NDP – Design Documentation

I. Abstract

NDP - Near Data Processing is a technique where processing of data is pushed closer to the source of the data in an attempt to leverage locality and limit the a) data transfer and b) processing cycles needed across the set of data.

Our approach will consider use cases on Spark, and specifically focus on the use cases where Spark is processing data using SQL (Structured Query Language). This is typical of data stored in a tabular format, and now very popular in Big Data applications and analytics. Today, these Spark cases use SQL queries, which operate on tables (see background for more detail). These cases require reading very large tables from storage in order to perform relational queries on the data.

Our approach will consist of **pushing down** portions of the SQL query to the storage itself so that the storage can perform the operations of filter, project, and aggregate and thereby limit the data transfer of these tables back to the compute node. This limiting of data transfer also has a secondary benefit of limiting the amount of data needing to be processed on the compute node after the fetch of data is complete.

II. Assumptions and Non-goals

We will assume the audience has had some exposure to Spark, relational databases, and NDP techniques. While we will cover the Spark internals related to the Datasource V2, and we do make every effort to define the important aspects of Spark we will encounter, is not a goal of this document to be an exhaustive Spark internals document. Thus, we will not cover exhaustively all references that we make to Spark internals. For the aspects we refer to but do not go into in detail, we will leave it as an exercise for the reader to investigate the code in further detail using the class and method names we have provided.[1]

This document will sprinkle in Scala code and terms of art from Scala to illustrate various interfaces. Just keep in mind that some familiarity with these terms is helpful since we will not go into any detail regarding Scala itself.

A convention we use throughout is to insert a colon and blank line before code like this:

```
println("A block of scala code")
```

[1] The Apache Spark code is located here: https://github.com/apache/spark

III. Background

In a typical Spark use case, the user starts the application (called driver in Spark). Then Spark distributes processing amongst its worker nodes, all of which typically exist on different machines (Physical, Virtual/Container). In a typical use case as shown below in ***Figure 1***, processing occurs in the worker nodes. These worker nodes issue requests to the storage to read in the tables being processed. Once the data requests are satisfied, the workers process the SQL operations on the tables and essentially traverse the data in order to perform the SQL query.

For example, take a typical SQL Query of:
SELECT name FROM employee WHERE department = "HR"

In such an example, the entire table (in this case the "employee" table) needs to be read from storage in order that the query can be processed. Also, dissecting the query further shows that there are two parts to the query involved in selecting the data from the table:
SELECT name is what we call a ***project*** since we are selecting a specific column of data.
And WHERE department = "HR" is what we call a ***filter***, which chooses specific rows of data.

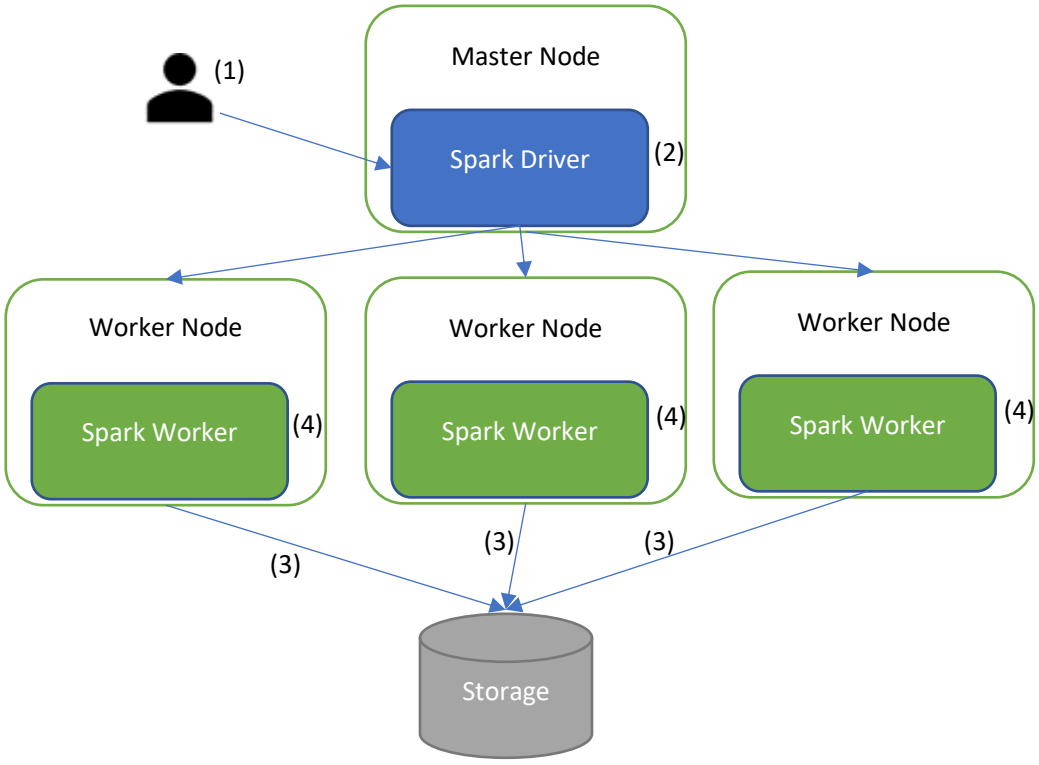

Figure 1*:* User (1) starts application. Spark (2) distributes jobs across its workers. The workers (3) reads data from storage and (4) perform the operations on the data.

IV. Introduction

Our general approach with NDP, will be to push down specific portions of the SQL query to the storage.

Pushdown is a performance optimization that prunes extraneous data while reading from a data source to reduce the amount of data to scan and read for queries with supported expressions. Pruning data reduces the I/O, CPU, and network overhead to optimize query performance.

In more concrete terms, pushdown in this context indicates sending an SQL query to storage along with the read operation. The storage will then process the data and apply the query local to the data before returning the data to the client, which in this case is Spark.

In the example above we identified a ***project*** and a ***filter***. Both can be pushed down to the storage along with aggregate operators.

In ***Figure 2***, we show an example case of pushdown with Spark. We will get to the details of this approach later. This is meant to merely introduce the general ideas and approach for now.

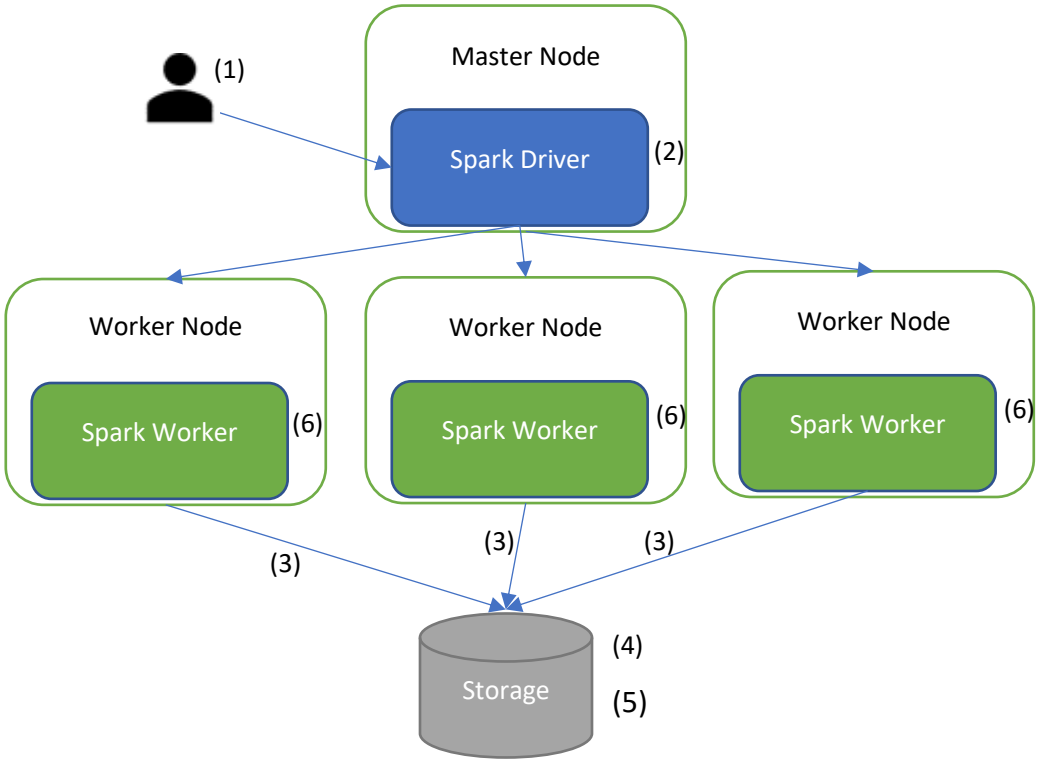

Figure 2*:* User (1) starts application. Spark (2) distributes jobs across its workers. The workers (3) fetch data from storage and push down operations. The (4) storage reads the data and (5) applies the query. When the worker gets data back and processes the data (6), there is less data to process. This results in less data to be transferred and processed.

V. Architecture

We will introduce the general architecture of our approach and outline the major components, which we will break down in detail as we proceed later.

To more easily understand the components in our architecture we will follow through an example of a query with pushdown.

Our high-level architecture starts with the user launching the application, which invokes Spark.

We need Spark to provide information about the SQL query in order that we can communicate this query to the storage along with the read request. Spark provides an API called the Data Source V2 API, which allows a data source to be able to get callbacks from Spark for such reasons as filter pushdown, project pushdown, and aggregate pushdown. We leverage this API by creating a Spark V2 Datasource for the purpose of NDP pushdown. This datasource will be responsible for adhering to the V2 Datasource API for such reasons as 1) pushdown, 2) partition inquiry, 3) data row read, etc. This datasource will also need to adhere to the API provided by the NDP server through its NDP Client.

The NDP client is a module that provides a user access to the services of the NDP server, such as the service of read with pushdown.

The NDP Server provides the service of read with pushdown by 1) reading the data and 2) utilizing the SQL engine to operate on the data with the pushed down query.

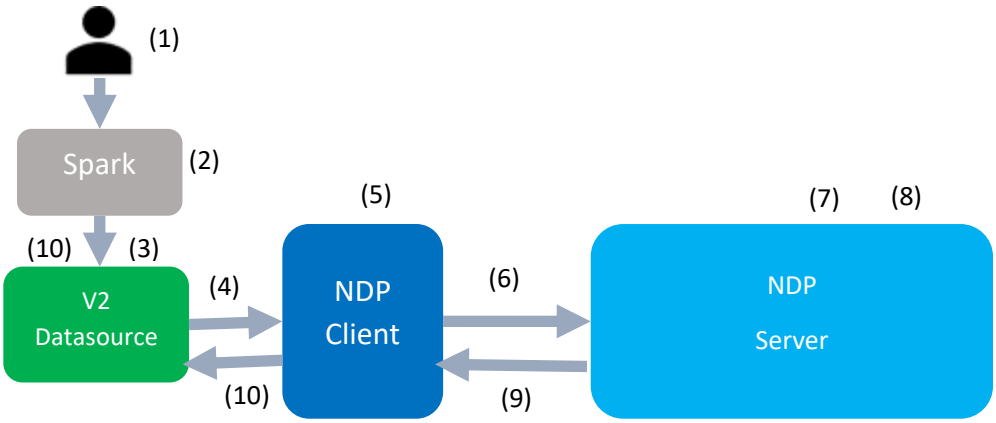

Figure 3: NDP High Level Component Diagram. User (1) invokes application using our new datasource, which uses Spark (2) to execute a query. Spark detects and invokes our V2 datasource (3) in order to perform pushdown, and in order to read the data (4). The NDP Client code (5) provides the Datasource an API in order to (6) read with pushdown from the NDP Server. The NDP server performs a read of the data (7), applying the SQL query (8) before returning the data (9). The V2 datasource receives the data back from NDP (10), and forms the appropriate rows before returning the data to spark at (11).

VI. Spark V2 Datasource

The V2 Datasource we created supports filter, project, and aggregate pushdowns. As of Spark version 3.1.1[2], the datasource API only supports filter and project pushdowns. However, work is underway[3] to add aggregate pushdown support to Spark, and we decided to include this patch with aggregate support in our work, under the assumption that this work will eventually become a permanent part of Spark.

In order to understand the nature of a Spark V2 Datasource, we need to introduce a bit of detail concerning Spark itself. Specifically, we will discuss the normal sequence of handling for a Spark Query and how the V2 Datasource is able to transform the Logical Plan.

A. Starting the Spark Query

In ***Figure 4,*** we list the sequence of steps in handling a Spark SQL Query. The class that starts the entire sequence and drives it forward is called a QueryExecution[4]. QueryExecution calls the SparkOptimizer[5] in order to transform an analyzed LogicalPlan into an optimized LogicalPlan. The SparkOptimizer performs all the logical optimizations at the step named Logical Optimizations in the diagram. This will iterate across the set of possible optimizations until all are applied, and the result is an Optimized Logical Plan. It is worth mentioning that another name for optimization in this context is Rule, and you will often hear Rule and optimization used interchangeably. In fact, "Rule" is the class that each of these optimizations derives from[6].

[2] Spark 3.1.1 release on 3/2/2021. https://spark.apache.org/news/spark-3-1-1-released.html
[3] Aggregate push down is being added as part of SPARK-23390 in this PR: https://github.com/apache/spark/pull/29695
[4] According to Spark QueryExecution is: "The primary workflow for executing relational queries using Spark. Designed to allow easy access to the intermediate phases of query execution for developers." See QueryExecution.scala.
[5] It is worth mentioning that the SparkOptimizer class contains within it all of the various lists of rules that it will execute along with the ordering of these specific rules.
[6] See org.apache.spark.sql.catalyst.rules.Rule in Rule.scala

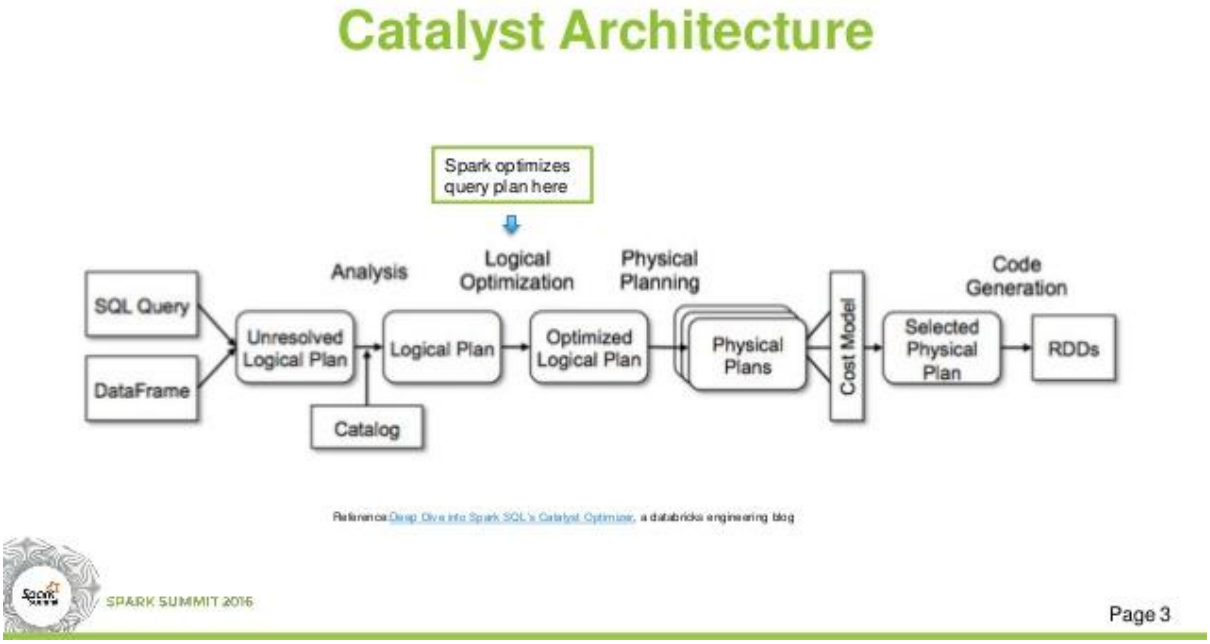

Figure 4: Spark Catalyst Architecture has many steps. The Logical Optimization step is the most interesting for a V2 Datasource since this is the stage where the V2ScanRelationPushDown Rule is applied, which uses the V2 Datasource to transform the logical plan.

B. V2ScanRelationPushDown

One of the LogicalOptimizations is called V2ScanRelationPushDown:

```
object V2ScanRelationPushDown extends Rule[LogicalPlan]
```

This V2ScanRelationPushDown "Rule" object is just one of the many Rules invoked during the Logical Optimization step in order to transform the Logical Plan into the Optimized Logical Plan. The V2ScanRelationPushDown will only influence a V2 datasource, since the APIs this rule uses to interact with a datasource use the V2 API.

C. First Steps: DataFrameReader, TableProvider, and Table

The first step in defining a V2 datasource is to define a class which is derived from TableProvider[7], to represent an API to a table.

This TableProvider provides an override called getTable:

```
override def getTable(schema: StructType,
                      transforms: Array[Transform],
                      options: util.Map[String, String]): Table
```

When a query is being processed by Spark, the first interaction with the Data Source v2 is initiated by the DataFrameReader object[8], which is initiated by the user when initiating a read of a Dataframe. The

[7] TableProvider is "the base interface for a V2 Data Source". See TableProvider.java for more details.
[8] A DataFrameReader is an interface used to load a DataFrame from external storage. See DataFrameReader.scala for more details.

DataFrameReader detects our data source, determines that the data source supports V2, fetches our TableProvider object, and asks the data source to create a table object using the API TableProvider.getTable(). The object which getTable() returns is a class we define, which derives from the Table class, and which extends Table with SupportsRead to indicate reading is allowed from the Table. This Table derived class needs to define an override called newScanBuilder:

```
override def newScanBuilder(params: CaseInsensitiveStringMap): ScanBuilder
```

This override returns yet another object we define, which is derived from the ScanBuilder class.

The V2ScanRelationPushDown object (invoked by SparkOptimizer) calls the Table.newScanBuilder() API to create the builder.

D. ScanBuilder

More importantly, this derived ScanBuilder class needs to include mix-in interfaces for any type of pushdown it supports. For example, we define our ScanBuilder derived class using all manner of pushdowns including filter, project, and aggregate[9]. We call our derived class PushdownScanBuilder.

```
class PushdownScanBuilder(schema: StructType,
                          options: util.Map[String, String])
  extends ScanBuilder
    with SupportsPushDownFilters
    with SupportsPushDownRequiredColumns
    with SupportsPushDownAggregates[10]
```

In addition, this PushdownScanBuilder class we define above needs to define overrides for each of the types of push downs it supports:

```
override def pruneColumns(requiredSchema: StructType): Unit
override def pushFilters(filters: Array[Filter]): Array[Filter]
override def pushAggregation(aggregation: Aggregation): Unit[11]
```

The V2ScanRelationPushdown optimization decides what to push down based on which mix-ins are supported by the ScanBuilder object.

As the V2ScanRelationPushdown optimization runs, the data source can expect to receive push down calls for each of the push downs that it supports. These pushdowns are listed below.

1) push down the filters using the pushFilters(Array[Filter]) API.

2) push down the projects for column pruning via the pruneColumns(requiredSchema: StructType)

[9] Just keep in mind that this is also a part of the aggregate pushdown patch.
[10] Just keep in mind that this is also a part of the aggregate pushdown patch.
[11] Just keep in mind that this is also a part of the aggregate pushdown patch.

3) push down the aggregates via pushAggregation(aggregation: Aggregation) [12]

Once the push down is complete, it is expected that the datasource retains state about the filters that were pushed down. This state will be used later by the datasource in order to send the pushdown query as part of the read. The datasource also overrides methods to allow Spark to retrieve the expressions that have been pushed down.

```
override def pushedFilters: Array[Filter]
```

```
override def pushedAggregation(): Aggregation
```[13]

Once the pushdowns are complete, the V2ScanRelationPushdown rule invokes another override of the ScanBuilder class.

```
override def build(): Scan
```

This returns another class that we derive from: The Scan object.

E. Transforming the logical plan

It is worth mentioning that one of the overrides in the Scan object is:

```
override def readSchema(): StructType
```

This allows the V2ScanRelationPushdown to fetch the currently pushed down columns, originally sent via the ScanBuilder.pruneColumns() call.

The V2ScanRelationPushdown can fetch from the ScanBuilder, the current set of pushdown Aggregates and Filters. And from the Scan it can fetch the list of pushed down columns.

Using the above set of information, the V2ScanRelationPushdown object transforms the logical plan to adhere to the current set of pushdowns. What does this mean? It means that Spark transforms the plan to take into account that the datasource, not Spark, will be performing these operations. Spark will remove some operations from the Logical Plan that no longer need to be performed after the data is fetched. This also has the effect of essentially preparing Spark for the format (number of columns and data type of columns), of the data that is going to be returned by the data source.

F. Scan Object

The purpose of the Scan object is to represent an operation on a V2 Datasource.

One of the interesting overrides of the Scan object is:

```
override def planInputPartitions(): Array[InputPartition]
```

This allows the datasource to provide the list of partitions to Spark.

[12] Just keep in mind that this is also a part of the aggregate pushdown patch.
[13] Just keep in mind that this is also a part of the aggregate pushdown patch.

Another override of the Scan object allows returning a PartitionReaderFactory

```
override def createReaderFactory(): PartitionReaderFactory
```

G. PartitionReaderFactory

Next in the lifecycle of the data source, our Scan object gets invoked in the context of a Batch Scan via Scan.createReaderFactory(). The Batch Scan is the Spark object which is responsible for scanning (aka reading) a batch of data from a Data Source V2.

The partitionReaderFactory is significant since it allows for overriding:

```
override def createReader(partition: InputPartition):
PartitionReader[InternalRow]
```

The createReader allows for creation of a PartitionReader[InternalRow], which is what Spark will use in order to read the rows of the table result from the datasource.

H. PartitionReader

Later in the context of the Spark worker, the PartitionReaderFactory's createReader(partition: InputPartition) API is used to create an PartitionReader object for a specific partition. The PartitionReader object extends a PartitionReader for an InternalRow, which is Spark's internal representation of a row object. We can think of the PartitionReader as an iterator across all the rows results for a given partition and has the following interfaces:

The next: API returns true/false depending on if there are rows remaining.

The get: API returns an InternalRow object.

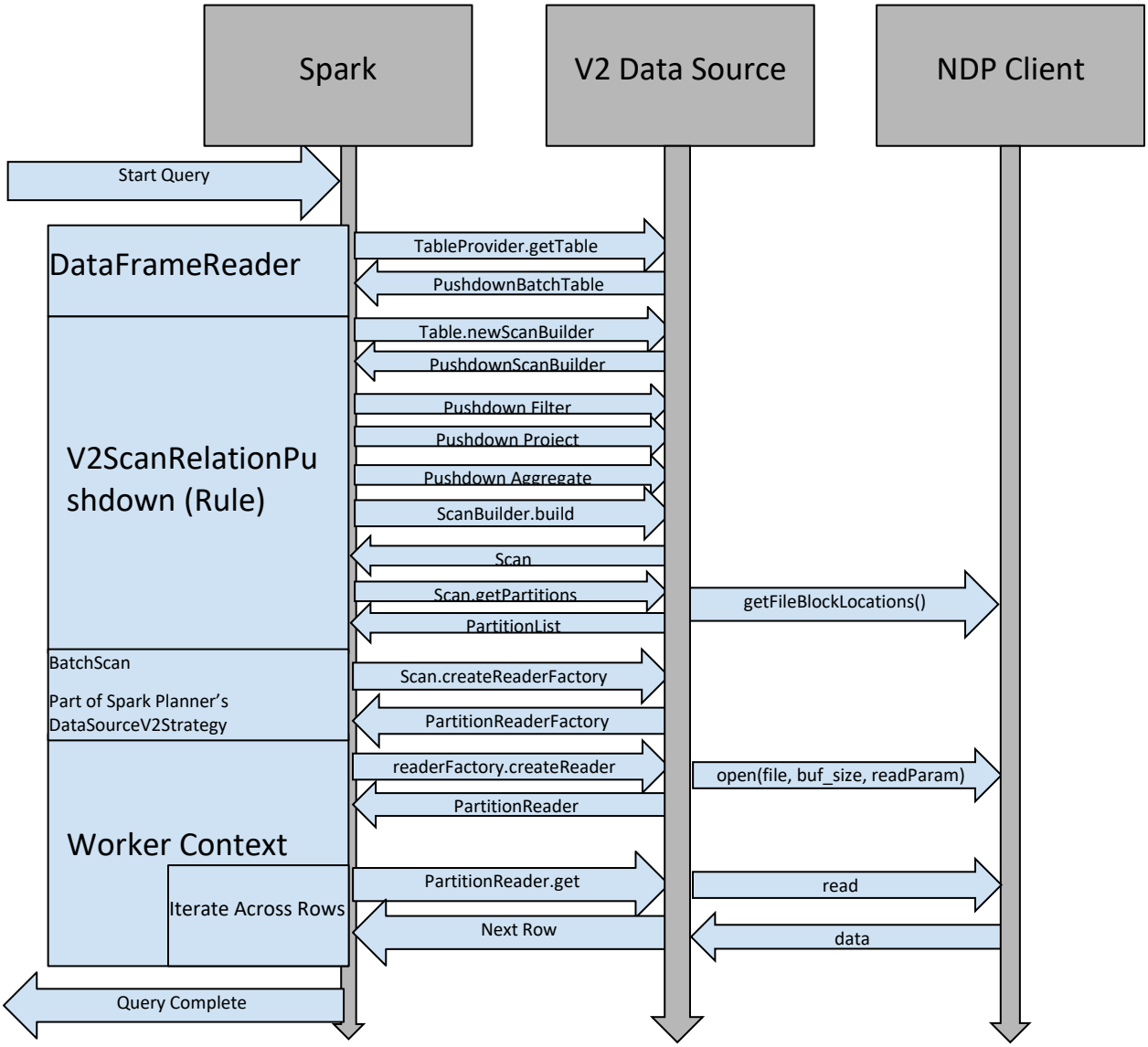

Figure 5: The detailed interactions between a Datasource V2 and Spark for HDFS. Below we will introduce the differences for HDFS.

VII. Datasource V2 for HDFS

The Datasource V2 supports the HDFS API through the NDP client.

Figure 5 above shows some interactions with the NDP client, which we will describe here starting with the fetch of block locations. As part of creating partitions, our HDFS Scan object will fetch the list of blocks from the NDP client. The HDFS Scan object will create one partition per block.

When our HDFS reader object (see above), needs to create an iterator across the rows of the partition, it will call the open method of the NDP client. It is worth mentioning that this open will also provide an additional parameter with pushdowns (if supplied by Spark).

VIII. References

IBM Blog about aggregate push down: https://developer.ibm.com/technologies/analytics/blogs/apache-spark-data-source-v2-aggregate-push-down/

IBM Pull Request for adding aggregate push down to spark: https://github.com/apache/spark/pull/29695

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