QFlock – Design Documentation

1. Abstract

The amount of geographically distributed massive data is constantly increasing. Novel frameworks are expected to process geographically distributed data at their locations without moving entire raw datasets to a single location. In this document, we discuss challenges, requirements, and solutions in designing geographically distributed data processing frameworks and protocols.

Specifically, we will talk about a solution we have been developing called QFlock, which will allow for distributing and even federating queries across data centers. The idea here being to make it possible for the user at a single pane of glass to issue a query across data that is distributed across some set of data centers. And to not only allow this for homogenous data schemas, but also allow the federation of queries across data sets that are not homogeneous.

2. Document Layout

We will be performing research on a variety of approaches. As each approach is completed, we will update this section with details.

Versions 1 and 2 are detailed in sections 3-9

Version 3 is described in section 10

3. Overall Architecture

Here we will describe the overall architecture and the relationships between the major components. We will keep the descriptions at a high level. In the below diagrams the green color indicates components that we have written.

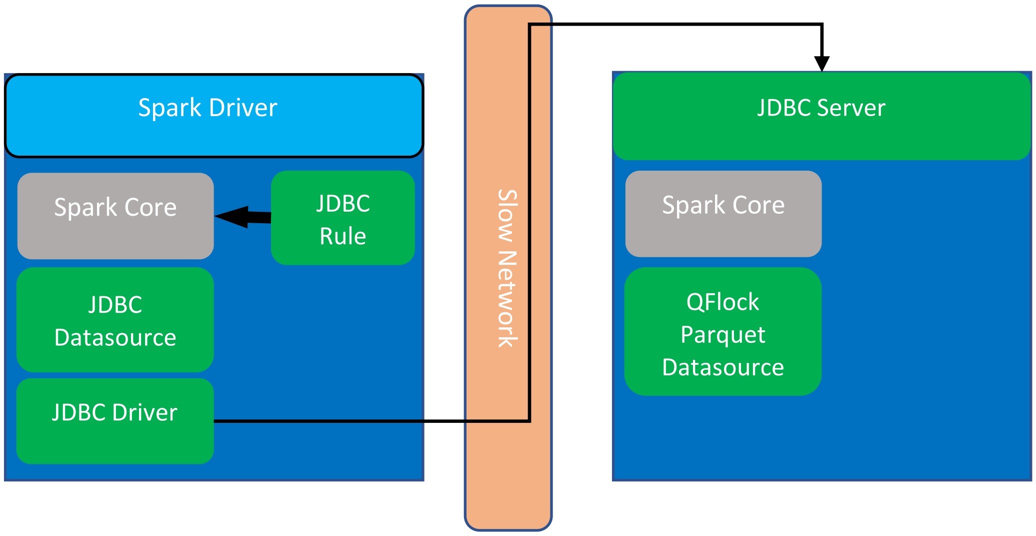

Figure 1. Overall Architecture JDBC Rule, JDBC Datasource, JDBC Driver, JDBC Server

4. QFlock JDBC Rule

The QFlock Rule is a custom Spark Rule, whose purpose is to intelligently detect when it is necessary to direct a query to our remote QFlock JDBC Server. Once the determination is made, the rule transforms the plan to inject the QFlock JDBC Datasource, which will execute the query using the remote JDBC Server.

4.1. Rule Engagement Criteria

The first task the JDBC Rule needs to complete is to parse the logical plan and decide if it should engage at all. The criteria for engagement of the rule are:

1. Is the data remote?
   We currently only engage for remote data since our JDBC Server is only available for remote data.
2. Can and should we engage to reduce data?
   This criterion simply checks if the queries for this plan can be handled. There is a small set of non-trivial query filter operators we still need to add support for, and for those we will not engage. In cases where the rule does not engage, the
3. It is worth pointing out that additional criteria are a potential focus of future efforts, so expect future enhancements here.

4.2. Transforming Plan

When the rule is engaged, it needs to transform the Plan to include our QFlock JDBC Datasource for any relation that is located remotely and requires optimization.

More specifically, the rule needs to transform all project, filter relations. All these relations need to be replaced with an instance of our JDBC Relation. So, for example, a standard Spark project, filter relation such as the below:

```
- Project [i_item_sk#145L, i_item_id#146, i_item_desc#149, i_current_price#150]
   - Filter((((i_current_price#150 >= 68.0) AND (i_current_price#150 <= 98.0)) AND
i_manufact_id#158L IN (677,940,694,808))
    - Relation(
tpcds.item[i_item_sk#145L,i_item_id#146,i_rec_start_date#147,i_rec_end_date#148,i_item_desc#149,i
_current_price#150,i_wholesale_cost#151,i_brand_id#152L,i_brand#153,i_class_id#154L,i_class#155,i
_category_id#156L,i_category#157,i_manufact_id#158L,i_manufact#159,i_size#160,i_formulation#161,i
_color#162,i_units#163,i_container#164,i_manager_id#165L,i_product_name#166] parquet)
```

Might be replaced with a JDBC Relation such as this:

```
RelationV2[i_item_sk#284L, i_item_id#285, i_item_desc#288, i_current_price#289] class
com.github.qflock.extensions.jdbc.QflockJdbcBatchTable
```

4.3. Extracting query

As part of transforming the plan, the rule also needs to extract the filter and project. After extraction, there is additional logic to then re-form the SQL Query. This SQL query will be set as a parameter to the QFlock JDBC Datasource.

5. QFlock JDBC Datasource

The QFlock JDBC Datasource is a custom V2 Spark Datasource, which uses the QFlock JDBC Driver in order to communicate via the JDBC API with the remote QFlock JDBC Server.

5.1. Partitioning

Partitioning is extremely important since it allows for breaking up of a single large partition into pieces that can be processed in parallel. This is how Spark achieves its parallelism. We are using one partition per Parquet row group.

To ensure that JDBC Server honors the partitioning, we pass the row group offset and row group count to the JDBC Server along with each query. In addition, since Spark has no way to execute a query on a subset of row groups, the JDBC Server utilizes a new data source we created to only run the indicated query on a subset of row groups.

5.2. JDBC Connection

JDBC (Java Database Connectivity) defines a standard interface to be used by all drivers. The JDBC Datasource uses this standard interface to communicate with our driver. The API being used here is completely standard JDBC.

JDBC has a notion of a “connection” which needs to be established. That connection object can then be used to create a statement object. The statement object can then be used to execute a query and retrieve a result.

The query itself is passed as an argument to the execute query command, but what about other parameters that need to be sent to the JDBC Server? These can be set as properties on the connection. See below for more details on the required parameters in the JDBC Driver API.

5.3. Options

There are a few supported options for the JDBC Datasource. Keep in mind that since this datasource is inserted by our JDBC Rule, these parameters are mostly set by that Rule.

Parameter	Description	Example
format	The format of the file	parquet
url	The URL of JDBC	jdbc:qflock://hostname:1433/database
driver	The java class for the JDBC driver	com.github.qflock.jdbc.QflockDriver
query	The query text to execute	SELECT * FROM call_center
numRowGroups	The total number of row groups in the table	42
schema	The schema of the table	ss_quantity:long:true,ss_list_price:double:true

5.4. Global Options

The only global option we support is: "qflockJdbcUrl"

This is set on the spark session via something such as the below:

```
spark = pyspark.sql.SparkSession                              \
                     .builder                                 \
                     .appName(app_name)                       \
                     .config("qflockJdbcUrl",                 \
                              "jdbc:qflock://server:1433/db") \
                     .enableHiveSupport()                     \
                     .getOrCreate()
```

6. QFlock JDBC Driver

JDBC[1] (Java Database Connectivity) provides Java access to any type of data source. In our case we have our own JDBC Driver, which allows access from Java to our JDBC Server. The QFlock JDBC Driver is a standard Java JDBC Driver, which adheres to the standard JDBC interfaces, and uses our QFlock JDBC Thrift API to communicate with the QFlock JDBC Server.

The JDBC Driver API is very rich. We have focused on implementing and enabling the key APIs, which are required by Spark to enable executing queries. Other APIs will be considered for implementation later as needed.

6.1. Connect

6.1.1 Connect String

When the JDBC connect method is called, it can choose to accept or reject the connect based on the type of connection string. It is through this API that a driver can choose which connect requests to accept and which to allow other drivers a chance to process.

In the case of our driver, our connect string is expected to have the form:

jdbc:qflock://qflock-jdbc-server:1433/database

Where the fields are defined as:

- jdbc:qflock: is the sequence that identifies this as a connect to be processed by our Driver.
- qflock-jdbc-server - is the host name of the server that contains our JDBC server. In most of our setups it is something like qflock-jdbc-dc2.
- 1433 - is the port that our JDBC server listens on.
- database - is the name of the database we are trying to access on our JDBC server.

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

6.1.2 Connect Parameters

The JDBC Driver's connect method takes a set of parameters that can be defined by the client.

The additional properties that are required to be set for our Driver include:

- rowGroupOffset
- rowGroupCount
- tableName

These parameters define the options for the operations to be executed on this connection. These parameters are communicated to the server as part of the connect method execution, and then can be used later by the server. For instance, these can be used by the server when a query is executed.

7. QFlock JDBC Thrift API

This QFlock JDBC Thrift API allows the JDBC Driver to communicate with the JDBC server.

Since we are using a Thrift[2] API, it means that we will utilize Thrift libraries for all aspects of the connection between client and server. It's worth also mentioning that since our client Driver is written in Java and our server is written in Python, this solution fits our scenario well.

We have defined a Thrift API between client and server, based on the API found at https://github.com/damiencarol/thrift-jdbc-server

This API provides a basic Thrift API using a JDBC object model. We have taken and adopted this approach, extending it extensively where needed to fit our solution.

[2] The Apache Thrift software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages. https://thrift.apache.org/

8. QFlock JDBC Server

The QFlock JDBC Server is a component which uses our QFlock JDBC Thrift API to accept JDBC type requests. The server executes these requests largely using Spark itself and returns the results using the same QFlock JDBC Thrift API.

8.1. Query Handling

When the server receives a new request to perform a query, it uses pyspark to execute the query. Once the query data is retrieved, we convert the data into a pandas dataframe and NumPy array before converting it to a sequence of bytes, suitable for compression. We found this sequence of operations to be the most efficient for moving from a dataframe to a sequence of bytes. Each column is converted separately before being placed back into the thrift API which has an array of columns to be returned.

8.2. Compression

Compression is performed using the Zstandard library. Each column is compressed separately provides the best compression ratio, since the data type and data distribution have the potential to be very different between different columns.

8.3. Temporary Views and Request IDs

With Spark, the only way to execute an SQL query against a table is to use a view. View creation is expensive, and not something we can afford to do for every query. Therefore, the JDBC Server creates tables ahead of time which can be used when the queries arrive.

Every query also has a row group range (offset, count), which needs to be communicated to the QFlock Parquet Datasource. Normally we specify the datasource parameters at dataframe creation time. However, since the views (and hence the dataframe the view is based on) are computed ahead of time we cannot specify the row group parameters in the dataframe.

Instead, we have created a notion of request ID. A request ID is essentially a number that is associated with each dataframe/view. Prior to sending a request to the datasource, the JDBC server will fetch a request ID and give the datasource the row group offset/count to use for that request ID. Then when the query arrives at the datasource it will know which row group offset/count to use for that query.

9. QFlock Parquet Datasource

The QFlock Parquet Datasource allows access to specific row groups of a parquet file. For example, in our case the QFlock JDBC Server will use the QFlock Parquet Datasource to execute queries for certain row groups.

Here is an example of how to use the QFlock Parquet datasource.

```
df = self._spark.read                           \
                .format("qflockDs")             \
                .option("format", "parquet")    \
                .option("schema", schema)       \
                .option("tableName", tableName) \
                .option("path", file_path)      \
                .option("dbName", dbName)       \
                .option("requestId", request_id) \
                .load()
```

9.1. APIs

`QflockTableDescriptor.addTable(tableName, view_count)`

This API allows the data source to accept a new table for tracking row group ranges.

`QflockTableDescriptor.getTableDescriptor(tableName): QflockTableDescriptor`

This API allows the data source to return a table descriptor, which can be used for later operations.

`QflockTableDescriptor.fillRequestInfo(offset, count): Int`

This API returns an integer Request ID which is now tracking a specific row group offset/count.

`QflockTableDescriptor.freeRequest(requestID: Int)`

This API allows the user to free up a request.

10. Optimizations

Version 3 of our work enhances the approaches from prior versions with new optimizations designed to fit a variety of common use cases.

We have three approaches which we will list here and explain in the following sections.

- Aggregate pushdown
- Caching of query data
- Pushdown of additional Projects

10.1. Aggregate Pushdown

Aggregate pushdown is a technique where an aggregate operation like MIN, MAX, SUM or COUNT is fully pushed down to the data source and even to the remote server. In our case this means pushing the aggregate to the JDBC server to be executed by the remote Spark cluster, which has access to the data. This approach takes advantage of the remote Spark's computing power to reduce the size of the data before transferring it back to the client. In many cases the aggregate pushdown reduces the result size by 90% or more since an aggregate operation result is a tiny fraction of the data being evaluated. This will result in significant savings especially if the network between the data centers is relatively slower than local data center network.

In our testing, we measured significant performance gains in 3 of the TPC-DS tests, 10, 35 and 69.

10.1.1 Details

Aggregate pushdown starts with our Spark Rule, which pattern matches the Aggregate operation in the Logical plan. The aggregate operation is clipped out of the plan to be replaced with a relation representing an operation to our data source. In addition, all the necessary parameters are constructed for transmission to our JDBC Data source and eventually to our JDBC Server. The most important of these is the query itself which will be executed on the remote Spark cluster. The QFlock JDBC Rule constructs a SQL query from the logical plan Catalyst nodes representing the aggregate. This SQL query is sent to the remote JDBC server to be executed.

10.2. Query Caching

In our research we found that some TPC-DS queries tended to issue the same query to the data source multiple times. In theory if we cached these queries in some way on the client side, we could help to reduce the network transfer even further and provide significant gains.

In fact, we found that providing automatic caching of repeated queries provides significant gains in many cases. One example of this is test 47, which shows repeated queries and benefits from this approach.

10.2.1 Details

Our caching implementation starts with detecting the need the need for caching during our JDBC Rule evaluation. The number of times a specific query is being pushed down is tracked here. This will allow us to decide if caching is necessary or not. Later when the data is fetched at the JDBC data source we can decide if the data needs to be cached based upon a few factors such as:

- How many times the data will be needed. Caching will be useful if the data is needed two or more times.
- How much data is being cached. We track the total amount of data being cached and if we exceed it, we will stop caching until space frees up.

Another advantage of knowing how many times the data is referenced is that we can drop the data once it is no longer needed. We track the number of references to the data and when it reaches the expected max, we drop the data, thereby freeing up more memory for caching other data.

10.3. Second Pass Pushdown of Projects

As we mentioned previously, pushdown of operations is done using a Rule Based approach. With this approach our custom Spark Rule will parse the logical plan and enable pushdown using well established Spark mechanisms. These mechanisms, which we implement in our Spark Rule, transform the logical plan by inserting our data source (with pushdown) in the place of the operations to be pushed. Our rule runs like any other Spark Rule, it receives a Logical Plan as input and returns a Logical Plan as output. Our rule will be run by Spark automatically, whenever Spark determines it needs to run all the rules.

In our research we discovered that our Spark Rule needs to be able to pushdown projects to our data source in a second pass or invocation of our Rule. Why does this occur? It is needed since there are cases where Spark runs our rule multiple times for the same query. The first time the rule runs, we insert our data source with, for example, pushdown of filter. But then the next time our rule runs, it detects additional new operations such as projects, which Spark added since the last time our rule ran. To enable optimal performance, these new projects also need to be pushed down to our JDBC data source.

This optimization enhances our rule to detect these additional operators and successfully transform a logical plan which already has pushdowns to include the new pushdowns.

10.3.1 Details

This optimization enhances our existing JDBC rule to include a second pass of project pushdown. This enhances the existing pattern matching in our rule to include matching on an already inserted JDBC data source relation which has additional inserted projects above it. After detecting the pattern, we then check that the projects themselves are suitable for pushdown and supported by our JDBC data source. Assuming all checks pass, we then insert a new relation, with a new enhanced query to include the additional projects.

11. References

Hadoop HDFS: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction

Spark: https://spark.apache.org/