
Checking status of PXF servers on coordinator host and 2 segment hosts...
PXF is running on 3 out of 3 hosts
```

Creating the PXF extension

Create the `pxf` extension in each database that needs external table access. `pxf cluster register`, run as the last step of installing PXF, already placed the extension's control, SQL, and shared library files under `$GPHOME` on every host, so you don't need to repeat it here.

1. Connect to the target database and create the extension:

```
CREATE EXTENSION IF NOT EXISTS pxf;
```

If the extension already exists from a previous PXF version, update it instead:

```
ALTER EXTENSION pxf
UPDATE;
```

Repeat this step in every database where you want to query external tables.

2. Grant roles access. Only WHPG superusers can use the `pxf` protocol by default, so grant `SELECT` to let a role read external tables through PXF, and `INSERT` to let it write them:

```
GRANT SELECT ON PROTOCOL pxf TO
<role_name>;
GRANT INSERT ON PROTOCOL pxf TO
<role_name>;
```

> **Note**
>
> If you need to drop the `pxf` extension, use `DROP EXTENSION pxf CASCADE` to drop it along with any external table still using the `pxf` protocol.

Next steps

See Managing the PXF cluster for the PXF configuration files and ongoing cluster management commands, or go straight to configuring a connector, such as Object stores or Hadoop, to reach an external data source.

6 Managing the PXF cluster

Once PXF is running, you manage it from the coordinator host using the `pxf cluster` commands.

Understanding PXF directories and configuration files

PXF uses two main directories:

- `$PXF_HOME` is the read-only directory where PXF is installed, for example `/usr/local/pxf`. It holds the PXF service itself, along with the connector site templates under `templates/` that you copy into a server configuration.
- `$PXF_BASE` is the writable runtime directory you create when you initialize your cluster, for example `$HOME/pxf-base`. It holds your configuration, including your server configurations under `servers/`. Keeping `$PXF_BASE` separate from `$PXF_HOME` means a PXF upgrade doesn't overwrite your configuration.

See Connecting to external data for how PXF servers and profiles work.

Relocating $PXF_BASE

Move `$PXF_BASE` to a new location if you're running low on disk space or reorganizing storage across hosts:

1. Stop PXF on every host:

   ```
   pxf cluster stop
   ```

2. On the coordinator, move `$PXF_BASE` to the new location and update the environment variable, including in `~/.bashrc`:

   ```
   mv $PXF_BASE /new/path/to/pxf-base
   export PXF_BASE=/new/path/to/pxf-base
   ```

3. Sync the relocated directory out to every segment host, then start PXF:

   ```
   pxf cluster sync
   pxf cluster start
   ```

4. Once PXF is confirmed running from the new location, clean up the old `$PXF_BASE` directory on each segment host.

Starting, stopping, and restarting PXF

Start, stop, or restart PXF across the whole cluster with a single command, rather than repeating it on every host. Only the `gpadmin` user can run these commands:

```
pxf cluster start
pxf cluster stop
pxf cluster restart
```

Applying a configuration change

Sync any change to a file under `$PXF_BASE/conf`, or to a server configuration, out to every segment host, then restart PXF. For example, to increase the PXF Java heap size:

1. Edit `PXF_JVM_OPTS` in `$PXF_BASE/conf/pxf-env.sh` on the coordinator:

   ```
   export PXF_JVM_OPTS="-Xmx4g -
   Xms2g"
   ```

2. Sync the change to every segment host:

   ```
   pxf cluster sync
   ```

3. Restart PXF to apply it. Changes made only on the coordinator have no effect until you sync and restart:

   ```
   pxf cluster restart
   ```

Configuring advanced settings

Adjust settings under `$PXF_BASE/conf` to fit your environment, the same way as any other configuration change:

- Change the listen address, overall log level, or thread limit by editing `pxf-application.properties`.
- Change the JVM heap size by editing `pxf-env.sh`.
- Set the hostname or port WHPG uses to reach PXF by setting the `PXF_HOST` and `PXF_PORT` environment variables in `~/.bashrc` on the relevant host. This change also requires restarting WHPG itself, not just PXF.
- Fine-tune logging for individual loggers by editing `pxf-log4j2.xml`.
- Define a custom PXF profile in `pxf-profiles.xml`, pairing a profile name with Java classes that implement PXF's plugin interfaces to read or write it.

See PXF configuration files for the full list of properties in each file.

Monitoring PXF

`pxf cluster status` confirms PXF is both running and responsive on every host:

```
pxf cluster
status
```

PXF also exposes HTTP endpoints on each host for monitoring tools, at `http://<host>:5888/<endpoint>`:

Endpoint	Description
`actuator/health`	The status of the PXF service.
`actuator/info`	Build information for the PXF service.
`actuator/metrics`	JVM, Tomcat, system, and PXF-specific metrics.
`actuator/prometheus`	The same metrics, in a format a Prometheus server can scrape.

7 Querying data with the PXF foreign data wrapper

Query a PXF-connected data source through standard Postgres foreign tables, using the PXF foreign data wrapper `pxf_fdw`, instead of `pxf://` external tables. `pxf_fdw` gives you `CREATE SERVER`, `CREATE USER MAPPING`, and `CREATE FOREIGN TABLE` statements for the same connectors. Both interfaces read and write through the same server configurations, so switching between them changes only the SQL objects and grants you use, not the underlying data source setup.

> **Note**
>
> `pxf_fdw` is available only for WHPG 7 and later.

Enabling the extension

`pxf_fdw` runs through the same PXF service as `pxf://` external tables, so complete Installing PXF first, then Configuring and starting PXF through Initializing and starting PXF. You don't need the Creating the PXF extension step that follows, since `pxf_fdw` is its own extension.

Like `pxf`, `pxf_fdw` registers per database. `pxf cluster register` already installs the extension files on every host, so you only need to create it in each database where you want to use it:

```
CREATE EXTENSION
pxf_fdw;
```

Available foreign data wrappers

`pxf_fdw` provides one foreign data wrapper per PXF connector:

Foreign data wrapper	Connector
`hdfs_pxf_fdw`	HDFS
`hive_pxf_fdw`	Hive
`hbase_pxf_fdw`	HBase
`s3_pxf_fdw`	Amazon S3 and other S3-compatible object stores
`gs_pxf_fdw`	Google Cloud Storage
`abfss_pxf_fdw`	Azure Data Lake Storage Gen2
`wasbs_pxf_fdw`	Azure Blob Storage
`jdbc_pxf_fdw`	External SQL databases through JDBC
`file_pxf_fdw`	Local file storage

Creating a server

A foreign data wrapper server points at the same `$PXF_BASE/servers/<server_name>` directory you'd configure for external tables. See Object stores, Hadoop, JDBC, or Network file system for how to create and populate that directory for your connector. Once it exists, reference it with the `config` option:

```
CREATE SERVER
hdfs_hdp
    FOREIGN DATA WRAPPER hdfs_pxf_fdw
    OPTIONS (config 'hdfssrvcfg');
```

`config` names the server directory, `$PXF_BASE/servers/hdfssrvcfg` in this example, not the FDW server name itself.

Creating a user mapping

A user mapping grants a role permission to use a server. Add connector credentials here instead of in the server's site XML file if you want them to be user-specific rather than shared:

```
CREATE USER MAPPING FOR gpadmin SERVER
hdfs_hdp;
```

For example, an S3 user mapping can carry that user's own access and secret keys:

```
CREATE USER MAPPING FOR francisco
    SERVER
s3_hdp
    OPTIONS (accesskey 'FRANCISCOS_AWS_ACCESS_KEY', secretkey 'FRANCISCOS_AWS_SECRET_KEY');
```

Creating and querying a foreign table

`CREATE FOREIGN TABLE` takes a `resource` option instead of a `LOCATION` clause. `resource` means the same path or identifier you'd use in a `pxf://` location, an HDFS path, an `<hive_database>.<hive_table>` pair, an object store bucket and key, and so on, depending on the connector. An optional `format` option selects the data format, using the same suffixes documented in PXF profiles, for example `format 'parquet'` for `hdfs:parquet`. Omit `format` for delimited text.

```
CREATE FOREIGN TABLE hdfs_data (id int, name
text)
    SERVER
hdfs_hdp
    OPTIONS (resource '/data/pxf_data', format 'csv', delimiter
',');

SELECT * FROM hdfs_data;
```

Granting access

Foreign tables use Postgres's standard FDW grants rather than the `GRANT ... ON PROTOCOL pxf` syntax described in Managing the PXF cluster. A role needs `USAGE` on the foreign server, its own user mapping, and the usual table-level grant:

```
GRANT USAGE ON FOREIGN SERVER hdfs_hdp TO
<role_name>;
GRANT SELECT ON hdfs_data TO <role_name>;
```

8 Connecting to external data

Reach external data from WarehousePG (WHPG) by pointing a PXF external table at a server, for connection details, and a profile, for the connector and data format to read or write it with. Understand how servers, profiles, and formats work before you configure one for the connector you need, on Object stores, Hadoop, Connecting to SQL databases over JDBC, or Network file system.

Every connector, regardless of the external source it reaches, relies on the same two pieces of configuration: a server for connection details and a profile for the data format, such as Parquet, Avro, or JSON. An external table's `LOCATION` clause references both:

```
CREATE EXTERNAL TABLE sales (id int, name text, amount
numeric)
    LOCATION ('pxf://data/sales.csv?
PROFILE=s3:text&SERVER=example')
    FORMAT 'CSV'
(delimiter=',');
```

Understanding PXF servers

PXF connects to an external data source through a named server configuration to read or write data. Each server is a directory under `$PXF_BASE/servers` that contains one or more site XML files with the connection settings for that source, such as the endpoint URL and credentials.

When you initialize your cluster, `pxf cluster prepare` creates an empty `default` server. To read and write from several sources, create multiple servers by adding a separate subdirectory for each one under `$PXF_BASE/servers`, with its own site XML files. `$PXF_HOME/templates` holds a sample site XML file for each connector, ready to copy into your server directory and edit. See Configuration templates for the properties in each.

> **Note**
>
> If `LOCATION` omits `SERVER`, PXF falls back to `$PXF_BASE/servers/default`.

Understanding PXF profiles

A PXF profile name follows a `<connector>:<format>` pattern, for example `hdfs:parquet` or `s3:avro`, and PXF uses it to pick which connector and format code handles the request.

PXF ships with a built-in profile for each connector and format combination it supports, compiled into the PXF service. See PXF profiles for the full list.

PXF also supports custom profiles, defined in `pxf-profiles.xml` and backed by your own Java plugin classes.

The `format` part of a profile works the same way regardless of which connector it's paired with. PXF supports text, CSV, Parquet, ORC, Avro, JSON, and SequenceFile, depending on the connector. See PXF profiles for the full list of formats and which connectors support each.

Choosing a connector

Configure the connector that matches your external data source.

- Object stores: Amazon S3, MinIO, Azure Blob Storage, Azure Data Lake Storage, and Google Cloud Storage.
- Hadoop: HDFS, Hive, and HBase.
- Connecting to SQL databases over JDBC: external SQL databases through a JDBC driver.
- Network file system: a shared network file system.

8.1 Connecting to Hadoop

PXF's Hadoop connectors, the Hadoop Distributed File System (HDFS), Hive, and HBase, share the same underlying cluster configuration. Configure the connection to HDFS first if you also plan to use Hive, since Hive builds on top of it.

- HDFS: Read and write files directly on HDFS.
- Hive: Read Hive table data.
- HBase: Read HBase table data.
- Authenticating with Kerberos: Connect to a Kerberized Hadoop cluster and configure user impersonation.

8.1.1 Connecting to HBase

Configure a server for HBase, then query its table data through external tables. PXF only supports reading from HBase, not writing to it.

Configuring the server

Create a server that connects to an HBase cluster.

> **Note**
>
> If you plan to filter on HBase external tables, copy `$PXF_HOME/share/pxf-hbase-*.jar` to every node in the HBase cluster first, and add its location to `$HBASE_CLASSPATH`. HBase's region servers need this JAR to execute a pushed-down filter.

1. Create a server directory under `$PXF_BASE/servers`, and copy the `hbase-site.xml` template into it. For example, to configure a server named `hbase_server`:

   ```
   mkdir $PXF_BASE/servers/hbase_server
   cp $PXF_HOME/templates/hbase-site.xml $PXF_BASE/servers/hbase_server
   ```

2. Edit `hbase-site.xml` with your connection details:

   ```
   <?xml version="1.0" encoding="UTF-8"?
   >
   <configuration>

   <property>

   <name>hbase.rootdir</name>
           <value>hdfs://<namenode_host>:
   <namenode_port>/hbase</value>
       </property>

   <property>

   <name>hbase.zookeeper.quorum</name>
           <value><zookeeper_host>
   </value>
       </property>
   </configuration>
   ```

   Where:

   - `hbase.rootdir` is the location on HDFS where HBase stores its data.
   - `hbase.zookeeper.quorum` is the ZooKeeper ensemble HBase uses for coordination.

   See Configuration templates for the full list of `hbase-site.xml` properties.

3. Sync the change to every segment host, then restart PXF to apply it:

   ```
   pxf cluster sync
   pxf cluster restart
   ```

Reading data

PXF maps each HBase column to a `<column_family>:<qualifier>` column name. The following example reads an HBase table named `employees`, with `personal` and `job` column families, using the `hbase_server` server:

```
CREATE EXTERNAL TABLE employee_hbase ("personal:name" text, "job:department" text, "job:salary"
int)
    LOCATION ('pxf://employees?
PROFILE=hbase&SERVER=hbase_server')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');

SELECT * FROM employee_hbase;
```

WarehousePG (WHPG) caps column names at 63 characters and requires them to be character-based, while an HBase qualifier name can be longer or contain binary data. When a qualifier doesn't fit those rules, map it indirectly through a lookup table instead. Create a table named `pxflookup` in HBase with a single column family named `mapping`, then add a row per mapped qualifier, using the target HBase table's name as the row key and the WHPG column alias as the qualifier under `mapping`:

```
create 'pxflookup', 'mapping'
put 'pxflookup', 'employees', 'mapping:name', 'personal:name'
put 'pxflookup', 'employees', 'mapping:dept', 'job:department'
```

Reference the aliases directly as column names when you create the external table:

```
CREATE EXTERNAL TABLE employee_hbase_mapped (name text, dept
text)
    LOCATION ('pxf://employees?
PROFILE=hbase&SERVER=hbase_server')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');
```

Add a `recordkey bytea` column to also read the row's key, HBase's unique per-row identifier that isn't stored in any column family. Type it `text` instead of `bytea` if you want to filter on it in a `WHERE` clause and have that filter pushed down to HBase.

8.1.2 Connecting to HDFS

Configure a server for the Hadoop Distributed File System (HDFS), then read and write its data through external tables. If you also plan to use Hive, configure HDFS first, since Hive builds on this same connection.

Configuring the server

Create a server that connects to HDFS.

1. Create a server directory under `$PXF_BASE/servers`, and copy the `core-site.xml` template from `$PXF_HOME/templates` into it. For example, to configure a server named `hdfssrvcfg`:

   ```
   mkdir -p $PXF_BASE/servers/hdfssrvcfg
   cp $PXF_HOME/templates/core-site.xml $PXF_BASE/servers/hdfssrvcfg
   ```

2. Edit `fs.defaultFS` in that file to point to your HDFS NameNode:

   ```
   <property>

   <name>fs.defaultFS</name>
       <value>hdfs://<namenode_host>:<namenode_port>
   </value>
   </property>
   ```

   See Configuration templates for the full list of `core-site.xml` properties.

3. Sync the change to every segment host, then restart PXF to apply it:

   ```
   pxf cluster sync
   pxf cluster restart
   ```

Reading data

Read data from HDFS by creating a readable external table with the profile for its format and the server you configured. For example, to read a CSV file using the `hdfssrvcfg` server:

```
CREATE EXTERNAL TABLE pxf_hdfs_example (id int, name text, age
int)
    LOCATION ('pxf://<hdfs_path>/data.csv?
PROFILE=hdfs:csv&SERVER=hdfssrvcfg')
    FORMAT 'CSV'
(delimiter=',');

SELECT * FROM pxf_hdfs_example;
```

PXF also supports structured formats like Parquet, through the same profile-based syntax:

```
CREATE EXTERNAL TABLE pxf_parquet_read (id int, name
text)
    LOCATION ('pxf://parquet_data?
PROFILE=hdfs:parquet&SERVER=hdfssrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');

SELECT * FROM pxf_parquet_read;
```

See PXF profiles for the full list of supported formats, including worked examples of Avro, JSON, and multi-byte delimiters.

Writing data

Create a writable external table with the `pxfwritable_export` formatter to write WHPG data out to HDFS, then a separate readable external table at the same location to query it back:

```
CREATE WRITABLE EXTERNAL TABLE pxf_parquet_write (id int, name
text)
    LOCATION ('pxf://parquet_data?
PROFILE=hdfs:parquet&SERVER=hdfssrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_export');

INSERT INTO pxf_parquet_write VALUES (1, 'New
York');
```

```
CREATE EXTERNAL TABLE pxf_parquet_read_back (id int, name
text)
    LOCATION ('pxf://parquet_data?
PROFILE=hdfs:parquet&SERVER=hdfssrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');

SELECT * FROM pxf_parquet_read_back;
```

The same pattern applies to other connectors, using their own profile prefix, for example `s3:parquet`. See Object stores for an example.

Next steps

With HDFS configured, you're ready to connect to Hive, which builds on this same connection, or HBase. If your cluster uses Kerberos, see Authenticating with Kerberos.

8.1.3 Connecting to Hive

Configure a server for Hive, then query its table data through external tables. PXF only supports reading from Hive, not writing to it. Configure the Hadoop Distributed File System (HDFS) first, since Hive builds on that same connection.

Configuring the server

Hive tables live on HDFS, so PXF reuses the server you already configured for HDFS, for example `hdfssrvcfg` from HDFS. Add `hive-site.xml` to that same server directory.

1. Copy `hive-site.xml` from the templates directory into the server:

   ```
   cp $PXF_HOME/templates/hive-site.xml $PXF_BASE/servers/hdfssrvcfg
   ```

2. Edit `hive-site.xml` to point to your Hive metastore:

   ```
   <?xml version="1.0" encoding="UTF-8"?
   >
   <configuration>

   <property>

   <name>hive.metastore.uris</name>

   <value>thrift://<metastore_host>:9083</value>
       </property>
   </configuration>
   ```

   `hive.metastore.uris` is the Thrift URI PXF uses to reach your Hive metastore. See Configuration templates for the full list of `hive-site.xml` properties.

3. Sync the change to every segment host, then restart PXF to apply it:

   ```
   pxf cluster sync
   pxf cluster restart
   ```

Reading data

Read a Hive table by creating a readable external table with the `hive` profile and the server you configured. For example, to read a table named `employee` in the `hr` Hive database, using the `hdfssrvcfg` server:

```
CREATE EXTERNAL TABLE emp_hive (id int, name
text)
    LOCATION ('pxf://hr.employee?
PROFILE=hive&SERVER=hdfssrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');

SELECT * FROM emp_hive;
```

The path in `LOCATION` is `<hive_database>.<hive_table>`.

The `hive` profile works with any format Hive supports, reading through Hive's SerDe (serializer/deserializer), the class Hive itself uses to parse that format. For a table stored in text, RCFile, or ORC, switch to the matching `hive:<format>` profile instead for faster reads that skip the SerDe and go straight to the files.

`hive:orc` also supports `VECTORIZE=true`, to read up to 1024 rows at once. Vectorized reads don't support complex types or the `timestamp` data type:

```
CREATE EXTERNAL TABLE emp_hive_orc_vectorized (id int, name
text)
    LOCATION ('pxf://hr.employee?
PROFILE=hive:orc&SERVER=hdfssrvcfg&VECTORIZE=true')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');
```

See PXF profiles for the full list of Hive profiles.

Filtering on partitions

For a Hive table partitioned on one or more columns, a `WHERE` clause on a partition column lets PXF skip non-matching partitions instead of reading and filtering every row in WarehousePG (WHPG). Which comparisons trigger this depends on the partition column's type:

- On a string-typed column, `=`, `<>`, `<`, `<=`, `>`, and `>=` all work.
- On an integral-typed column (such as `int`), `=` and `<>` work when you enable `hive.metastore.integral.jdo.pushdown` in `hive-site.xml`.

You must define partition columns at the end of your `CREATE EXTERNAL TABLE` column list, matching Hive's names and order.

Hive excludes a row from partition queries if its value doesn't match the partition column's type, commonly a `NULL`. PXF instead surfaces that row under an actual `NULL`, so `IS NULL` queries can return different row counts between Hive and PXF.

8.1.4 Authenticating with Kerberos

Authenticate PXF to a Kerberized Hadoop cluster using a Kerberos principal and keytab, instead of connecting as an unauthenticated user. This authentication method applies to any Hadoop connector, the Hadoop Distributed File System (HDFS), Hive, or HBase.

Enabling Kerberos authentication

Add these settings to the same server directory you already configured for HDFS, Hive, or HBase.

> **Note**
>
> This procedure requires a Kerberos principal and keytab for PXF, generated with `kadmin` (MIT Kerberos) or `ktpass` (Active Directory) on your Kerberos Key Distribution Center (KDC), or provided by your Kerberos administrator.

1. Copy the `pxf-site.xml` template into your server directory:

   ```
   cp $PXF_HOME/templates/pxf-site.xml $PXF_BASE/servers/<server_name>
   ```

2. Edit `pxf.service.kerberos.principal` and `pxf.service.kerberos.keytab` with your principal and keytab path:

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

   PXF replaces `_HOST` in the principal with each host's fully qualified domain name. See Configuration templates for the full list of `pxf-site.xml` properties.

3. Copy the keytab file to `$PXF_BASE/keytabs` on every segment host, at the same path. `pxf cluster sync` doesn't distribute this directory, so you need to place it there yourself.

4. Sync the server configuration to every segment host, then restart PXF to apply it:

   ```
   pxf cluster sync
   pxf cluster restart
   ```

To connect to a Kerberized Hive metastore over JDBC instead, set `hadoop.security.authentication` to `kerberos` in that server's `jdbc-site.xml`. See Connecting to SQL databases over JDBC.

Tuning Kerberos parameters

Edit optional properties the same way as Enabling Kerberos authentication above, in that same server's `pxf-site.xml`, then sync and restart PXF to apply the change:

- `pxf.service.user.impersonation`: By default, PXF impersonates the WarehousePG (WHPG) user running the query when it connects to the external system, so the Hadoop cluster sees requests as coming from that user rather than from PXF's own principal. Set this to `false` to disable impersonation and connect as the PXF principal for every query instead.
- `pxf.service.user.name`: Impersonates a specific proxy user rather than the connecting WHPG user. Use this on unsecured clusters where a proxy user is already configured, or to make every WHPG user appear as this one user on the Hadoop side.
- `pxf.service.kerberos.constrained-delegation`: Set to `true` to impersonate using Kerberos constrained delegation (S4U2Self/Proxy) instead of requiring PXF's principal to be a Hadoop proxy user, for clusters that support it. This delegation mode needs additional configuration on your Active Directory or IPA server.
- `pxf.service.kerberos.ticket-renew-window`: The fraction of a Kerberos ticket's lifespan PXF waits through before refreshing it, `0.8` by default. Set it to `0` to request a new ticket on every request.

8.2 Connecting to a SQL database over JDBC

Configure a server for your SQL database, then read and write its data through external tables over JDBC.

Configuring the server

Create a server that connects to an external SQL database over JDBC.

1. Create the server directory and copy the `jdbc-site.xml` template into it:

```
mkdir -p $PXF_BASE/servers/jdbcsrvcfg
cp $PXF_HOME/templates/jdbc-site.xml $PXF_BASE/servers/jdbcsrvcfg
```

2. Edit `jdbc-site.xml` with your connection details. For example, to connect to a WHPG database named `gpadmin` using the bundled PostgreSQL JDBC driver:

```
<?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://<host>:
<port>/gpadmin</value>
    </property>

<property>

<name>jdbc.user</name>
        <value><username>
</value>
    </property>

<property>

<name>jdbc.password</name>
        <value><password>
</value>
    </property>
</configuration>
```

PXF bundles the PostgreSQL and Hive JDBC drivers. For any other database, copy the vendor's JDBC driver JAR to `$PXF_BASE/lib` before applying the change.

`jdbc-site.xml` also has commented-out properties for connection pooling, batch and fetch sizes, session and connection-level settings, and user impersonation. Uncomment and set only the ones you need. See Configuration templates for the full list.

3. Sync the change to every segment host, then restart PXF to apply it:

```
pxf cluster sync
pxf cluster restart
```

Reading data

Read data from the external database by creating a readable external table with the `jdbc` profile. For example, to read the `test` table from the `gpadmin` database configured above:

```
CREATE EXTERNAL TABLE jdbc_read_example (id
int)
    LOCATION ('pxf://public.test?
PROFILE=jdbc&SERVER=jdbcsrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');

SELECT * FROM jdbc_read_example;
```

The path in `LOCATION` is `<schema>.<table>`. This same pattern works against any JDBC-compliant database, not only WHPG, once you point `jdbc.url` and `jdbc.driver` at that database and provide its driver JAR.

The data still lives in the source database. You can load it into a local table with `CREATE TABLE AS`:

```
CREATE TABLE test_local AS SELECT * FROM jdbc_read_example;
```

Writing data

Create a writable external table with the same `jdbc` profile to insert data into the external database:

```
CREATE WRITABLE EXTERNAL TABLE jdbc_write_example (id
int)
    LOCATION ('pxf://public.test?
PROFILE=jdbc&SERVER=jdbcsrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_export');

INSERT INTO jdbc_write_example SELECT * FROM some_local_table;
```

To query the data, create a separate readable external table at the same location:

```
CREATE EXTERNAL TABLE pxf_read_back (id
int)
    LOCATION ('pxf://public.test?
PROFILE=jdbc&SERVER=jdbcsrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');

SELECT * FROM
pxf_read_back;
```

8.3 Connecting to a network file system

Configure a server for your network file system, then read and write its data through external tables.

Configuring the server

Create a server that points PXF at a directory mounted at the same path on every WarehousePG (WHPG) host.

1. Create a server directory under `$PXF_BASE/servers`, and copy the `pxf-site.xml` template from `$PXF_HOME/templates` into it. For example, to configure a server named `nfssrvcfg`:

   ```
   mkdir -p $PXF_BASE/servers/nfssrvcfg
   cp $PXF_HOME/templates/pxf-site.xml $PXF_BASE/servers/nfssrvcfg
   ```

2. Uncomment `pxf.fs.basePath` and set it to the mount point, and set `pxf.service.user.impersonation` to `false`, since this connector always accesses files as the OS user running PXF rather than the connecting WHPG user:

   ```
   <property>

   <name>pxf.fs.basePath</name>

   <value>/mnt/extdata/pxffs</value>
   </property>
   <property>

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

   <value>false</value>
   </property>
   ```

   The path in `LOCATION` is relative to `pxf.fs.basePath`. See Configuration templates for the full list of `pxf-site.xml` properties.

3. Sync the change to every segment host, then restart PXF to apply it:

   ```
   pxf cluster sync
   pxf cluster restart
   ```

Reading data

Read a file from the mounted directory by creating a readable external table with the profile for its format and the server you configured. For example, to read a CSV file using the `nfssrvcfg` server:

```
CREATE EXTERNAL TABLE pxf_read_example (id int, name text, age
int)
    LOCATION ('pxf://data.csv?
PROFILE=file:text&SERVER=nfssrvcfg')
    FORMAT 'CSV'
(delimiter=',');

SELECT * FROM pxf_read_example;
```

PXF also supports structured formats like Parquet, through the same profile-based syntax:

```
CREATE EXTERNAL TABLE pxf_parquet_example
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://parquet_data/?
PROFILE=file:parquet&SERVER=nfssrvcfg')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';
```

The path in `LOCATION` can't be relative, and can't include the `$` character. See PXF profiles for the full list of supported formats, including worked examples of Avro, JSON, and multi-byte delimiters.

Writing data

Create a writable external table with the `pxfwritable_export` formatter to write WHPG data out to the mounted directory:

```
CREATE WRITABLE EXTERNAL TABLE pxf_write_example
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://parquet_data/?
PROFILE=file:parquet&SERVER=nfssrvcfg')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_export')
    ENCODING
'UTF8';

INSERT INTO pxf_write_example SELECT id, created, status FROM
some_local_table;
```

To query the data, create a separate readable external table at the same location:

```
CREATE EXTERNAL TABLE pxf_read_back
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://parquet_data/?
PROFILE=file:parquet&SERVER=nfssrvcfg')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';

SELECT * FROM
pxf_read_back;
```

8.4 Connecting to an object store

Configure a server for your object store, then read and write its data through external tables.

- S3-compatible object stores: Amazon S3 and MinIO.
- Azure: Azure Blob Storage and Azure Data Lake Storage.
- Google Cloud Storage.

8.4.1 Connecting to Azure

Configure a server for Azure Blob Storage or Azure Data Lake Storage, then read and write its data through external tables.

Configuring the server

Create a server directory under `$PXF_BASE/servers`, copy the matching XML template from `$PXF_HOME/templates` into it, and edit it with your connection details.

Blob Storage

```
mkdir -p $PXF_BASE/servers/wasbssrvcfg
cp $PXF_HOME/templates/wasbs-site.xml $PXF_BASE/servers/wasbssrvcfg
```

Edit `wasbs-site.xml` with your storage account name and key:

```
<?xml version="1.0" encoding="UTF-8"?
>
<configuration>

<property>
        <name>fs.azure.account.key.
<account_name>.blob.core.windows.net</name>
        <value><account_key>
</value>
    </property>
</configuration>
```

See Configuration templates for the full list of `wasbs-site.xml` properties.

Data Lake Storage Gen2

```
mkdir -p $PXF_BASE/servers/abfsssrvcfg
cp $PXF_HOME/templates/abfss-site.xml $PXF_BASE/servers/abfsssrvcfg
```

Edit `abfss-site.xml` with the client ID, secret, and token endpoint from an Azure AD app registration:

```
<?xml version="1.0" encoding="UTF-8"?
>
<configuration>

<property>

<name>fs.azure.account.auth.type</name>

<value>OAuth</value>
    </property>

<property>

<name>fs.azure.account.oauth.provider.type</name>

<value>org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider</value>
    </property>

<property>

<name>fs.azure.account.oauth2.client.endpoint</name>
        <value><client_endpoint>
</value>
    </property>

<property>

<name>fs.azure.account.oauth2.client.id</name>
        <value><client_id>
</value>
    </property>

<property>

<name>fs.azure.account.oauth2.client.secret</name>
        <value><client_secret>
</value>
    </property>
</configuration>
```

See Configuration templates for the full list of `abfss-site.xml` properties.

Sync the change to every segment host, then restart PXF to apply it:

```
pxf cluster sync
pxf cluster restart
```

Reading data

Read data from the object store by creating a readable external table with the profile for its format and the server you configured.

Blob Storage

```
CREATE EXTERNAL TABLE pxf_read_example (id int, name text, age
int)
    LOCATION ('pxf://<container>@<account_name>.blob.core.windows.net/<path>/data.csv?
PROFILE=wasbs:text&SERVER=wasbssrvcfg')
```

```
    FORMAT 'CSV'
(delimiter=',');

SELECT * FROM pxf_read_example;
```

Data Lake Storage Gen2

```
CREATE EXTERNAL TABLE pxf_read_example_adls (id int, name text, age
int)
    LOCATION ('pxf://<account_name>.dfs.core.windows.net/<path>/data.csv?
PROFILE=abfss:text&SERVER=abfsssrvcfg')
    FORMAT 'CSV'
(delimiter=',');

SELECT * FROM pxf_read_example_adls;
```

PXF also supports structured formats like Parquet, through the same profile-based syntax:

```
CREATE EXTERNAL TABLE pxf_parquet_example
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<container>@<account_name>.blob.core.windows.net/<path>/?
PROFILE=wasbs:parquet&SERVER=wasbssrvcfg')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';
```

See PXF profiles for the full list of supported formats, including worked examples of Avro, JSON, and multi-byte delimiters.

Writing data

Create a writable external table with the `pxfwritable_export` formatter to write WHPG data out to Azure, then a separate readable external table at the same location to query it back.

Blob Storage

```
CREATE WRITABLE EXTERNAL TABLE pxf_write_example
(
    id bigint,
    created timestamp without time zone,
    status
integer
```

```
)
    LOCATION ('pxf://<container>@<account_name>.blob.core.windows.net/<path>/?
PROFILE=wasbs:parquet&SERVER=wasbssrvcfg&COMPRESSION_CODEC=snappy')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_export')
    ENCODING
'UTF8';

INSERT INTO pxf_write_example SELECT id, created, status FROM
some_local_table;
```

```
CREATE EXTERNAL TABLE pxf_read_back
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<container>@<account_name>.blob.core.windows.net/<path>/?
PROFILE=wasbs:parquet&SERVER=wasbssrvcfg')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';

SELECT * FROM
pxf_read_back;
```

Data Lake Storage Gen2

```
CREATE WRITABLE EXTERNAL TABLE pxf_write_example_adls
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<account_name>.dfs.core.windows.net/<path>/?
PROFILE=abfss:parquet&SERVER=abfsssrvcfg&COMPRESSION_CODEC=snappy')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_export')
    ENCODING
'UTF8';

INSERT INTO pxf_write_example_adls SELECT id, created, status FROM
some_local_table;
```

```
CREATE EXTERNAL TABLE pxf_read_back_adls
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<account_name>.dfs.core.windows.net/<path>/?
PROFILE=abfss:parquet&SERVER=abfsssrvcfg')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';

SELECT * FROM
pxf_read_back_adls;
```

8.4.2 Connecting to Google Cloud Storage

Configure a server for Google Cloud Storage, then read and write its data through external tables.

Configuring the server

Create a server that connects to Google Cloud Storage (GCS).

1. Create a server directory under `$PXF_BASE/servers`, and copy the `gs-site.xml` template from `$PXF_HOME/templates` into it. For example, to configure a server named `gcssrvcfg`:

   ```
   mkdir -p $PXF_BASE/servers/gcssrvcfg
   cp $PXF_HOME/templates/gs-site.xml $PXF_BASE/servers/gcssrvcfg
   ```

2. Edit `gs-site.xml` with the path to a Google Cloud service account JSON key file, readable by the PXF service on every host:

   ```
   <?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><path_to_keyfile>
   </value>
       </property>
   </configuration>
   ```

   See Configuration templates for the full list of `gs-site.xml` properties.

3. Sync the change to every segment host, then restart PXF to apply it:

   ```
   pxf cluster sync
   pxf cluster restart
   ```

Reading data

Read data from the object store by creating a readable external table with the profile for its format and the server you configured. For example, to read a CSV file using the `gcssrvcfg` server:

```
CREATE EXTERNAL TABLE pxf_read_example (id int, name text, age
int)
    LOCATION ('pxf://<bucket>/<path>/data.csv?
PROFILE=gs:text&SERVER=gcssrvcfg')
    FORMAT 'CSV'
(delimiter=',');

SELECT * FROM pxf_read_example;
```

PXF also supports structured formats like Parquet, through the same profile-based syntax:

```
CREATE EXTERNAL TABLE pxf_parquet_example
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<bucket>/<path>/?
PROFILE=gs:parquet&SERVER=gcssrvcfg&COMPRESSION_CODEC=snappy')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';
```

See PXF profiles for the full list of supported formats, including worked examples of Avro, JSON, and multi-byte delimiters.

> **Note**
>
> `gs:json` supports reading only, unlike the equivalent JSON profiles for other object stores.

Writing data

Create a writable external table with the `pxfwritable_export` formatter to write WHPG data out to GCS:

```
CREATE WRITABLE EXTERNAL TABLE pxf_write_example
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<bucket>/<path>/?
PROFILE=gs:parquet&SERVER=gcssrvcfg&COMPRESSION_CODEC=snappy')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_export')
    ENCODING
'UTF8';

INSERT INTO pxf_write_example SELECT id, created, status FROM
some_local_table;
```

To query the data, create a separate readable external table at the same location:

```
CREATE EXTERNAL TABLE pxf_read_back
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<bucket>/<path>/?
PROFILE=gs:parquet&SERVER=gcssrvcfg')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';

SELECT * FROM
pxf_read_back;
```

8.4.3 Connecting to an S3-compatible object store

Configure a server for an S3-compatible object store, then read and write its data through external tables.

Configuring the server

Create a server that connects to an S3-compatible object store, such as Amazon S3 or MinIO.

1. Create a server directory under `$PXF_BASE/servers`, and copy an XML template from `$PXF_HOME/templates` into it. For example, to configure a server named `s3srvcfg` for Amazon S3:

```
mkdir -p $PXF_BASE/servers/s3srvcfg
cp $PXF_HOME/templates/s3-site.xml $PXF_BASE/servers/s3srvcfg
```

2. Edit the site XML file to authenticate to S3, using whichever of the following methods fits your environment.

Static access keys

Set your AWS access key and secret key directly:

```
<?xml version="1.0" encoding="UTF-8"?
>
<configuration>

<property>

<name>fs.s3a.access.key</name>
        <value><access_key>
</value>
    </property>

<property>

<name>fs.s3a.secret.key</name>
        <value><secret_key>
</value>
    </property>

<property>

<name>fs.s3a.fast.upload</name>

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

Temporary credentials, through AWS Assume Role

Obtain temporary credentials yourself, for example through `aws sts assume-role`, then set them along with the session token, and switch the credential provider to one that accepts a session token:

```
<property>

<name>fs.s3a.access.key</name>
    <value><temporary_access_key>
</value>
</property>
<property>

<name>fs.s3a.secret.key</name>
    <value><temporary_secret_key>
</value>
</property>
<property>

<name>fs.s3a.session.token</name>
    <value><session_token>
</value>
</property>
<property>

<name>fs.s3a.aws.credentials.provider</name>

<value>org.apache.hadoop.fs.s3a.TemporaryAWSCredentialsProvider</value>
</property>
```

Named AWS profile

Point PXF at a profile in `~/.aws/credentials` instead of storing keys in `s3-site.xml`:

```
<property>

<name>fs.s3a.aws.credentials.provider</name>

<value>com.amazonaws.auth.profile.ProfileCredentialsProvider</value>
</property>
```

Add your keys to `~/.aws/credentials` on every host, under the `default` profile or a named profile:

```
[test]
aws_access_key_id = <access_key>
aws_secret_access_key = <secret_key>
```

If you used a named profile instead of `default`, select it by setting `AWS_PROFILE` in `$PXF_BASE/conf/pxf-env.sh`:

```
export AWS_PROFILE=test
```

Credential process

Use the same `ProfileCredentialsProvider` as the named profile method, but point the profile at a script instead of static keys. The AWS SDK runs the script and reads its stdout as JSON, containing the access key, secret key, session token, and expiration, so this method works with credentials that rotate:

```
[test]
credential_process = /path/to/credential-script.sh
```

IAM role, through an EC2 instance profile

On EC2, attach an IAM role to the instances running PXF, and omit credentials from `s3-site.xml` entirely. PXF then pulls temporary credentials from the instance metadata service automatically. Set the credential provider explicitly only if you need to rule out another provider taking precedence:

```
<property>

<name>fs.s3a.aws.credentials.provider</name>

<value>com.amazonaws.auth.InstanceProfileCredentialsProvider</value>
</property>
```

Additional tuning properties

`s3-site.xml` accepts any other Hadoop S3A property, such as `fs.s3a.buffer.dir` for local buffering during upload, `fs.s3a.multipart.size` for the multipart upload threshold, or `fs.s3a.connection.maximum` for connection pooling. See the S3A section of the Hadoop-AWS module documentation for the full list of available properties.

MinIO and other S3-compatible stores also need an endpoint and path-style access, in addition to whichever authentication method you chose above:

```
<?xml version="1.0" encoding="UTF-8"?
>
<configuration>

<property>

<name>fs.s3a.endpoint</name>
        <value>http://<host>:<port>
</value>
    </property>

<property>

<name>fs.s3a.access.key</name>
        <value><access_key>
</value>
    </property>

<property>

<name>fs.s3a.secret.key</name>
        <value><secret_key>
</value>
    </property>

<property>

<name>fs.s3a.fast.upload</name>

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

<property>

<name>fs.s3a.path.style.access</name>

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

> **Note**
>
> Always include the URI scheme (`http://` or `https://`) in `fs.s3a.endpoint` . Omitting it causes connection errors such as `Unsupported or unrecognized SSL message` when PXF tries to reach the endpoint.

See Configuration templates for the full list of `s3-site.xml` properties, or minio-site.xml for MinIO and other S3-compatible stores.

3. Sync the change to every segment host, then restart PXF to apply it:

```
pxf cluster sync
pxf cluster restart
```

Reading data

Read data from the object store by creating a readable external table with the profile for its format and the server you configured. For example, to read a CSV file using the `s3srvcfg` server:

```
CREATE EXTERNAL TABLE pxf_read_example (id int, name text, age
int)
    LOCATION ('pxf://<bucket>/<path>/data.csv?
PROFILE=s3:text&SERVER=s3srvcfg')
    FORMAT 'CSV'
(delimiter=',');

SELECT * FROM pxf_read_example;
```

PXF also supports structured formats like Parquet, through the same profile-based syntax:

```
CREATE EXTERNAL TABLE pxf_parquet_example
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<bucket>/<path>/?
PROFILE=s3:parquet&SERVER=s3srvcfg&COMPRESSION_CODEC=snappy')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';
```

See PXF profiles for the full list of supported formats, including worked examples of Avro, JSON, and multi-byte delimiters.

Writing data

Create a writable external table with the `pxfwritable_export` formatter to write WHPG data out to the object store:

```
CREATE WRITABLE EXTERNAL TABLE pxf_write_example
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<bucket>/<path>/?
PROFILE=s3:parquet&SERVER=s3srvcfg&COMPRESSION_CODEC=snappy')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_export')
    ENCODING
'UTF8';

INSERT INTO pxf_write_example SELECT id, created, status FROM
some_local_table;
```

To query the data, create a separate readable external table at the same location:

```
CREATE EXTERNAL TABLE pxf_read_back
(
    id bigint,
    created timestamp without time zone,
    status
integer
)
    LOCATION ('pxf://<bucket>/<path>/?
PROFILE=s3:parquet&SERVER=s3srvcfg')
    FORMAT 'CUSTOM' (FORMATTER =
'pxfwritable_import')
    ENCODING
'UTF8';

SELECT * FROM
pxf_read_back;
```

Overriding credentials per query

Regardless of how the server authenticates, a user can override its access key and secret key for a single query, using custom options in the `LOCATION` clause:

```
CREATE EXTERNAL TABLE pxf_s3_override (id int, name
text)
    LOCATION ('pxf://<bucket>/<path>/data.csv?PROFILE=s3:text&SERVER=s3srvcfg&accesskey=
<access_key>&secretkey=<secret_key>')
    FORMAT 'CSV'
(delimiter=',');
```

> **Important**
>
> Credentials passed this way are visible as part of the external table definition. Avoid this method in a production environment.

This override works only for plain Amazon S3, not for Azure, Google Cloud Storage, or MinIO servers.

9 PXF reference

Reference material for administering PXF.

- Commands: The `pxf` and `pxf cluster` commands.
- Profiles: The built-in profiles PXF ships for each connector.
- Configuration files: The files under `$PXF_BASE/conf` and the properties you can set in each.
- Configuration templates: The site XML templates under `$PXF_HOME/templates` and the properties in each.

9.1 PXF commands

PXF provides two related commands. Run `pxf <command>` to act on the local host only, or `pxf cluster <command>` to run the same action across every host in the cluster over SSH, using the coordinator's `PXF_HOME`, `PXF_BASE`, and other environment variables. Use the `pxf cluster` form for routine cluster administration, covered in Managing the PXF cluster.

Cluster commands

Run these from the coordinator host.

pxf cluster prepare

Creates `$PXF_BASE` on every host and copies the default configuration templates from `$PXF_HOME/conf` into it. See Initializing and starting PXF for usage in context.

```
pxf cluster prepare
```

pxf cluster register

Installs the PXF extension's control, SQL, and shared library files under `$GPHOME` on every host. Needed because `make install` only places these files under `$GPHOME` on the coordinator. See Installing PXF for usage in context.

```
pxf cluster
register
```

pxf cluster start

Starts the PXF service on every host. See Starting, stopping, and restarting PXF for usage in context.

```
pxf cluster start
```

pxf cluster stop

Stops the PXF service on every host. See Starting, stopping, and restarting PXF for usage in context.

```
pxf cluster stop
```

pxf cluster restart

Stops, then starts, the PXF service on every host. See Starting, stopping, and restarting PXF for usage in context.

```
pxf cluster restart
```

pxf cluster status

Reports whether the PXF service is both running and responsive on every host. See Monitoring PXF for usage in context.

```
pxf cluster
status
```

pxf cluster sync

Copies `$PXF_BASE/{conf,lib,servers}` from the coordinator to every other host. See Applying a configuration change and Relocating $PXF_BASE for usage in context.

```
pxf cluster sync [options]
```

Option	Description
`--delete`	Also remove files on the remote hosts that no longer exist on the coordinator.

Example:

```
pxf cluster sync --delete
```

pxf cluster migrate

Migrates configuration from an older PXF installation into a new `$PXF_BASE`.

```
pxf cluster migrate
```

pxf cluster init

(Deprecated) Installs the PXF extension under `$GPHOME`. Use `pxf cluster register` instead.

```
pxf cluster init
```

pxf cluster reset

(Deprecated) No operation.

```
pxf cluster reset
```

Local commands

Run these on a single host to act on that host's PXF instance only.

pxf prepare

Creates `$PXF_BASE` on the local host and copies the default configuration templates into it. Equivalent to `pxf cluster prepare`, scoped to this host.

```
pxf prepare
```

pxf register

Installs the PXF extension under `$GPHOME` on the local host. Useful after a WHPG upgrade.

```
pxf
register
```

pxf start

Starts the local PXF service instance.

```
pxf start
```

pxf stop

Stops the local PXF service instance.

```
pxf stop
```

pxf restart

Restarts the local PXF service instance.

```
pxf restart
```

pxf status

Reports whether the local PXF service instance is running.

```
pxf
status
```

pxf sync

Copies `$PXF_BASE/{conf,lib,servers}` from the local host to another host. Unlike `pxf cluster sync`, you specify the target host explicitly.

```
pxf sync <hostname> [options]
```

Option	Description
`--delete`	Also remove files on `<hostname>` that no longer exist locally.

Example:

```
pxf sync sdw3 --delete
```

pxf migrate

Migrates configuration from an older PXF installation into a new `$PXF_BASE` on the local host.

```
pxf migrate
```

pxf version

Shows the PXF server version.

```
pxf version
```

pxf init

(Deprecated) Installs the PXF extension under `$GPHOME` on the local host. Use `pxf register` instead.

```
pxf init
```

pxf reset

(Deprecated) No operation.

```
pxf reset
```

9.2 PXF configuration files

`$PXF_BASE/conf` holds the files you use to configure the PXF service itself, separate from the server configurations you use to connect to external data sources. See Managing the PXF cluster for how to apply a change to any of these files.

pxf-application.properties

Configures the PXF Service application. Commonly changed properties:

Property	Description	Default
`server.address`	The address PXF listens on.	`localhost`
`pxf.max.threads`	The maximum number of PXF Tomcat threads.	200
`pxf.connection.timeout`	The connection timeout for read operations.	5m
`pxf.connection.upload-timeout`	The connection timeout for write operations.	5m
`pxf.log.level`	The log level for the PXF service.	info

pxf-env.sh

Configures the PXF Java virtual machine and runtime paths. Commonly changed properties:

Property	Description	Default
`JAVA_HOME`	The path to the Java JRE home directory.	`/usr/java/default`
`PXF_LOGDIR`	The PXF log directory.	`$PXF_BASE/logs`
`PXF_RUNDIR`	The PXF run directory.	`$PXF_BASE/run`
`PXF_JVM_OPTS`	The default options for the PXF JVM, including heap size.	`-Xmx2g -Xms1g`
`PXF_OOM_KILL`	Whether PXF terminates itself automatically on an out-of-memory error.	true
`PXF_OOM_DUMP_PATH`	The absolute path to the dump file PXF generates on an out-of-memory error.	No dump file (empty)
`PXF_LOADER_PATH`	Additional directories and JARs for PXF to class-load.	(empty)
`LD_LIBRARY_PATH`	Additional directories and native libraries for PXF to load.	(empty)

You already edited this file to set `JAVA_HOME` in Configuring and starting PXF.

`PXF_HOST` and `PXF_PORT` aren't part of `pxf-env.sh`. They're plain environment variables you set in `~/.bashrc` on a host, and they tell the `pxf` extension where to reach the PXF service, rather than configuring the service itself. Set them only if you're running PXF on a non-default port or on a host other than the one WHPG expects.

pxf-log4j2.xml

Configures PXF and third-party library logging. The root logger, which covers PXF's own code, logs at the level set by `pxf.log.level` in `pxf-application.properties`, `info` by default. Third-party libraries PXF depends on, such as `org.apache.hadoop` and `org.apache.parquet`, already have their own `<Logger>` entries tuned to a quieter level, so they don't flood the log at `info`.

Add a `<Logger>` element inside `<Loggers>` to set a level for a specific package, for example to debug PXF's Hive connector:

```
<Logger name="org.greenplum.pxf.plugins.hive"
level="debug"/>
```

pxf-profiles.xml

Defines custom PXF profiles, pairing a name with the Java classes that read or write data for it. PXF's built-in profiles are already defined for you, so you only need this file if you're adding a custom profile:

```
<profiles>

<profile>

<name>my_profile</name>
        <description>A custom
profile</description>

<plugins>

<fragmenter>com.example.MyFragmenter</fragmenter>

<accessor>com.example.MyAccessor</accessor>

<resolver>com.example.MyResolver</resolver>
        </plugins>
    </profile>
</profiles>
```

`fragmenter`, `accessor`, and `resolver` are fully qualified names of Java classes implementing PXF's plugin interfaces. Reference the profile from a `LOCATION` clause with `PROFILE=my_profile`.

9.3 PXF configuration templates

`$PXF_HOME/templates` holds a template site XML file for each connector. Copy the template for your connector into a server directory under `$PXF_BASE/servers/<server_name>`, then edit it as described on the relevant connector page, for example Object stores or Hadoop. Reference `<server_name>` as the `SERVER` parameter in an external table's `LOCATION` clause.

core-site.xml

Property	Default	Description
`fs.defaultFS`	`hdfs://0.0.0.0:8020`	The address of the HDFS NameNode.

hdfs-site.xml

Property	Default	Description
`dfs.permissions.enabled`	`true`	Enforces HDFS file permission checks.
`dfs.block.access.token.enable`	`true`	Requires a block access token for datanode communication.
`dfs.support.append`	`true`	Allows appending to existing files.
`hadoop.proxyuser.gpadmin.hosts`	`*`	The hosts `gpadmin` can proxy from when impersonation is enabled.
`hadoop.proxyuser.gpadmin.groups`	`*`	The groups `gpadmin` can impersonate when impersonation is enabled.
`hadoop.security.authorization`	`true`	Enables service-level authorization checks.

hive-site.xml

Property	Default	Description
`hive.metastore.uris`	`thrift://localhost:9083`	The URI PXF uses to reach the Hive metastore.
`hive.metastore.integral.jdo.pushdown`	`true`	Pushes down predicates on integral partition columns to the metastore, improving performance for tables with many partitions.

hbase-site.xml

Property	Default	Description
`hbase.rootdir`	`hdfs://0.0.0.0:8020/hbase`	The location on HDFS where HBase stores its data.
`hbase.zookeeper.quorum`	`127.0.0.1`	The ZooKeeper ensemble HBase uses for coordination.

mapred-site.xml

Property	Default	Description
`mapreduce.framework.name`	`local`	The MapReduce execution framework.
`mapreduce.input.fileinputformat.input.dir.recursive`	`true`	Reads files in subdirectories recursively.

yarn-site.xml

Property	Default	Description
`yarn.resourcemanager.address`	`0.0.0.0:8032`	The address of the YARN ResourceManager.

s3-site.xml

For Amazon S3 itself, without the endpoint or path-style properties MinIO and other S3-compatible stores need. See `minio-site.xml` for those.

Property	Default	Description
`fs.s3a.access.key`	Required, no default	Your AWS access key ID.
`fs.s3a.secret.key`	Required, no default	Your AWS secret access key.
`fs.s3a.fast.upload`	`true`	Uses fast upload buffering for writes.

minio-site.xml

For MinIO and other S3-compatible object stores.

Property	Default	Description
`fs.s3a.endpoint`	Required, no default	The object store's endpoint URL, including the URI scheme.
`fs.s3a.access.key`	Required, no default	The access key for the object store.
`fs.s3a.secret.key`	Required, no default	The secret key for the object store.
`fs.s3a.fast.upload`	`true`	Uses fast upload buffering for writes.
`fs.s3a.path.style.access`	`true`	Uses path-style bucket addressing, required by MinIO and most non-AWS S3-compatible stores.

Both `s3-site.xml` and `minio-site.xml` also accept any other Hadoop S3A property, such as `fs.s3a.buffer.dir` for local buffering during upload, `fs.s3a.multipart.size` for the multipart upload threshold, or `fs.s3a.connection.maximum` for connection pooling. See the S3A section of the Hadoop-AWS module documentation for the full list of available properties. See Connecting to an S3-compatible object store for the properties specific to each authentication method.

gs-site.xml

For Google Cloud Storage.

Property	Default	Description
`google.cloud.auth.service.account.enable`	`true`	Authenticates using a Google Cloud service account.
`google.cloud.auth.service.account.json.keyfile`	Required, no default	The path to your Google Cloud service account JSON key file.
`fs.AbstractFileSystem.gs.impl`	`com.google.cloud.hadoop.fs.gcs.GoogleHadoopFS`	The file system implementation class for Google Cloud Storage.

abfss-site.xml

For Azure Data Lake Storage Gen2.

Property	Default	Description
`fs.azure.account.auth.type`	`OAuth`	The authentication type.
`fs.azure.account.oauth.provider.type`	`org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider`	The OAuth token provider class.
`fs.azure.account.oauth2.client.endpoint`	Required, no default	Your Azure OAuth client endpoint.
`fs.azure.account.oauth2.client.id`	Required, no default	Your Azure OAuth client ID.
`fs.azure.account.oauth2.client.secret`	Required, no default	Your Azure OAuth client secret.

wasbs-site.xml

For Azure Blob Storage.

Property	Default	Description
`fs.adl.oauth2.access.token.provider.type`	`ClientCredential`	The OAuth token provider type.
`fs.azure.account.key.<storage_account>.blob.core.windows.net`	Required, no default	The access key for your Azure Blob Storage account. Replace `<storage_account>` in the property name with your account name.
`fs.AbstractFileSystem.wasbs.impl`	`org.apache.hadoop.fs.azure.Wasbs`	The file system implementation class for Azure Blob Storage.

jdbc-site.xml

Connection properties

Property	Default	Description
`jdbc.driver`	Required, no default	The JDBC driver class name, for example `org.postgresql.Driver`.
`jdbc.url`	Required, no default	The JDBC connection URL, for example `jdbc:postgresql://localhost/postgres`.
`jdbc.user`	Required, no default	The username for connecting to the external database.
`jdbc.password`	Required, no default	The password for connecting to the external database.

Optional properties

These properties are commented out in the template. Uncomment and set only the ones you need.

Property	Default	Description
`jdbc.pool.enabled`	`true`	Enables connection pooling.
`jdbc.pool.property.maximumPoolSize`	`15`	The maximum number of pooled connections to the external database.
`jdbc.pool.property.connectionTimeout`	`30000`	The maximum time, in milliseconds, to wait for a connection from the pool.
`jdbc.pool.property.idleTimeout`	`30000`	The maximum time, in milliseconds, a connection sits idle in the pool.
`jdbc.pool.property.minimumIdle`	`0`	The minimum number of idle connections to maintain in the pool.
`jdbc.pool.qualifier`	None	Creates a separate connection pool for each value of this property. Useful for per-user pools alongside `jdbc.session.property` settings that switch users after connecting.
`jdbc.statement.writeSize`	`100`	The number of rows batched together before writing.
`jdbc.statement.fetchSize`	`1000`	The number of rows fetched at a time when reading. The MySQL JDBC driver instead streams rows one at a time by default.
`jdbc.connection.transactionIsolation`	None	The transaction isolation level: `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, or `SERIALIZABLE`. Confirm your database supports the level you choose.
`jdbc.session.property.<name>`	None	Sets a session-level variable in the external database before running a query. Add one property per variable, replacing `<name>` in the property name.
`jdbc.connection.property.<name>`	None	Sets a JDBC connection property. Add one property per value, replacing `<name>` in the property name.
`pxf.service.user.impersonation`	`false`	Impersonates the connecting WHPG user when connecting to the external database.
`hadoop.security.authentication`	None	Set to `kerberos` when connecting to a Kerberized Hive metastore.
`jdbc.read.prepared-statement`	`false`	Uses a `PreparedStatement` instead of a `Statement` when reading. Some drivers, such as Teradata's FastExport, require this.

pxf-site.xml

For Kerberos authentication, user impersonation, predicate pushdown, and ORC and Parquet write behavior. This template applies to any server, not only Hadoop connectors. See Authenticating with Kerberos for the Kerberos and impersonation properties in context.

Property	Default	Description
`pxf.service.kerberos.principal`	`gpadmin/_HOST@EXAMPLE.COM`	The Kerberos principal the PXF service uses. PXF replaces `_HOST` with the host's fully qualified domain name.
`pxf.service.kerberos.keytab`	`${pxf.base}/keytabs/pxf.service.keytab`	The path to the Kerberos keytab file.
`pxf.service.user.impersonation`	`true`	Impersonates the connecting WHPG user when accessing the external system.
`pxf.service.kerberos.constrained-delegation`	`false`	Uses Kerberos constrained delegation (S4U2Self/Proxy) for impersonation instead of requiring PXF to be a Hadoop proxy user. Requires additional Active Directory or IPA server configuration.
`pxf.service.kerberos.ticket-renew-window`	`0.8`	The fraction of a Kerberos ticket's lifespan PXF waits through before refreshing it.
`pxf.service.user.name`	`${user.name}`	Overrides the proxy user PXF presents to the external system. Set this on unsecured clusters with a proxy user already configured, or to make every WHPG user appear as this one user.
`pxf.fs.basePath`	None	The base path PXF uses when building a file URI for a `file:*` profile. Required for any server that accesses data using a `file:*` profile.
`pxf.ppd.hive`	`true`	Enables predicate pushdown for Hive profiles.
`pxf.sasl.connection.retries`	`5`	The number of retries when a NameNode refuses a SASL connection with a `GSS initiate failed` error.
`pxf.orc.write.decimal.overflow`	`round`	The behavior when writing a bare `NUMERIC` column, mapped internally to `DECIMAL(38, 10)`, and a value's integer digit count exceeds what that precision and scale allow: `error` fails the write, `round` rounds the value to fit, or `ignore` writes `NULL`. Define the column as `NUMERIC(<precision>, <scale>)` instead of a bare `NUMERIC` for more control over when an overflow happens.
`pxf.orc.write.timezone.utc`	`true`	Writes ORC timestamp values using UTC instead of the PXF JVM's local timezone.
`pxf.parquet.write.decimal.overflow`	`round`	The behavior when writing a bare `NUMERIC` column, mapped internally to `DECIMAL(38, 18)`, and a value's integer digit count exceeds what that precision and scale allow: `error` fails the write, `round` rounds the value to fit, or `ignore` writes `NULL`. Define the column as `NUMERIC(<precision>, <scale>)` instead of a bare `NUMERIC` for more control over when an overflow happens.

9.4 PXF profiles

Specify a profile as the `PROFILE` parameter in an external table's `LOCATION` clause. See Understanding PXF profiles for what a profile is.

HDFS, object stores, and network file system

`hdfs`, `s3`, `abfss` (Azure Data Lake Storage), `wasbs` (Azure Blob Storage), `gs` (Google Cloud Storage), and `file` (network file system) all share the same format suffixes. Combine a connector prefix with a format suffix to form a profile name, for example `s3:parquet` or `file:csv`.

Format	Profile suffix	Available on	Notes
Delimited text	`:text`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`, `file`	Plain text.
CSV	`:csv`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`, `file`	
Multi-line text	`:text:multi`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`, `file`	Supports multi-line records.
Fixed-width text	`:fixedwidth`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`, `file`	Reads columns at fixed byte offsets.
Parquet	`:parquet`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`, `file`	Columnar, binary, with the schema embedded in the file.
ORC	`:orc`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`, `file`	Columnar, binary.
Avro	`:avro`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`, `file`	Row-based, binary, paired with a schema.
JSON	`:json`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`, `file`	
SequenceFile	`:SequenceFile`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`	Hadoop's binary key-value container format.
Avro in a SequenceFile	`:AvroSequenceFile`	`hdfs`, `s3`, `abfss`, `wasbs`, `gs`	Avro-encoded records stored inside a SequenceFile container.

`file:SequenceFile` and `file:AvroSequenceFile` aren't available, since SequenceFile is a Hadoop-specific container format. See Object stores, Hadoop, and Network file system for connecting to each of these sources.

Only `:text` and `:csv` use `FORMAT 'TEXT'` or `FORMAT 'CSV'`. Every other format needs `FORMAT 'CUSTOM'`, with `FORMATTER='pxfwritable_import'` on a readable external table or `FORMATTER='pxfwritable_export'` on a writable one.

Hive

Profile	Description
`hive`	Reads any Hive-supported storage format through Hive's SerDe layer.
`hive:text`	Reads a Hive table stored as text files directly, skipping the SerDe layer.
`hive:rc`	Reads a Hive table stored as RCFile directly, skipping the SerDe layer.

Profile	Description
`hive:orc`	Reads a Hive table stored as ORC directly, skipping the SerDe layer. Supports vectorized reads, see Hive for the `VECTORIZE` option.

The `hive` profile reads through Hive's SerDe (serializer/deserializer), the class Hive itself uses to parse the table's storage format, so it works with any format Hive supports. A `hive:<format>` profile instead skips the SerDe and reads the underlying files directly, which is faster but only available for the formats listed above. See Hive for connecting to a Hive metastore.

HBase

Profile	Description
`hbase`	Reads an HBase table. See HBase.

JDBC

Profile	Description
`jdbc`	Reads and writes data through a JDBC driver. See Connecting to SQL databases over JDBC.

Examples

The following examples show the options and syntax specific to each format, using a mix of connectors to illustrate that the same options apply regardless of which connector you pair the format with.

Multi-byte delimiters

The `:text` and `:csv` profiles only support a single-byte delimiter. To read data with a multi-byte or multi-character delimiter, such as a currency symbol, use a `:csv` profile with the `pxfdelimited_import` custom formatter instead:

```
CREATE EXTERNAL TABLE mbyte_delim (id int, city text, country
text)
    LOCATION ('pxf://data/multibyte_currency?
PROFILE=s3:csv&SERVER=s3srvcfg')
    FORMAT 'CUSTOM' (FORMATTER='pxfdelimited_import',
DELIMITER='¤');
```

`DELIMITER` is required and accepts a string up to 32 bytes. Add `QUOTE` and `ESCAPE` if your data quotes or escapes values:

```
CREATE EXTERNAL TABLE mbyte_delim_quoted (id int, city text, country
text)
    LOCATION ('pxf://data/multibyte_quoted?
PROFILE=s3:csv&SERVER=s3srvcfg')
    FORMAT 'CUSTOM' (FORMATTER='pxfdelimited_import', DELIMITER='¤', QUOTE '"', ESCAPE
'\');
```

If your data's line ending isn't a line feed, add `NEWLINE` (`CR` or `CRLF`) to both the formatter options and the `LOCATION` clause. This formatter only supports reading data, not writing it.

Avro

Avro is a compact binary format that pairs each record with a schema describing its fields. A schema looks like this:

```
{
  "type": "record",
  "name": "User",
  "fields": [
    { "name": "id", "type": "int" },
    { "name": "name", "type": "string" }
  ]
}
```

Create a readable external table with an `:avro` profile, pointing at an existing Avro file:

```
CREATE EXTERNAL TABLE pxf_avro_read (id int, name text, details
text[])
    LOCATION ('pxf://data/emp_data.avro?
PROFILE=gs:avro&SERVER=gcssrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');

SELECT * FROM
pxf_avro_read;
```

If you don't provide a `SCHEMA` option, PXF generates the Avro schema from the external table's column definitions, using each column name as the Avro field name:

```
CREATE WRITABLE EXTERNAL TABLE pxf_avro_write (id int, username text, followers
text[])
    LOCATION ('pxf://data/pxf_write.avro?
PROFILE=gs:avro&SERVER=gcssrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_export');

INSERT INTO pxf_avro_write VALUES (1, 'Mickey', ARRAY['Pluto', 'Donald',
'Mini']);
```

Create a separate readable external table at the same location to query the data back, since you can't query a writable external table directly. To use your own schema file instead of letting PXF generate one, add the `SCHEMA` option to the `LOCATION` clause:

```
CREATE WRITABLE EXTERNAL TABLE pxf_avro_write_with_schema (id int, username text, followers
text[])
    LOCATION ('pxf://data/pxf_write.avro?
PROFILE=gs:avro&SERVER=gcssrvcfg&SCHEMA=/path/to/avro_schema.avsc')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_export');
```

JSON

PXF reads and writes JSON in two layouts. The default expects one JSON object per line, known as JSON Lines. An alternate mode expects a single root object whose value is an array of records, useful when each file already holds a complete JSON document rather than one record per line.

Use dot notation to project nested fields into columns:

```
CREATE EXTERNAL TABLE pxf_json_read (created_at text, "user.id" int, "user.location"
text)
    LOCATION ('pxf://data/pxf_examples/events.jsonl?
PROFILE=file:json&SERVER=filesrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');

SELECT * FROM
pxf_json_read;
```

For a file that holds a single root object with an array of records instead of one object per line, add `IDENTIFIER`, naming the field that marks the start of each record:

```
CREATE EXTERNAL TABLE pxf_json_read_single (created_at text, "user.id"
int)
    LOCATION ('pxf://data/pxf_examples/events.json?
PROFILE=file:json&SERVER=filesrvcfg&IDENTIFIER=created_at')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_import');
```

PXF writes each row as its own JSON Lines record by default:

```
CREATE WRITABLE EXTERNAL TABLE pxf_json_write (created_at text, id int, location
text)
    LOCATION ('pxf://data/pxf_examples/events_out?
PROFILE=file:json&SERVER=filesrvcfg')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_export');

INSERT INTO pxf_json_write VALUES ('2026-07-21', 1,
'Austin');
```

Add `ROOT` to write a single root object holding an array of records instead, naming the root-level attribute:

```
CREATE WRITABLE EXTERNAL TABLE pxf_json_write_single (created_at text, id int, location
text)
    LOCATION ('pxf://data/pxf_examples/events_root?
PROFILE=file:json&SERVER=filesrvcfg&ROOT=records')
    FORMAT 'CUSTOM'
(FORMATTER='pxfwritable_export');
```

Create a separate readable external table at the same location, with a matching `IDENTIFIER`, to query the data back, since you can't query a writable external table directly. PXF only writes scalar columns or one-dimensional arrays, so a writable external table can't include nested or column-projected fields.