Platform Extension Framework (PXF): Enabling Parallel Query Processing Over Heterogeneous Data Sources In Greenplum

Venkatesh Raghavan, Alexander Denissov, Francisco Guerrero, Oliver Albertini, Divya Bhargov, Lisa Owen, Shivram Mani, Lav Jain
Pivotal Inc.

ABSTRACT

With the explosion of data stores and cloud services, data now resides across many disparate systems and in a variety of formats. When multiple data sets exist in external systems, it is often necessary to perform a lengthy ETL (extract, transform, load) operation to get data into the database. But what if we only needed a small subset of the data? What if we only want to query the data to answer a specific question or to create a specific visualization? In this case, it's 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.

In this paper, we propose Greenplum Database Platform Extension Framework (PXF) for accomplishing this task. PXF is an open source project that provides parallel, high throughput data access and federated query processing across heterogeneous data sources via built-in connectors that map a Greenplum external table definition to an external data source. PXF's architecture enables users to efficiently query large datasets from multiple external sources, without requiring those datasets be loaded into Greenplum.

KEYWORDS

Federated Query Processing, Massively Parallel Databases, Hadoop, Hive, SQL on Hadoop, Transactional Databases, JDBC, External Tables, HDFS, Cloud Object Storage

1 INTRODUCTION

Organizational business units are driven by a clear objective – building tools to make data-driven decisions. Enterprises as well as the research community have moved away from the traditional approach of storing all the data in a centralized data warehouse[19]. The decision on how and where to store the data is driven by four main factors, namely: (1) total cost of ownership of the solution, (2) format in which the data is available, (3) service level requirements of the application, and lastly (4) the human element – the platform and tools with which the data analysts / engineers are familiar. However, as an organization grows, what started as a small operation easily balloons into a massive data ocean composed of multiple data lakes, data streams, and one-off databases. It becomes cost prohibitive to re-architect and migrate the different data solutions, and then extract value from these disparate sources. At this stage, there arises a need to have a unified means to query these data stores in a scalable manner.

To motivate, consider a data source tracking monthly sales across many years as shown in Figure 1. Transactional data represents real-time data, also known as 'hot data', and is frequently accessed by the application. This operational data is stored in MySQL. Data subject to more analytic and fewer operational operations, also known as 'warm data', is stored in Greenplum. Older or 'cold data' is accessed rarely and is archived in AWS S3.

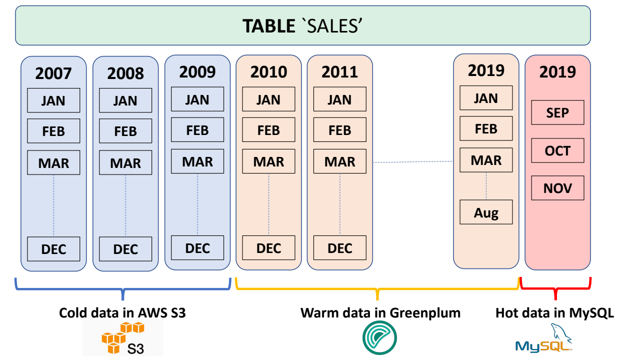

Figure 1: Federated Data Storage Layout.

```
EXPLAIN SELECT sum(amt)  FROM sales WHERE date >= '2019-01-01';

                                   QUERY PLAN
--------------------------------------------------------------------------------------------
 Aggregate  (cost=21970.73..21970.74 rows=1 width=32)
   -> Gather Motion 3:1  (slice1; segments: 3)  (cost=21970.67..21970.72 rows=1 width=32)
        -> Aggregate  (cost=21970.67..21970.68 rows=1 width=32)
             -> Result  (cost=0.00..20710.00 rows=168089 width=13)
                  -> Append  (cost=0.00..20710.00 rows=168089 width=13)
                       -> Seq Scan on sales_1_prt_monthly_109 sales  (cost=0.00..901.25 rows=7123 width=13)
                            Filter: date >= '2019-01-01'::date
                       -> Seq Scan on sales_1_prt_monthly_110 sales  (cost=0.00..901.25 rows=7123 width=13)
                            Filter: date >= '2019-01-01'::date
                       -> Seq Scan on sales_1_prt_monthly_111 sales  (cost=0.00..901.25 rows=7123 width=13)
                            Filter: date >= '2019-01-01'::date
                       -> Seq Scan on sales_1_prt_monthly_112 sales  (cost=0.00..901.25 rows=7123 width=13)
                            Filter: date >= '2019-01-01'::date
                       -> Seq Scan on sales_1_prt_monthly_113 sales  (cost=0.00..901.25 rows=7123 width=13)
                            Filter: date >= '2019-01-01'::date
                       -> Seq Scan on sales_1_prt_monthly_114 sales  (cost=0.00..901.25 rows=7123 width=13)
                            Filter: date >= '2019-01-01'::date
                       -> Seq Scan on sales_1_prt_monthly_115 sales  (cost=0.00..901.25 rows=7123 width=13)
                            Filter: date >= '2019-01-01'::date
                       -> Seq Scan on sales_1_prt_monthly_116 sales  (cost=0.00..901.25 rows=7123 width=13)
                            Filter: date >= '2019-01-01'::date
                       -> External Scan on sales_1_prt_recent sales  (cost=0.00..13500.00 rows=111112 width=13)
                            Filter: date >= '2019-01-01'::date
 Optimizer status: Postgres query optimizer
(24 rows)
```

Figure 2: Federated Query Plan.

In this example, as the end of the year approaches, a user may want to determine the total sales for the current year. To obtain this the user needs to access both the hot data stored in MySQL as well as warm data for the year 2019 (stored in Greenplum). Greenplum's polymorphic storage capability coupled with the *Platform Extension Framework* (PXF) makes multi-temperature data queries simple and seamless. Figure 2 shows the federated query plan for such a query that operates on local as well as external data.

In this paper, we propose Greenplum's *Platform Extension Framework* (PXF), an open source project[1], that enables users to query heterogeneous data sources via pre-built connectors installed with Greenplum. PXF speeds up retrieval of external data with highly parallel, high-throughput access. PXF also provides the ability to execute portions of a query statement on other data sources. These connectors map Greenplum external table definitions to external data sources – data objects in the cloud, data in Hadoop, or data in other relational databases – in different formats and structures. PXF is built on the following principles:

SQL Support. The framework is built to provide a SQL-based interface for querying different data sources just like any managed table inside the Greenplum database.

Simple. The API used to register the data source, data format, user impersonation information, and formatting options (such as delimiters) is *easy to use* and *standardized*. Section 3.1 discusses our proposed PXF external table API.

Support Different Sources. The framework is able to extract data from different transactional or analytical databases such as Postgres, MySQL, Oracle, Hive, and Greenplum.

Support Different Data Formats. PXF handles a wide variety of data formats ranging from unstructured data like CSV and text; row-oriented formats like JSON and Avro; as well as column-oriented storage formats like ORC and Parquet.

Cloud Independent. The framework is able to query data stored on various cloud storage solutions.

Extensible. The framework allows custom connectors to other data sources and formats. The optimization in query processing is independent of the data sources and formats.

Scalable Multi-Core Ready. PXF is multi-core aware, and is able to partition and then distribute the work of fetching data from the different external sources.

Upgradable. As more data sources, formats, and optimizations are developed, the framework facilitates easy upgrade independent of the host database system.

The paper is organized as follows. We introduce the Greenplum database architecture in Section 2. In Section 3, we present the PXF external table API, architecture and describe how users can set up the connections to different data servers. The underlying technical concepts needed to build our framework are elaborated in Section 4. Section 5 highlights the differentiating features of PXF. Section 6 discusses how PXF works on Kubernetes. Section 7 presents our experimental study followed by a discussion on the related work in Section 8. We summarize this paper with final remarks in Section 9.

2 PRELIMINARIES

In this section, we describe the high level architecture of Greenplum Database and a very brief introduction to cloud object storage.

2.1 Greenplum Data Warehouse

Greenplum Database [10] is an MPP analytics database that adopts a shared-nothing architecture with multiple cooperating processors (typical deployments include tens or hundreds of segment hosts). Figure 3 shows the high level architecture of Greenplum. Greenplum handles storage and processing of large amounts of data by distributing the load across several servers to create an array of individual databases, all working together to present a single database image. The *master* is the entry point to Greenplum; it is the entity to which clients connect and submit SQL statements. The master coordinates work with other database instances, called *segments*, to handle data processing and storage. A query submitted to the Greenplum master is optimized and broken into smaller components which are then dispatched to the segments which work together to deliver the final results. The interconnect, a standard Gigabit Ethernet switching fabric, is the networking layer responsible for inter-process communication between the segments.

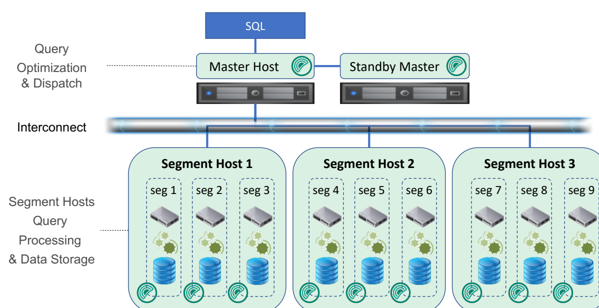

Figure 3: Sample Greenplum Database Architecture.

The Greenplum optimizer[18] generates a plan that includes explicit data movement directives; the cost of moving data is taken into account during optimization. Greenplum executor includes several pipelined execution stages, with explicit communication between segments at each stage. For instance, a multi-stage aggregation plan is used to compute local aggregation at each segment in parallel, then Greenplum redistributes the intermediate aggregation result based on the grouping columns to compute the final result.

2.2 Cloud Object Storage

Cloud object stores are increasingly becoming a viable storage option especially for big data analytics platforms. This is mainly due to the scalability and the high availability guarantees provided by the cloud providers. Amazon Simple Storage Service (AWS S3) is an object storage service that provides a web services interface to customers for storing and retrieving data. S3 Select is an S3 feature designed to increase query performance by up to 400% and reduce querying costs by as much as 80%[12]. The feature uses simple SQL expressions to retrieve a subset of an object's data rather than the entire object, which can be up to 5 TB in size. Google Cloud Platform infrastructure provides Google Cloud Storage [8], a RESTful online file storage web service to store and access data. Azure Blob storage provides users a scalable cloud solution to store unstructured data such as text, log files, and binary data on Microsoft Azure - a cloud computing service [5].

3 PLATFORM EXTENSION FRAMEWORK

The Platform Extension Framework (PXF) provides parallel, high throughput data access and federated queries across heterogeneous data sources. PXF provides the ability to execute portions of the

[1] https://github.com/greenplum-db/pxf

query statement on other data sources. In this section, we (1) introduce the PXF external table API by which users can specify data residing in external sources, (2) describe a high level architectural view of our proposed framework and (3) describe the means to configure the endpoints of the external data servers.

3.1 PXF External Tables API

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. After configuring PXF and assigning privileges, you can use the **CREATE EXTERNAL TABLE** command to create an external table using the *pxf* protocol. The syntax for a creating an external table is as 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 specifies the *pxf* protocol as a URI that identifies the path to the location of the external data. For example, if the external data store is HDFS, this identifies the absolute path to a specific HDFS file. If external data store is Hive, this identifies a schema-qualified Hive table name.
- The *PROFILE* clause specifies the profile used 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.
- The *SERVER* provides the named server configuration PXF uses to access the data. A server definition provides the information required for PXF to access an external data source, and includes information such as server location, access credentials, and other relevant properties.

To illustrate the use of the external table API, consider the use case presented in Section 1. An external table to access the "hot" transactional data stored in a MySQL database is defined as follows.

Example 1 (MySQL External Table Storing Hot Data).

```
CREATE READABLE EXTERNAL TABLE  recent_sales
(id int, sdate date, amt decimal(10,2))
LOCATION ('pxf://company.salesdata?
PROFILE=Jdbc&SERVER=mysql-db')
FORMAT 'CUSTOM' (FORMATTER='pxfwritable_import');
```

The connection parameters, such as the remote database URL, username, and password, and which JDBC driver to use, is defined in a 'jdbc-site.xml' file located in a PXF configuration directory named 'mysql-db'. See Figure 5 in Section 3.3 for more details.

Similarly we can create an external table for the archival data stored in S3 as defined below. Figure 6 in Section 3.3 has the configuration details needed to access the S3 server.

Example 2 (External Table in AWS S3 Storing Cold Data).

```
CREATE READABLE EXTERNAL TABLE  sales_archive
(id int, sdate date, amt decimal(10,2))
LOCATION ('pxf://bucket/sales_data/before_2010_sales.csv?
PROFILE=s3:csv&SERVER=s3') FORMAT 'CSV';
```

3.2 PXF Architecture

The PXF framework is composed of three main components.

- **PXF Extension** is a C client library that implements the PostgreSQL extension. This extension runs on each database segment and communicates with the PXF Server using REST API calls.
- **PXF Server** is a Java web application deployed in Apache Tomcat[1] running on each Greenplum segment host in the cluster. The PXF Server receives requests from the PXF extension and translates those requests to the external systems.
- **PXF Command Line Interface** is a Go application that enables users to manage the PXF Servers in their Greenplum installation.

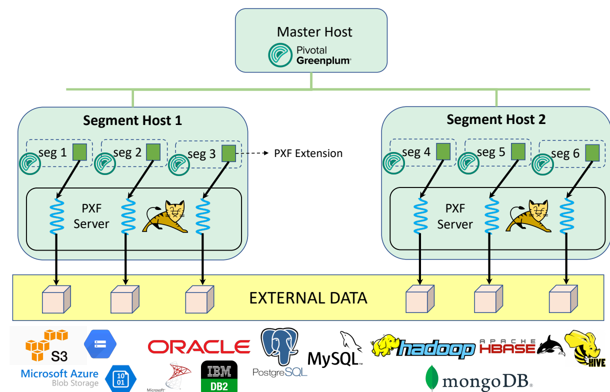

Figure 4: PXF Extension Framework Architecture.

When a Greenplum Database user runs a query against a PXF external table, the query plan is generated and then dispatched from the Greenplum master to the Greenplum segments. The *PXF Extension* running on each Greenplum segment process, in turn, forwards the request to the PXF Server running on the same host. A typical PXF deployment topology, as shown in Figure 4, places a *PXF Server* on each host that runs one or more Greenplum segment processes. A PXF Server receives at least as many requests as the number of segment processes running on the host, and potentially more if the query is complex and contains multiple processing slices. Each such request gets a PXF Server thread assigned to it.

PXF CLI is a utility that is implemented in Go which provides a convenient way to administer PXF servers running on Greenplum. With the PXF CLI, you can run commands on a specific PXF server (resident on an individual Greenplum segment host) to perform actions like starting and stopping the server, checking the server's status, and other administrative tasks. To make PXF administration scalable for large installations, we also provide the ability to run this command across the entire installation.

3.3 Configuration of External Data Servers

PXF supports concurrent access to data residing on different servers. To achieve this, PXF needs information about each server it accesses, such as server location, credentials, access keys, or other information depending on the external system. For instance, Figure 5 denotes the MySQL parameters needed for the external table

recent_sales introduced in Example 1, and Figure 6 denotes the S3 parameters needed to access the external data *sales_archive* described in Example 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property>
        <name>jdbc.driver</name>
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property>
        <name>jdbc.url</name>
        <value>
           jdbc:mysql-db://company.example.com/salesdata
        </value>
    </property>
    <property>
        <name>jdbc.user</name>
        <value>YOUR_DATABASE_JDBC_USER</value>
    </property>
    <property>
        <name>jdbc.password</name>
        <value>YOUR_DATABASE_JDBC_PASSWORD</value>
    </property>
</configuration>
```

Figure 5: MySQL parameters defined in jdbc-site.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property>
        <name>fs.s3a.access.key</name>
        <value>YOUR_S3_ACCESS_KEY</value>
    </property>
    <property>
        <name>fs.s3a.secret.key</name>
        <value>YOUR_S3_SECRET_KEY</value>
    </property>
</configuration>
```

Figure 6: AWS S3 parameters defined in s3-site.xml.

The configuration files for each server are stored in the local filesystem where PXF is installed. A typical PXF installation places binaries in the `$PXF_HOME` directory and configuration files in the `$PXF_CONF` directory. Configuring a new server is as easy as creating a new directory inside `$PXF_CONF/servers` and filling in a template configuration file. For some external data sources, additional dependencies are required. To access Oracle, the Oracle JDBC driver needs to be placed in `$PXF_CONF/lib`; PXF will classload files placed in the `$PXF_CONF/lib` directory. Figure 7 illustrates a PXF installation with multiple server configurations.

A server configuration uses a specific PXF connector to access the external data source. For example, the PXF JDBC connector is used to access tables on an Oracle server, whereas the PXF Hadoop connector is used to access files residing on a Hadoop cluster.

The server configuration is specified in the `SERVER` parameter of the `LOCATION URI` as described in Subsection 3.1. PXF uses the name of the `SERVER` to determine the location of the configuration files on the local filesystem. For example, when `SERVER`

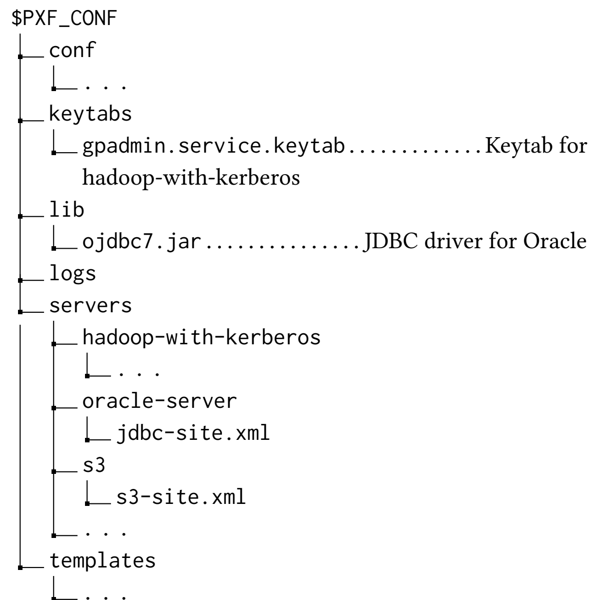

Figure 7: Layout of the $PXF_CONF directory to support multiple server configurations.

`= hadoop-with-kerberos` is specified in the `LOCATION URI`, PXF reads `*-site.xml` configuration files from the `$PXF_CONF/servers/hadoop-with-kerberos/` directory. In some cases, additional files are required to access a server; for example, to access a Kerberized Hadoop cluster, a keytab file has to be provided to login to the external system.

4 PXF CONCEPTS

Internally, PXF defines three interfaces to read and write data from external data sources: the **Fragmenter**, the **Accessor**, and the **Resolver**. PXF supplies multiple implementations for these interfaces that utilize different communication protocols and support multiple formats of data. To simplify usability, PXF provides a concept of **Profile**, a simple name that encapsulates a combination of specific implementations of a fragmenter, accessor and resolver.

4.1 Fragmenter

A PXF **Fragmenter** is a functional interface that splits the overall dataset from an external data source into a list of independent **fragments** that can be read in parallel. The fragmenter does not retrieve the actual data, it works only with metadata. The fragment metadata describes the location of a given data fragment and (optionally) the offset of the data within the overall dataset.

Processing of a single query is distributed across all Greenplum segments and multiple PXF Server threads, which run independent of each other. They know only about the overall count of Greenplum segments and the identity of the process that sent a query request. Each Greenplum segment performs two types of REST calls to the PXF Server during the query against a PXF external table. A single **fragmentation call** followed by zero or more **bridge calls**.

During the **fragmentation call**, the fragmenter divides up the overall dataset on the external system into fragments or chunks

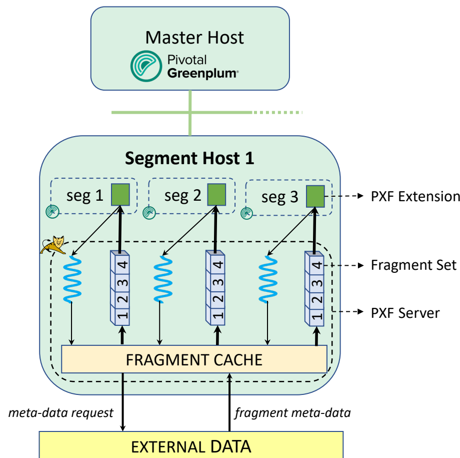

Figure 8: PXF Fragmentation.

of data, along with metadata about each fragment. The process of retrieving the set of fragment metadata is called **fragmentation** and is depicted in Figure 8. While a fragment is the smallest indivisible chunk of data that is processed by a single PXF Server thread, the nature of fragments is different for various types of external systems. For example, a fragment can be an HDFS split for a file stored in Hadoop HDFS, or a subset of records from a JDBC table. The process of obtaining metadata about fragments also differs depending on the external system - for example, it can be a call to Hadoop HDFS NameNode for a file stored in Hadoop HDFS or a calculation of interval buckets for a partitioned JDBC table. Refer to Section 5.6 for details on how PXF caches metadata during the fragment call. The partitioning options supported by PXF for external JDBC tables are discussed in Section 5.4.

The resulting fragment set needs to be divided evenly and fairly among the Greenplum servers. This is achieved by having all Greenplum segments determine the overall workload via the **fragmentation call** to, and then applying a modulo-based hashing function to the set of fragment metadata to determine which subset of fragments the given process has to work on. Each Greenplum segment then iterates over the assigned subset of fragments, and performs a **bridge call** for each such fragment to read the actual data. A different PXF Server thread processes each bridge call for a single fragment of data. During the **bridge call**, the PXF Server thread receives fragment metadata from the Greenplum segment, and is responsible for fetching data from the external system for the given fragment using a specific **Accessor**.

4.2 Accessor

A PXF **Accessor** is a functional interface that is responsible for reading or writing data from/to external data sources, as well as converting data into individual records.

Because different types of external data sources require different protocols and authentication methods, there is a need to build a separate accessor implementation for each type of external server. Furthermore, these data sources can support a variety of data formats. Thus, PXF provides different accessors to connect to remote databases using JDBC, cloud object storage vendors, and remote Hadoop clusters using HDFS Client libraries. These PXF accessors are able to read text, Parquet, Avro, and CSV data, among others.

When an accessor processes a request to retrieve data for a specific fragment, it first fetches the configuration of the external data source. Next, it analyses the metadata for the requested fragment, and uses the client API libraries for its corresponding protocol to request the required data from the external data source. The data it reads is broken it into records, such as lines of a text file, rows in a JDBC result set, or Parquet records, to name a few. Each such record, while still being in a proprietary format, is then wrapped into a common data structure that is used across PXF to exchange data.

When supported by the external data source, as in the case of JDBC, Hive, S3 Select, among others, the accessor is also responsible for pushing down predicates and projecting columns from the incoming query. We elaborate in more detail on predicate pushdown and column projection in Section 5.

4.3 Resolver

A PXF **Resolver** is a functional interface that decodes (when reading) or encodes (when writing) the record read from/to the external system by an accessor into individual fields. It then maps field data types and values into a format that Greenplum or the external system understands. Once a record is processed by the resolver, it can be converted into an on-the-wire representation (such as CSV or binary) and shipped to Greenplum to be re-created as a tuple, or into a format like Parquet or Avro, and sent to the external system.

Resolver implementations are usually specific to the data formats that they process. For instance, PXF provides resolvers to convert CSV lines, JDBC ResultSet columns, Parquet records, Avro records, and JSON documents. When reading text files, it may be useful not to change the data and let Greenplum do the post-processing; PXF provides built-in resolvers that facilitate this.

4.4 Profile

To simplify usability, PXF provides a concept of **Profile** that is a simple named mapping that represents both the protocol for connecting to an external data source and the format of the data that needs to be processed. For example, to read text file from S3 cloud storage, a user specifies the profile *s3:text*, while reading a parquet file from HDFS requires the *hdfs:parquet* profile.

Users specify the profile parameter value when they define Greenplum external table. At runtime, PXF receives the value of the profile parameter with the request for data from Greenplum and refers to the PXF system configuration to select the appropriate Fragmenter, Accessor, and Resolver implementations for data retrieval and processing.

4.5 Connector

A **Connector** is a logical grouping of multiple components (fragmenters, accessors, resolvers, and profiles) that enables users to process data from a given type of external data source. While it does not have a specific technical representation, connector is a useful concept to describe overall capabilities of PXF when working with external data. PXF provides the following native connectors to external systems:

- **JDBC connector.** Allows PXF to access data from external databases such as Oracle, MySQL, Postgres, Hive, and, in general, any database using the appropriate JDBC driver.
- **Cloud connector.** Enables access to data residing on Amazon S3, Google Cloud Storage, Azure Data Lake, Microsoft's Azure Blob Storage, among others.
- **Hadoop connector.** Facilitates retrieval of data stored in files on Hadoop.
- **Hive connector.** Reads data from Hive tables by accessing metadata from Hive MetaStore and then accessing the underlying files.
- **HBase connector.** Ingests data stored in HBase tables.

		Read	Write
File Storage	Hadoop 3.x	✓	✓
	Hadoop 2.x	✓	✓
	Simple Storage Service (S3)	✓	✓
	Google Cloud Storage	✓	✓
	Microsoft Azure Data Lake	✓	✓
	Azure Blob Storage	✓	✓

Table 1: Supported Hadoop Compatible File Systems.

		Read	Write
File Formats	Delimited Single Line Text (TEXT)	✓	✓
	Comma Separated Values (CSV)	✓	✓
	Delimited Text With Quoted Linefeeds	✓	✗
	Avro	✓	✓
	JSON	✓	✗
	Parquet	✓	✓
	AvroSequenceFile	✓	✓
	SequenceFile	✓	✓

Table 2: Supported File Formats.

For each specific external data source, PXF natively supports different file formats. For instance, for PXF connectors to Hadoop Compatible File Systems[17] such as HDFS, S3, Google Cloud Storage, Microsoft Azure Data Lake, and Azure Blob Storage, the supported file format support are shown in Table 1 and Table 2 respectively. When using the Hive connector to access Hive systems, PXF file format support is detailed in Table 3. When accessing Hive using the JDBC connector, PXF supports all file formats.

5 FEATURE HIGHLIGHTS

5.1 Predicate Pushdown

PXF supports filter pushdown when fetching external data. Constraints from the `WHERE` clause of a `SELECT` query are extracted and passed to the external data source for filtering. This optimization significantly reduces the amount of data that needs to be fetched from the remote system. For a given user query, Greenplum master generates a query plan tree which is then dispatched to the segments. The external scan operator in the query plan has scalar expressions that denotes the predicate on which the input tuples

		Read	Write
Hive File Formats	Text File	✓	✓
	SequenceFile	✓	✓
	Avro SequenceFile	✓	✓
	RCFile	✓	✓
	ORC	✓	✓
	Parquet	✓	✓
Hive Version	Hive 1.x	✓	✓
	Hive 2.x	✓	✓
	Hive 3.x	✓	✓

Table 3: File Format Supported for Hive via Hive Connector.

are pruned. Different data source systems support varying scalar comparison operators as well as data types. For instance, (1) Hive supports only text and integral data types for partition filtering, and (2) HBase does not support OR predicates. To ensure correctness, PXF connectors (1) convert a Greenplum predicate into a corresponding filter clause supported by the external system, and (2) ignore pushing predicates that are incompatible with the external system; deferring the processing of any additional filtering to Greenplum. Consider the external MySQL table shown in Example 1, where we fetch sales data recorded in a specific date range: `SELECT * FROM recent_sales WHERE DATE BETWEEN '2019-09-01' and '2019-10-15'`. The naive approach would be for PXF to fetch the external sales data in its entirety and apply the predicates within Greenplum. With predicate pushdown enabled, PXF augments the predicate to the external query, and the MySQL query translates to `SELECT * FROM company.salesdata WHERE DATE BETWEEN '2019-09-01' and '2019-10-15'`.

5.2 Column Projection

PXF improves performance by fetching data only for required columns from the external table. This not only reduces the amount of data that is fetched and transferred to Greenplum, but it can also significantly reduce disk I/O on the external system. For example, if the user is interested only in customer data for the query above, PXF will issue the following query to MySQL: `SELECT id, data FROM company.salesdata WHERE DATE BETWEEN '2019-09-01' and '2019-10-15'`. Note here that we drop the `'amt'` column from the projection list. Data formats such as ORC and Parquet are columnar and more suited for analytical workloads which require projection and grouping and support higher levels of columnar compression. PXF, when accessing such data, fetches only the projected columns which significantly improves latency by eliminating the cost of unnecessary external disk I/O operations.

5.3 Named Queries

PXF allows users to define a custom query on the data residing in a remote database. PXF JDBC connector executes this query when it retrieves data for the corresponding Greenplum external table.

5.3.1 Motivation. Consider *customer* and *order* tables stored in an external MySQL database that one wants to query from Greenplum. To accomplish this, the Greenplum user defines an external table

for each of these two tables. The MySQL connection parameters are defined in 'jdbc-site.xml' (see Figure 5 in Section 3.3):

Example 3 (Customer and Orders MySQL External Tables).

```
CREATE READABLE EXTERNAL TABLE customers
(id int, name text, city text, state text)
LOCATION  ('pxf://customers?PROFILE=Jdbc&SERVER=mysql-db')
FORMAT 'CUSTOM' (formatter='pxfwritable_import');

CREATE READABLE EXTERNAL TABLE orders
(amount int, month int, customer_id int)
LOCATION ('pxf://orders?PROFILE=Jdbc&SERVER=mysql-db')
FORMAT 'CUSTOM' (formatter='pxfwritable_import');
```

To create a report for the monthly sales totals for their customers, a Greenplum user will join the customer information with the sales orders, creating the following SQL query:

Example 4 (Query: Monthly Customer Sales).

```
SELECT c.name, c.city, o.month, sum(o.amount)
FROM   customers c JOIN orders o ON c.id = o.customer_id
GROUP BY c.name, c.city, o.month
```

When Greenplum executes this query, it performs two external scans; one for all customer information and the other for all orders. The actual work joining the tables and calculating aggregates happens inside Greenplum. It can be argued, that in this example it would have been more beneficial to run joins and aggregations in the remote database itself and then just return the results to Greenplum for presentation to users. This would avoid the overhead of sending the whole data set from the remote database across the network to PXF and then on to Greenplum. Another benefit in this scenario is that PXF would not need to process every record of both tables, thus preserving computational resources for other tasks.

Performing custom queries, joins, and aggregation on data wholly residing in a remote system in that system itself is the main purpose for the Named Query feature in PXF. This feature also becomes extremely beneficial if the data to be accessed is managed by Apache Hive in a Hadoop cluster. When the datasets are huge, as usually is the case when Apache Hadoop is involved, a Hadoop cluster often has more available computational resources than Greenplum to perform complex data processing involving joins and aggregations. Utilizing the power and resources of the Hadoop cluster to process data and retrieve the results from Apache Hive Server2 via the JDBC connector is a primary use case where PXF Named Query provides substantial benefits.

5.3.2 Implementation. Allowing the Greenplum user to specify an SQL query in the LOCATION field when defining a Greenplum external table would have created security concerns, especially for SQL injection attacks, and would require PXF to analyze submitted SQL for such vulnerabilities. Instead, we allow users to pre-define their queries and store them in a text file in the PXF server configuration directory for the remote database. This leaves verification of the SQL query correctness and security to a Greenplum administrator who is authorized to manage PXF configuration files.

For the example above, a user would define the following query:

Example 5 (Named Query: Monthly Customer Sales).

```
SELECT c.name, c.city, o.month, sum(o.amount) AS total
FROM   customers c JOIN orders o ON c.id = o.customer_id
GROUP BY c.name, c.city, o.month
```

This query text will then be placed in a file named with a *.sql* extension, for example *sales.sql*, and then added to the PXF configuration directory for the *mysql-db* server. A Greenplum user can refer to this query by file name when defining an external table:

Example 6 (Named Query External Table).

```
CREATE READABLE EXTERNAL TABLE sales_summary
(name text, city text, month int, total int)
LOCATION  ('pxf://query:sales?PROFILE=Jdbc&SERVER=mysql-db')
FORMAT 'CUSTOM' (formatter='pxfwritable_import');
```

Greenplum users can now view the query results as an external table in any SELECT command.

SELECT * FROM sales_summary;

Notice that the query name provided in the *LOCATION* URI is prefixed by **query:** string. This is an indication to PXF that when it processes a query for this external table, it should read the text of a query in a file named *sales.sql* in the server configuration directory and dispatch it to the remote server. Once the remote server executes the pre-defined query and responds with data, PXF forwards it to Greenplum to represent the resulting dataset.

PXF actually does a little more than just dispatch the pre-defined query to the remote server – PXF also wraps the query in an external statement so that it can apply optimizations such as column projection, predicate pushdown, and partition filtering.

The example above does not use any optimizations, so the actual query dispatched by PXF to the remote database will be:

Example 7 (Query Dispatched to MySQL).

```
SELECT name, city, month, total FROM (
   SELECT c.name, c.city, o.month, sum(o.amount) AS total
   FROM   customers c JOIN orders o ON c.id = o.customer_id
   GROUP BY c.name, c.city, o.month) pxfsubquery
```

5.3.3 Predicate Pushdown in Named Query. This predicate pushdown is also supported for the *named queries*. In Example 6 above, if the user is only interested in sales from customers in San Francisco, they will specify a filter as follows:

```
SELECT * FROM sales_summary WHERE city = 'San Francisco';
```

In this case, PXF will add the filter to the wrapping SQL clause and issue the following query to the remote database:

```
SELECT name, city, month, total FROM (
   SELECT c.name, c.city, o.month, sum(o.amount) AS total
   FROM   customers c JOIN orders o ON c.id = o.customer_id
   GROUP BY c.name, c.city, o.month) pxfsubquery
WHERE city = 'San Francisco'
```

Note that the original query stored in the *sales.sql* file has not changed, and that PXF applies the filter on the wrapping SQL statement only. PXF relies on the remote database to further process the resulting SQL statement and optimize it such that this filter is pushed further down into the aggregate query.

5.4 PXF JDBC Read Partitioning

PXF supports partitioning during JDBC query processing when the Greenplum user specifies the `PARTITION_BY` option in the external

table definition. It is important to note that partitioning does not refer to whether the data is actually stored in partitioned tables in the remote database. Rather, it provides a hint to PXF that the dataset in the remote table can be processed in parallel, with each partition being retrieved by a separate PXF thread. When the data is also partitioned in the same manner on the storage side in the remote database, the retrieval by the remote database will also be very efficient, as it needs to only scan a given partition and not the whole table to satisfy a query for a partition from PXF.

When the `PARTITION_BY` clause is provided in the external table definition, PXF server instances divide the work of retrieving individual partitions among themselves using a round-robin, modulo-based distribution. Each individual instance appends filters to the `WHERE` clause of the `SELECT` statement to restrict the returned dataset to only its assigned partition. This optimization allows multiple PXF server instances to fetch the remote dataset concurrently, which, when properly tuned, dramatically increases system throughput.

To obtain optimal performance, care should be taken when identifying the partition column. One should choose a partition column that can take advantage of concurrent reads. For example, when using PXF to read from an Oracle database, use an Oracle partitioning key column. When querying Greenplum, we can specify the special column named gp_segment_id to take advantage of parallel reads, where each segment scans only its own data and then hands off that data to the master.

PXF JDBC Read Partitioning can significantly speed up access to external database tables, speeding up data querying or data loading.

Partitioning with Named Query. Partition filtering is also supported for Named Queries. We extend Example 6 by partitioning the year's quarters as follows:

Example 8 (Named Query with Partitioning).

```
CREATE READABLE EXTERNAL TABLE sales_summary
(name text, city text, month int, total int)
LOCATION  ('pxf://query:sales?PROFILE=Jdbc&SERVER=mysql-db&
          PARTITION_BY=month:int&RANGE=1:12&INTERVAL=3')
FORMAT 'CUSTOM' (formatter='pxfwritable_import');
```

The query `SELECT * FROM sales_summary` triggers four PXF queries run by different PXF threads. PXF adds a partition filter to the wrapping SQL clause of each query, but parameters of the filter will differ for each PXF thread. For example, PXF generates this first query to the remote database:

```
SELECT name, city, month, total FROM (
   SELECT c.name, c.city, o.month, sum(o.amount) AS total
   FROM   customers c JOIN orders o ON c.id = o.customer_id
   GROUP BY c.name, c.city, o.month) pxfsubquery
WHERE month >= 1 AND month < 4
```

Accordingly, the second query advances the filter on the month field by an interval of 3:

```
SELECT name, city, month, total FROM (
   SELECT c.name, c.city, o.month, sum(o.amount) AS total
   FROM   customers c JOIN orders o ON c.id = o.customer_id
   GROUP BY c.name, c.city, o.month) pxfsubquery
WHERE month >= 4 AND month < 7
```

Just as with non-partitioned named queries (example above), the original query stored in the *sales.sql* file is unchanged.

5.5 Putting it all together

All of the features illustrated above work well together to allow users to specify named queries with additional aggregations applied to the results returned by PXF. If we want the pre-defined query to return results only for sales in California, we (1) create a new named query file *california_sales.sql*, (2) write a SQL similar to *sales.sql* with an additional filter to only keep California sales, and (3) use an aggregate function to collect the total number of sales.

Example 9 (California Sales Named Query).

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

The *state* column only exists in the remote table; it is not defined in the Greenplum external table. We can however use it in the *named query* since the query will be dispatched and processed by the remote database.

Next, we define an external table on California sales, and specify a partitioning schema on month with an interval of 3 months:

Example 10 (California Sales External Table).

```
CREATE READABLE EXTERNAL TABLE california_sales
(name text, city text, month int, total int)
LOCATION  ('pxf://query:california_sales?PROFILE=Jdbc&
SERVER=mysql-db&PARTITION_BY=month:int&RANGE=1:12&INTERVAL=3')
FORMAT 'CUSTOM' (formatter='pxfwritable_import')
```

Finally, consider the requirements of a query to return total sales amounts per month exceeding $100 in the city of San Francisco. The user will submit the following query to Greenplum:

Example 11 (Query: San Francisco Sales Exceeding $100).

```
SELECT name, month, total FROM california_sales
WHERE city = `San Francisco`
GROUP BY name, month, total
HAVING total > 100;
```

As illustrated previously, PXF will submit 4 queries to the remote database, one query each per calendar quarter. PXF constructs the first query as follows:

Example 12 (Query dispatched to MySQL by PXF).

```
SELECT name, city, month, total FROM (
   SELECT   c.name, c.city, o.month,
            sum(o.amount) AS total, count(o.id) AS count
   FROM     customers c JOIN orders o ON c.id = o.customer_id
   WHERE    c.state = 'CALIFORNIA'
   GROUP BY c.name, c.city, o.month
) pxfsubquery
WHERE city = 'San Francisco' AND month >= 1 AND month < 4
```

This query that PXF creates and dispatches to the remote database illustrates all of the concepts that we have introduced before:

- Internal subquery defined in `california_sales.sql` is left intact.

- Column projection adds only `name, city, month, total` to the list of columns to retrieve from remote database. The column count is not referenced by the Greenplum query and hence is not requested from the external database.
- Predicate pushdown optimization introduces the predicate `city='San Francisco'` that the user specified in the WHERE clause of the Greenplum query.
- Partition filtering optimization introduces the predicate `month >= 1 AND month < 4` that was derived from the options specified in the Greenplum external table definition LOCATION clause.

It is worth mentioning that in this example, the aggregate functions defined in the Greenplum query:

```
GROUP BY name, month, total
HAVING total > 100
```

are executed in Greenplum after PXF returns the data. Because Greenplum does not currently support aggregate pushdown optimizations, PXF is not made aware that the query it runs is part of a user query that contains aggregate functions.

5.6 Caching Metadata Information

In the initial implementation, every PXF Server thread performed the fragmentation process at the same time before applying the modulo function. For a workload that included tens of thousands of Hadoop HDFS files, we noticed both spiking of PXF Server memory usage and increased workload on a Hadoop Namenode process, which increased risks of system instability and was detrimental for overall performance. To mitigate this, we introduced a cache for fragments metadata inside each PXF Server.

When requests for fragment metadata arrive to PXF Server for a given query, they first look up the fragments list in the cache. The key to a cache entry consists of the PXF configuration server name, Greenplum transaction ID, path to the external resource, and any SQL statement predicates that a user may have specified for the query. If the fragment information has already been retrieved by another PXF Server thread processing the same query from a different Greenplum segment, then that fragment set is used and returned, without making additional calls to the external system. If the information is not yet available, the given thread must be the first one to request it. In this case, the cache locks the entry and proceeds with the fragmentation process by issuing a callback. Once the fragment metadata is retrieved from the external system, it is placed in the cache entry and the cache unlocks it, making it available for future requests. The processing thread then returns the information to the caller.

Since all the threads start processing requests from the same query at about the same time, we configured a ten second cache entry expiration time. This value gives sufficient time for a thread that is "a bit late" to still get the information, while limiting the amount of time potentially large metadata sits in the JVM memory. In the worst case, when a thread requests the information after the cache entry has already expired, it will obtain it once again and then the fragment information will expire again 10 seconds thereafter.

5.7 Security

PXF supports access to services that are secured with the Kerberos authentication protocol. PXF also supports configuration of, and concurrent access to, multiple kerberized and non-kerberized services. You can access Kerberos-authenticated Hadoop, Hive, and HBase services with PXF.

To access a kerberized service, PXF requires the configuration files for the service, as well as a principal and keytab to authenticate against Kerberos. Each server configuration is independent and requires its own set of configuration files and keytabs. The kerberized service can be either configured with a headless keytab or with a service keytab. When using a service keytab, you generate a unique keytab for the PXF server instance running on each host. PXF logs in to a Kerberos Realm using the principal and keytab configured for the given server. PXF initiates the login process the first time it attempts to access the kerberized service (i.e. the first time a user or application issues a query against the kerberized service). PXF then caches the token and reuses it for subsequent queries to the system. PXF maintains a list of tokens for the different kerberized services it has accessed. When a token expires, PXF automatically performs a re-login and caches the new token that it obtains.

5.8 PXF and Cloud Object Store

PXF currently supports querying data stored in AWS S3, Google Cloud Storage, and Azure Blob store. To motivate our discussion, consider an example user query on customer data stored on S3 and filtered by age. Without the support of S3 Select, PXF must fetch all of the data from S3 and return it to Greenplum to then be processed. This naive approach is resource-intensive; it requires significantly more CPU time for Greenplum to filter unwanted results and greater network bandwidth to fetch the entire data set (only to be later discarded by the filter). PXF supports AWS S3 Select optimization that allows data filtering on AWS S3's side which vastly reduces the amount of data that must be transmitted over the network. In addition, this approach helps save compute resources on the Greenplum cluster. It is important to note that the resource utilization savings is directly proportional to the size of the stored data, as well as the selectivity of the filtering dimensions. It is a common use case for queries on cloud database management systems (like data mining, transactional, analytical and ad-hoc queries) to operate on only a small portion of the vast amounts of data stored. This makes PXF with S3 Select crucial for building a scalable solution for querying data lakes.

6 DISCUSSION: PXF + KUBERNETES

Kubernetes (K8s)[4] is fast becoming the ubiquitous way to deploy containerized applications in a multi-cloud setup. Greenplum for Kubernetes (GP4K) [14] provides a simplified way to deploy, scale, heal, and even decommission Greenplum instances in the cloud. PXF is available on Kubernetes as part of the Greenplum Workbench[14]. This environment simplifies the deployment of PXF and automates configuration, HA, patching, upgrades, and other day1 and day2 operations. For example, the user can store `PXF_CONF` contents in an S3 object store, which PXF can later use to automatically initialize `PXF_CONF` directory wherever PXF is deployed. We considered the two PXF+GP4K deployment topologies described below.

Colocated PXF Service. In this topology, the PXF Server is placed in a container alongside every Greenplum segment. This would maximize colocation and reduce network hops to zero, but would result in many more PXF servers than necessary, including on mirrors. Resources reserved for PXF servers take away from the pool of resources that could otherwise be used for Greenplum (or other applications in a multi-tenant K8s cluster). K8s schedules all containers in a Pod as a unit. Therefore if there is any out-of-control PXF server that is utilizing CPU or memory beyond the prescribed limits, then the whole pod including the segment container are subjected to OOM termination.

PXF Microservice. This cloud native deployment model of PXF instantiates the PXF Server within containers inside Kubernetes pods independently of Greenplum segments. This means that Greenplum would have to go through the cluster network to reach PXF.

Our experiments revealed that the additional network hop results in a maximum 10% performance penalty. This drawback however is out-weighed by the ease of setup and use. For instance, `PXF_HOST` will be the same across all pods. Moreover, being an independent application could allow PXF to be used by other Greenplum clusters, or even non-Greenplum use cases in the future.

7 EXPERIMENTAL ANALYSIS

In our experimental study, we conducted an end-to-end evaluation of Greenplum with PXF running against three different external data sources: AWS S3, Hadoop, and another Greenplum instance.

7.1 TPC-H Benchmark

Our experiments are based on the TPC-H benchmark [2]. TPC-H is a widely-adopted decision support benchmark that consists of a set of business-oriented ad hoc queries. TPC-H consists of 8 tables and 21 query templates that can well represent a modern decision-supporting system and is an excellent benchmark for testing query optimizers. The rich SQL syntax (correlated subqueries, joins, `CASE` statement, aggregates, etc.) in the TPC-H queries is a good starting point for SQL compliance tests for any query engine.

7.2 Read and Write to Hadoop

7.2.1 Objective. We first study the read and write performance of `TPC-H LINEITEM` data stored in different file formats and sizes. More specifically, the popular CSV and Parquet file formats.

7.2.2 Experimental Setup. The setup included a Greenplum and a Dataproc [9] cluster running on Google Cloud. The Greenplum instance consists of a total of 16 hosts: 1 master and 15 segment hosts. Each segment host has 8 segments, for a total of 120 segments in the Greenplum cluster. Each Greenplum host uses a n1-highmem-8 instance type with 1000GB disks. The Dataproc cluster consists of 25 nodes running on n1-standard-2 instance types with 1000GB disks per node and runs Apache Hadoop v. 2.9.2. We generate `TPC-H LINEITEM CSV` datasets with scale factors of 1–5, 10, 30, 50, 100, 300, 500, 1000, 1500, and 3000 and store the dataset in Hadoop.

We create readable external tables for each scale factor using the PXF Hadoop connector. For each scale factor, we issue `SELECT l_linenumber` queries reading the entire dataset, and record the times. Next, we create writable external tables for each scale factor using the Hadoop connector. For each scale factor, we issue `INSERT` queries, unloading the entire dataset to the Dataproc cluster using `CSV` format and record the times. We then repeat this exercise for the Parquet format. Finally, we create readable external tables to read the Parquet files we previously wrote and record the times.

7.2.3 Analysis. We observe that PXF performance is linear as a function of the data size as shown in Figure 9. For read queries, the `l_linenumber` column is projected for the CSV and Parquet datasets. PXF does not perform any projection on CSV file formats, and the projections are deferred to Greenplum for processing. For Parquet files, PXF supports column projection that speedup reads. Parquet being a columnar storage, reads are faster due to reduced disk read operations, and projection further reduces the data being transferred to Greenplum via PXF. These results can influence how data engineers decide to store files on Hadoop. If reads are to be optimized, then Parquet formats might be preferred over CSV.

For `INSERT` queries, all columns are written to Hadoop. For inserts, PXF incurs CPU and application memory overhead to build the Parquet files; and then to write the Parquet files to Hadoop. Whereas, in the case of CSV, PXF only acts as a proxy between Greenplum and Hadoop, since Greenplum already serializes records as CSV files.

Figure 9: Hadoop Load and Unload of CSV and Parquet

7.3 JDBC Partitioning

7.3.1 Objective. Next, we highlight the importance of designing a good partitioning strategy to produce better transfer speedups and increase overall performance on JDBC reads using PXF.

7.3.2 Experimental Setup. We provision two Greenplum clusters (*cluster1* and *cluster2*) in Google Cloud, where each cluster has 1 master and 7 segment hosts. Each segment host has 4 segments for a total of 28 segments. We choose an n1-highmem-4 compute instance type with 100GB of disk space each. The two clusters are deployed in Google's us-central1-a region within the same subnet. We load TPC-H data in *cluster1*, while in *cluster2*, we create external tables that access TPC-H data residing in *cluster1* using the PXF JDBC connector. We create a set of TPC-H external tables with and without a JDBC partitioning strategy. The DDLs for the external tables for TPC-H LINEITEM table, with and without a partitioning schema are shown below:

```
CREATE EXTERNAL TABLE lineitem_external (LIKE lineitem)
LOCATION('pxf://lineitem?PROFILE=Jdbc&SERVER=greenplum')
FORMAT 'CUSTOM' (formatter='pxfwritable_import');
```

```
CREATE EXTERNAL TABLE lineitem_partitioned (LIKE lineitem)
LOCATION('pxf://lineitem?PROFILE=Jdbc&
PARTITION_BY=gp_segment_id:int&
RANGE=0:27&INTERVAL=1&SERVER=greenplum')
FORMAT 'CUSTOM' (formatter='pxfwritable_import');
```

When querying the partitioned table, the PXF JDBC connector issues 31 queries to *cluster1*:

- WHERE gp_segment_id < 0
- WHERE gp_segment_id >= 0 AND gp_segment_id < 1
- ...
- WHERE gp_segment_id >= 26 AND gp_segment_id < 27
- WHERE gp_segment_id >= 27
- WHERE gp_segment_id IS NULL

7.3.3 Analysis. We first run a simple INSERT INTO SELECT * FROM query to compare the performance of the non-partitioned table vs the partitioned table illustrated above. We observe a 6x transfer speedup utilizing the Greenplum's internal distribution policy partitioning strategy.

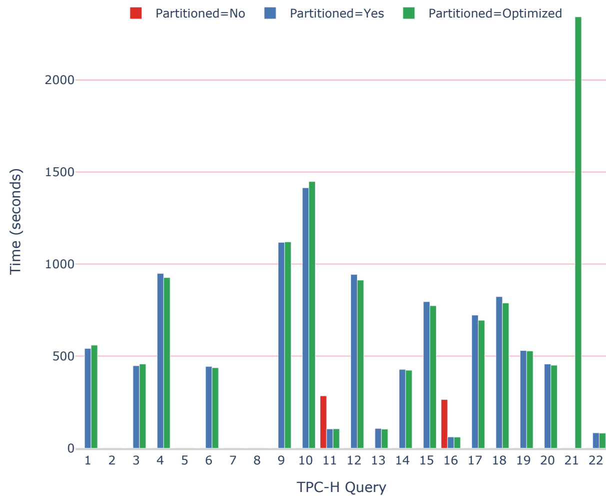

Figure 10: JDBC Partitioning: TPC-H SF100 Performance

To better assess the performance gains obtained by partitioning on Greenplum's internal distribution policy, we benchmark three scenarios on TPC-H queries: no partitioning; partitioning; and optimized partitioning. First, we run TPC-H queries on external tables without any partitioning. As shown in Figure 10, only queries 11 and 16 finish in under forty minutes, but these queries do not show acceptable performance. We then repeat the benchmark by partitioning all TPC-H tables using Greenplum's internal distribution policy as the partitioning strategy. With this partitioning strategy, a majority of queries show significant performance improvements from the previous benchmark, with queries completing in well under our forty minute limit. Lastly, we ran the workload over a mix of partitioned and non-partitioned tables. Here, tables with more than 1000 rows are partitioned, while tables with fewer than 1000 rows do not use any partitioning strategy. We call this strategy optimized. With this approach, query 21 is able to finish in under our timeout threshold. However, execution time increased slightly in some cases and degraded marginally for queries 1, 3 and 10.

7.4 PXF S3 Select Feature

7.4.1 Objective. We next study the performance impact of PXF's S3 Select support to access data on AWS S3.

7.4.2 Experimental Setup. For this study, we provision a single Greenplum cluster in Google Cloud with a similar configuration as in Section 7.3.2. We create TPC-H external tables with S3 Select disabled and enabled using the following DDL statements:

```
CREATE EXTERNAL TABLE lineitem_s3 (LIKE lineitem)
LOCATION ('pxf://bucket/tpch_data/100/?
PROFILE=s3:csv&SERVER=s3') FORMAT 'CSV';

CREATE EXTERNAL TABLE lineitem_s3_select_on (LIKE lineitem)
LOCATION ('pxf://bucket/tpch_data/100/?
PROFILE=s3:csv&SERVER=s3&S3_SELECT=ON') FORMAT 'CSV';

CREATE EXTERNAL TABLE lineitem_s3_select_auto (LIKE lineitem)
LOCATION ('pxf://bucket/tpch_data/100/?
PROFILE=s3:csv&SERVER=s3&S3_SELECT=AUTO') FORMAT 'CSV';
```

The `PROFILE=s3:csv` clause instructs PXF to read the file from S3 in CSV format. Currently, PXF supports S3 Select access to CSV and Parquet format object files stored on Amazon S3. Next, we ran TPC-H queries with `'S3 Select = Off'`, `'S3 Select = On'`, and `'S3 Select = Auto'`.

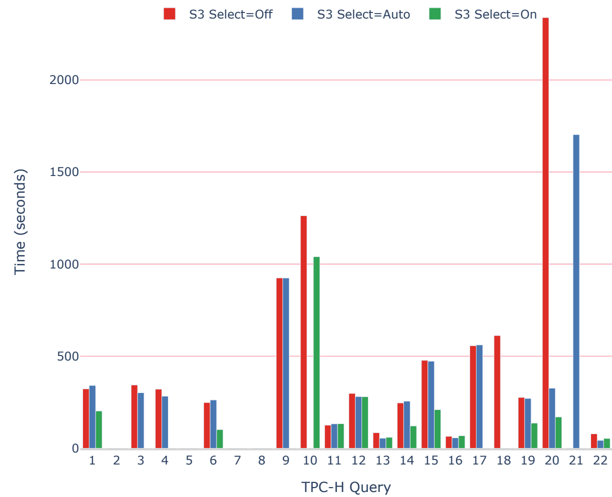

Figure 11: S3: TPC-H SF100 Performance

7.4.3 Analysis. Figure 11 shows queries with `'S3 Select = On'` are generally faster. However, we noticed that some queries running with `'S3 Select = On'` failed when we had high concurrency to the S3 service. When looking at the plan, we noticed that the `LINEITEM` table was scanned two times, which produced two external calls to the same dataset. This caused the S3 Select service to return incomplete results of the data, causing PXF to error out.

With `'S3 Select = Auto'`, PXF only uses S3 Select when it's adavantageous (predicate and column pushdown) and so S3 Select is disabled for the `LINEITEM` table. This minimized the errors encountered when running with `'S3 Select = On'`. This approach resulted in fewer errors from S3 Select, and more queries completing successfully but did not meet performance expectations (with the exception of queries 20, 21, and 22).

It is worth noting that for queries running without S3 Select, the time spent reading files of different sizes from S3 is consistent. In contrast, queries with S3 Select enabled tend to have a relatively slower performance the first time they run, while subsequent runs of the same query experience a speedup.

8 RELATED WORK

Federated Query Processing. The problem of querying external data sources from a SQL prompt is not unique to Greenplum. In 2003, the SQL standard was expanded to include Foreign Data Wrappers (FDWs) [3] for the purpose of managing external data sources. An FDW is a "contrib" module that can be plugged into Postgres to connect to one or more types of external data sources. Since that time, FDWs have been implemented for various external data sources including Postgres[15]. PXF offers two advantages over the currently available FDW implementations. First, PXF is able to connect to many different external data sources, including those on all the major clouds, and can access data in various different formats. In contrast, Postgres FDW implementations tend to be tightly-focused and limited in their application. Second, because PXF interfaces with Greenplum, it is able to access and write data in parallel across Greenplum segments. FDWs written for Postgres, however, must transfer data through a single Postgres instance.

There are plans within the Greenplum community to fully support the FDW API in a future release. At that time, it will be possible to access PXF through a PXF FDW module (`pxf_fdw`), which will allow users to use foreign tables that connect to PXF in parallel, as is currently possible with Greenplum-only external tables.

Presto [6] is a distributed SQL query engine which allows users to query a variety of data sources such as Hadoop, AWS S3, Alluxio, MySQL, Cassandra, Kafka, and MongoDB. Similar to PXF, users can access data from multiple data sources within a single query.

Database As A Service (DBaaS) Solutions. Snowflake [7] runs exclusively on the cloud, but does not provide federated query functionality. PXF and Snowflake are similar in that they can both query data stored in Amazon S3, but that is where the similarities end. The Amazon RedShift [11] and Microsoft Azure SQL Data Warehouse [13] cloud-based offerings allow querying of data in S3 and Azure, respectively. While they do support various data types, they lack federation of data. Google's BigQuery [16], on the other hand, does offer data federation, but only among the Google Drive (spreadsheets), Google Bigtable (transactional DB), and Google Cloud Storage (object storage) Google products.

9 SUMMARY

Increasingly, data is being stored in disparate sources: cloud object stores, transactional databases, Hadoop data lakes, as well as analytical data warehouses. Answering critical business questions requires retrieving, merging, filtering, and aggregating data from these multi-temperature heterogenous repositories. In this paper, we propose Platform Extension Framework (PXF) that supports parallel, high throughput data access and federated queries across heterogeneous data sources.

ACKNOWLEDGMENTS

We would like to thank Dov Dorin, Alon Goldshuv, Noa Horn, Oleksandr Diachenko, Kong-Yew Chan, Kavinder Dhaliwal and Raymond Yin for their contributions in building PXF. In addition, we want to thank Oz Basarir, Goutam Tadi, David Sharp, Karen Huddleston, and Jason Vigil for getting PXF successfully running on GP4K. Lastly, we appreciate Robert Glithero for his feedback.

REFERENCES

[1] 1999. Apache Tomcat. http://tomcat.apache.org/.
[2] 2001. TPC-H. http://www.tpc.org/tpch.
[3] 2003. Information technology - Database languages - SQL - Part 9: Management of External Data (SQL/MED). https://www.iso.org/standard/34136.html
[4] 2018. What is Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.
[5] Microsoft Azure. 2015. What is Azure Blob storage? https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-overview
[6] Lydia Chan. 2013. Presto: Interacting with petabytes of data at Facebook. http://prestodb.io
[7] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *ACM SIGMOD*. 215–226.
[8] Google. 2015. Google Storage API Reference. https://cloud.google.com/storage/docs/apis
[9] Google. 2018. Google Dataproc API Reference. https://cloud.google.com/dataproc/docs/
[10] Greenplum. 2015. The World's First Open-Source and Massively Parallel Data Platform. https://greenplum.org/
[11] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *ACM SIGMOD*. 1917–1923.
[12] Randall Hunt. 2018. S3 Select and Glacier Select. https://aws.amazon.com/blogs/aws/s3-glacier-select/
[13] Leonard G. Lobel and Eric D. Boyd. 2014. *Microsoft Azure SQL Database Step by Step* (1st ed.). Microsoft Press, Redmond, WA, USA.
[14] Jemish Patel, Goutam Tadi, Oz Basarir, Lawrence Hamel, David Sharp, Fei Yang, and Xin Zhang. 2019. Pivotal Greenplum© for Kubernetes: Demonstration of Managing Greenplum Database on Kubernetes. In *ACM SIGMOD*. 1969–1972.
[15] PostgreSQL. 2019. PostgreSQL:The World's Most Advanced Open Source Relational Database. http://www.postgresql.org
[16] Kazunori Sato. 2012. *An Inside Look at Google BigQuery*. Technical Report. https://cloud.google.com/files/BigQueryTechnicalWP.pdf
[17] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST)*. 1–10.
[18] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *ACM SIGMOD*. 337–348.
[19] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*. 2–11.