NDP – Design Documentation

1. Abstract

NDP - Near Data Processing is a technique where processing of data is pushed closer to the source of the data 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 to perform relational queries on the data.

Our approach will consist of **pushing down** operators 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.

2. Document Layout

We have performed research into a variety of different approaches. This document presents findings from all these approaches.

Sections 3-15 describe our NDP V1 based approach, which includes a Spark V2 Datasource which pushes down an SQL query to our NDP, with NDP using SQLite as the query engine. This approach was completed in 8/2021.

Sections 16-19 describe our NDP V2 based approach, which includes the Rule-based datasource, which sends a query description as a connected graph to our NDP engine.

3. 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, it 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]

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

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”)
```

4. Background

In a typical Spark use case, the user first starts the application, which is called “the driver” in Spark terms. The driver invokes the Spark engine, which processes the incoming query and 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.

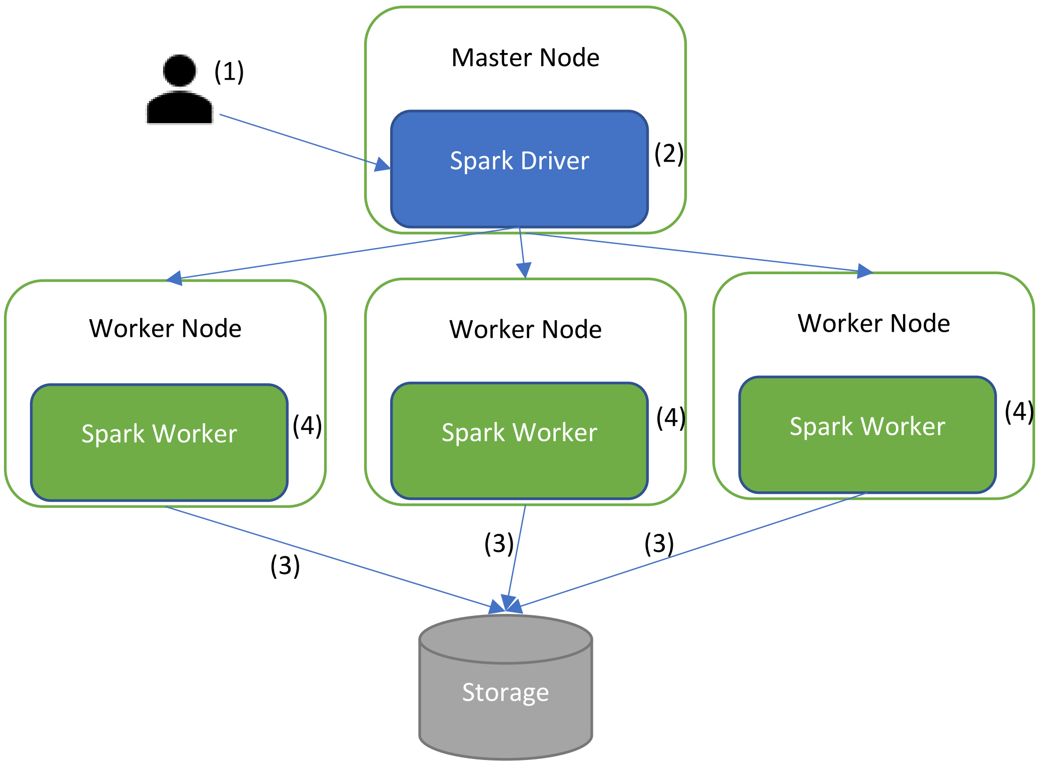


Figure 1. User (1) starts application. Spark (2) distributes jobs across its workers. The workers (3) reads data from storage and (4) perform the operations on the data.

5. 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 SQL operators 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.

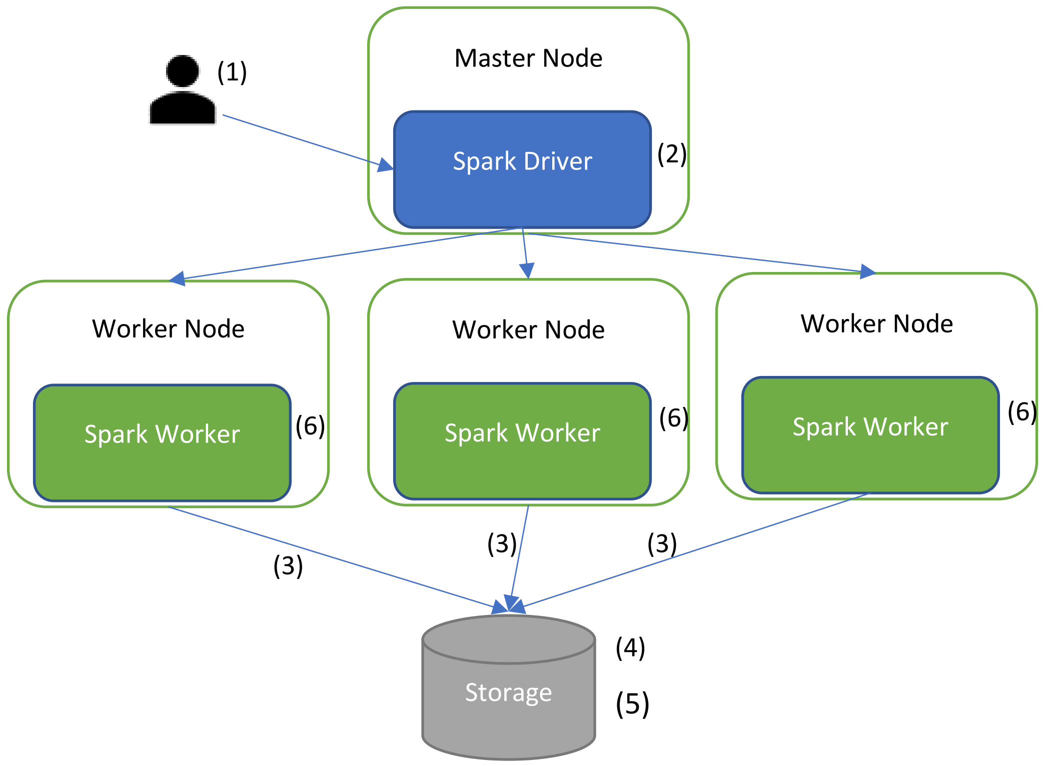


Figure 2. User (1) starts application. Spark (2) distributes jobs across its workers. The workers (3) fetch data from storage and push down operations. The (4) storage reads the data and (5) applies the query. When the worker gets data back and processes the data (6), there is less data to process. This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

6. 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 calls 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 access NDP server by the NDP client API.

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.

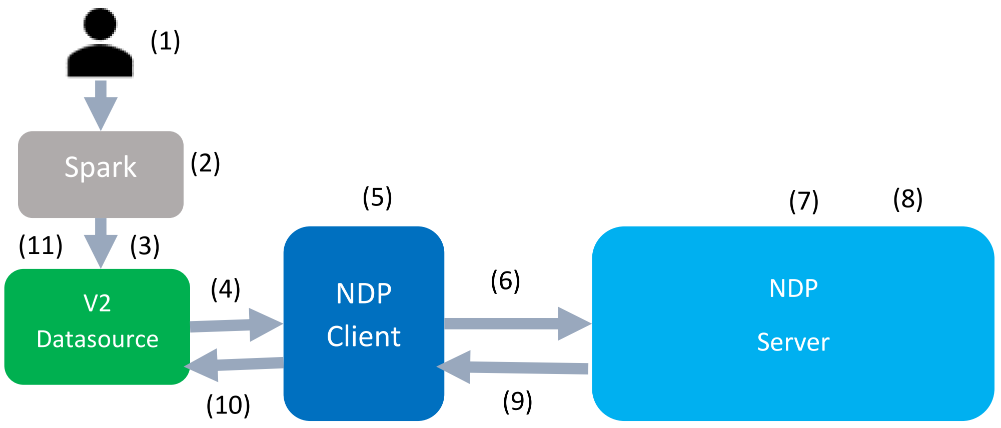


Figure 3. NDP High Level Component Diagram. User (1) invokes application using our new datasource, which uses Spark (2) to execute a query. Spark detects and invokes our V2 datasource (3) in order to perform pushdown, and in order to read the data (4). The NDP Client code (5) provides the Datasource an API 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).

7. 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.

[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

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.

7.1. Filter Operations supporting pushdown

Operation	Description
AND	Logical And
OR	Logical OR
NOT	Logical NOT
=	Equal To
<	Less Than
<=	Less Than or Equal to
>	Greater Than
>=	Greater Than or Equal to
IS NULL	Check if value is equal to NULL
IS NOT NULL	Check if value is equal to NULL
StringStartsWith	Check if the string contains another substring at the beginning
StringEndsWith	Check if the string contains another substring at the end
StringContains	Check if the string contains another substring.

7.2. Aggregate Operations supporting pushdown

These are the currently supported aggregate operations:

- SUM
- Average
- Max
- Min

7.3. 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

[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.

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].

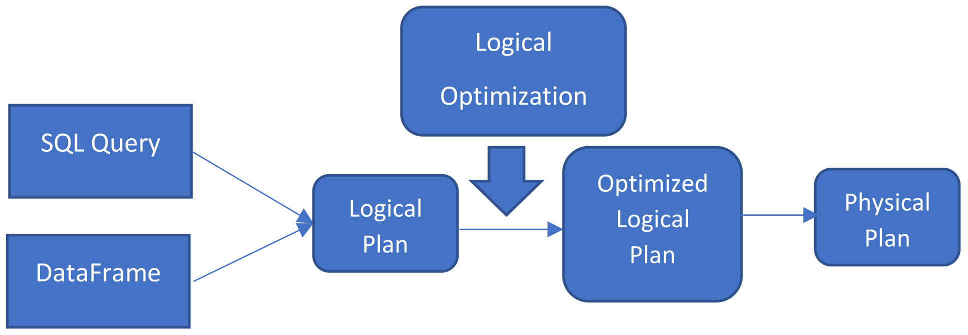


Figure 4. Spark Catalyst Architecture has many steps. The Logical Optimization step is the most interesting for a V2 Datasource since this is the stage where the V2ScanRelationPushDown Rule is applied, which uses the V2 Datasource to transform the logical plan.

7.4. 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.

[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

7.5. 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 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.

7.6. ScanBuilder[9]

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[10]. We call our derived class PushdownScanBuilder.

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

[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.
[9] According to the Spark API, a scan is: "A logical representation of a data source scan. This interface is used to provide logical information, like what the actual read schema is." See definition here.
[10] Just keep in mind that this is also a part of the aggregate pushdown patch.

```
    with SupportsPushDownAggregates
```
[11]

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

```
override def pushFilters(filters: Array[Filter]): Array[Filter]
override def pruneColumns(requiredSchema: StructType): Unit
override def pushAggregation(aggregation: Aggregation): Unit
```
[12]

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)

3) push down the aggregates via pushAggregation(aggregation: Aggregation) [13]

Once the push down is complete, it is expected that the datasource retains state about the filters, projects, aggregates 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
```
[14]

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.

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

7.7. 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.

7.8. 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.

Another override of the Scan object allows returning a PartitionReaderFactory

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

7.9. 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[15] 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.

[15] Batch Scan gets invoked in the context of the Spark QueryPlanner, which converts logical plans into physical plans using execution planning strategies. In our case the execution strategy that is being used to convert our Logical Plan into a Batch Scan is the DataSourceV2Strategy. For more details see: DataSourceV2Strategy.scala

7.10. 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.

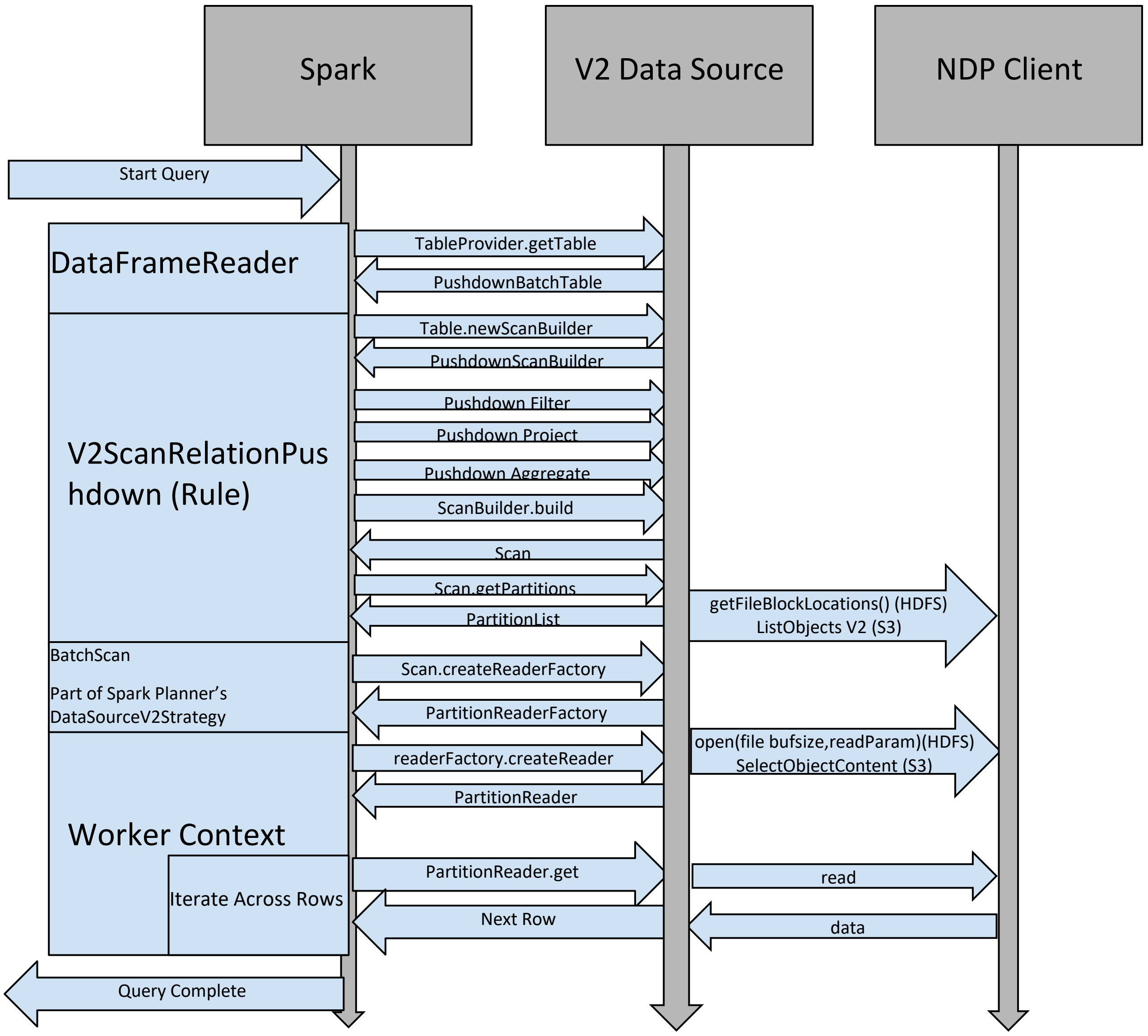


Figure 5. The detailed interactions between a Datasource V2 and Spark for HDFS and S3.

8. Aggregate pushdown

When aggregates are pushed down to both HDFS and S3, there needs to be an aggregation of the data for partitions. In other words, when an aggregate is pushed down to multiple partitions, each partition returns a result, which then needs to be aggregated by Spark in order to return the correct result to the user.

Spark's V2ScanRelationPushdown creates this aggregation. V2ScanRelationPushdown transforms the logical plan in different ways depending on the type of aggregation operation, but in general allows for an aggregation across all the partitions for a given query. For example, in the case of a SUM, the V2ScanRelationPushdown adds to the logical plan, a SUM Spark Catalyst aggregate operation across all the partition results. The result would then be for example, the sum across all values of all partitions. The same pattern is followed for the other aggregate operations, such that the same Spark Catalyst aggregate operation (SUM, MIN, MAX, Average) is performed across all the results returned from all partitions. In the case of an average for instance, the result would be the average across all values of all partitions. For more in-depth detail, see here for the implementation of the various Spark Catalyst expression aggregate operations:

- SUM: https://github.com/apache/spark/tree/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/expressions/aggregate/Sum.scala
- Average: https://github.com/apache/spark/tree/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/expressions/aggregate/Average.scala
- MIN: https://github.com/apache/spark/tree/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/expressions/aggregate/Min.scala
- MAX: https://github.com/apache/spark/tree/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/expressions/aggregate/Max.scala

9. Datasource V2 for HDFS

The Datasource V2 supports the HDFS API through our HDFS 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 **Figure 5** above), needs to create an iterator (which iterates across the rows of the partition) it will call the open method of the NDP client. Under the covers of the iterator, access to the actual HDFS blocks will occur via the InputStream[16], returned by the fs.open() (see below). It is worth mentioning that this open will also provide an additional parameter with pushdowns (if supplied by Spark). The details of InputStream implementation of the HDFS FileSystem are beyond the scope of this document.

[16] Definition of a Java InputStream, according to this link: https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html

"This abstract class is the superclass of all classes representing an input stream of bytes. Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input."

9.1. HDFS API Example

The HDFS open API contains a new parameter, which we have added. The readParam contains XML which describes the query and the schema.

```
val fileStream = fs.open(filePath, bufferSize, readParam)
```

9.2. HDFS API Example

Below is an example of the XML used to describe the read parameter sent to HDFS on the open()

```
<?xml version='1.0' encoding='UTF-8'?>
<Processor>
  <Name>dikeSQL</Name>
  <Version>0.1 </Version>
  <Configuration>
    <Schema>l_orderkey LONG, l_partkey LONG, l_suppkey LONG, l_linenumber LONG, l_quantity
NUMERIC, l_extendedprice NUMERIC, l_discount NUMERIC, l_tax NUMERIC, l_returnflag STRING,
l_linestatus STRING, l_shipdate STRING, l_commitdate STRING, l_receiptdate STRING, l_shipinstruct
STRING, l_shipmode STRING, l_comment STRING
    </Schema>
    <Query><![CDATA[SELECT  SUM("l_extendedprice" * "l_discount") FROM  CaerusObject s WHERE
l_shipdate IS NOT NULL AND s."l_shipdate" >= '1994-01-01' AND s."l_shipdate" < '1995-01-01' AND
s."l_discount" >= 0.05 AND s."l_discount" <= 0.07 AND s."l_quantity" < 24.0 ]]>
    </Query>
    <BlockSize>134217728</BlockSize>
    <FieldDelimiter>44</FieldDelimiter>
    <RowDelimiter>10</RowDelimiter>
    <QuoteDelimiter>34</QuoteDelimiter>
  </Configuration>
</Processor>
```

10. Datasource V2 for S3

The Datasource V2 supports the S3 API using the AWS SDK.

Our datasource fetches a list of partitioned files from the S3 server, using the S3 API ListObjects V2[17] to list the number and size of the files used to represent a specific table. We support file based partitioning and partitioning a single file. We support file based partitioning, where there is a file per partition (for example file.tbl.1, file.tbl.2, etc). In the case of a large file > 128 MB in size, the data source will break this up into partitions 128 MB in size.

When our S3 reader object (see **Figure 5** above), needs to create an iterator across the rows of the partition, it will call SelectObjectContent[18] to fetch the rows using the provided query.

[17] ListObjectsV2 (https://docs.aws.amazon.com/AmazonS3/latest/API/API_ListObjectsV2.html)
Returns list of objects in a bucket.

[18] SelectObjectContent
(https://docs.aws.amazon.com/AmazonS3/latest/API/API_SelectObjectContent.html)
This operation filters the contents of an object based on a simple structured query language (SQL) statement. In the request, along with the SQL expression, you must also specify a data serialization format (JSON, CSV, or Apache Parquet) of the object. Our S3 NDP server uses this format to parse object data into records, and returns only records that match the specified SQL expression.

11. S3 API Example

The S3 API we are using is called SelectObjectContent. Prior to calling that API, we need to create a SelectObjectContentRequest. This is the object which contains the actual SQL query.

```
val request: SelectObjectContentRequest = new SelectObjectContentRequest
request.setBucketName(bucket)
request.setKey(key)
request.setExpression(query)
request.setExpressionType(ExpressionType.SQL)
```

For a query with filter and project the query string might be:

`SELECT name, id, age FROM CaerusObject s WHERE age >= 18`NDP capable web service interface for Object Storage

In this chapter we will describe our solution for Web based Object stores such as AWS S3, Ozone, etc.
In our POC we demonstrated significant advantages of preprocessing data before it is sent over the network. This allows significant reduction in transferred data size, conserves of network bandwidth and takes advantage of data locality on a Storage Server.
We considered at least three (3) potential approaches:

1. Implement some sort of Reverse Proxy (Hadoop term is Application proxy)
2. Intercept data transfer inside Object Store server
3. Implement BackEnd "shim" layer between Object Store Server and actual physical BackEnd

All these approaches have their own advantages and disadvantages.
For our POC we choose to implement Reverse Proxy application and run it on Server Node (Data Node). This approach allowed us flexibility to retrieve and process data internally as well as retrieve data from Storage Server and then process it. Please note, that for benchmark we are retrieving data from local file system to avoid Storage Node related bottlenecks.

In our POC we are focusing on SelectObjectContent API (https://docs.aws.amazon.com/AmazonS3/latest/API/API_SelectObjectContent.html) .
This API is designed to filter data and allows passing of simplified SQL query as an arbitrary string.
Our Datasource V2 is using unmodified Amazon S3 AWS SDK for Java and takes advantage of ability to pass complex SQL queries over S3 SelectObjectContent API.

Our NDP proxy handles two types of S3 requests (SelectObjectContent and ListObjectsV2).
It uses POCO framework (https://pocoproject.org/about.html) to handle HTTP (REST API) related functionality.
Once the REST API request is accepted, NDP proxy configures and creates an instance of The SQLite engine, based on (https://www.sqlite.org/index.html), which in turn loads the SQLite plugin (developed in house).
SQLite plugin retrieves data from local filesystem (or, potentially, from any other source of data) and provides this information back to SQLite engine.

SQLite plugin (Loadable Extension) documentation can be found at (https://sqlite.org/loadext.html)
Sqlite3_module has following interface: (common for ALL Loadable Extensions)

```
struct sqlite3_module {
  int iVersion;
  int (*xCreate)(sqlite3*, void *pAux,
               int argc, char *const*argv,
               sqlite3_vtab **ppVTab,
               char **pzErr);
  int (*xConnect)(sqlite3*, void *pAux,
               int argc, char *const*argv,
               sqlite3_vtab **ppVTab,
               char **pzErr);
  int (*xBestIndex)(sqlite3_vtab *pVTab, sqlite3_index_info*);
  int (*xDisconnect)(sqlite3_vtab *pVTab);
  int (*xDestroy)(sqlite3_vtab *pVTab);
  int (*xOpen)(sqlite3_vtab *pVTab, sqlite3_vtab_cursor **ppCursor);
  int (*xClose)(sqlite3_vtab_cursor*);
  int (*xFilter)(sqlite3_vtab_cursor*, int idxNum, const char *idxStr,
                int argc, sqlite3_value **argv);
  int (*xNext)(sqlite3_vtab_cursor*);
  int (*xEof)(sqlite3_vtab_cursor*);
  int (*xColumn)(sqlite3_vtab_cursor*, sqlite3_context*, int);
  int (*xRowid)(sqlite3_vtab_cursor*, sqlite_int64 *pRowid);
  int (*xUpdate)(sqlite3_vtab *, int, sqlite3_value **, sqlite_int64 *);
  int (*xBegin)(sqlite3_vtab *pVTab);
  int (*xSync)(sqlite3_vtab *pVTab);
  int (*xCommit)(sqlite3_vtab *pVTab);
  int (*xRollback)(sqlite3_vtab *pVTab);
  int (*xFindFunction)(sqlite3_vtab *pVtab, int nArg, const char *zName,
                     void (**pxFunc)(sqlite3_context*,int,sqlite3_value**),
                     void **ppArg);
  int (*Rename)(sqlite3_vtab *pVtab, const char *zNew);
  /* The methods above are in version 1 of the sqlite_module object. Those
  ** below are for version 2 and greater. */
  int (*xSavepoint)(sqlite3_vtab *pVTab, int);
  int (*xRelease)(sqlite3_vtab *pVTab, int);
  int (*xRollbackTo)(sqlite3_vtab *pVTab, int);
  /* The methods above are in versions 1 and 2 of the sqlite_module object.
  ** Those below are for version 3 and greater. */
  int (*xShadowName)(const char*);
};
```

SQLite engine cranks its state machine and reports results back to REST API handler. REST API handler wraps data in AWS S3 data protocol (https://docs.aws.amazon.com/AmazonS3/latest/API/RESTSelectObjectAppendix.html) and sends it back to the client.
On a client side we are using Amazon S3 AWS SDK for Java (https://docs.aws.amazon.com/sdk-for-java/index.html), since it is (de facto) most widely available SDK.
In this document we use term “SQLite engine instance” with following meaning:
SQLite engine instance is a specific realization of SQL query extracted from REST API request. An SQLite engine instance may be viewed as memory blob allocated and initialized to serve a particular SQL query. Each time a REST API request is accepted, an instance of SQLite engine is created by allocating and initializing a blob of memory. This memory is released after REST API request is completed. SQLite engine instance memory blob contains data structures needed for SQLite engine to operate. This memory blob does NOT contain any code or anything else. Just data structures.

At this point it may be helpful to visualize our S3 based approach in **Figure 6** below.

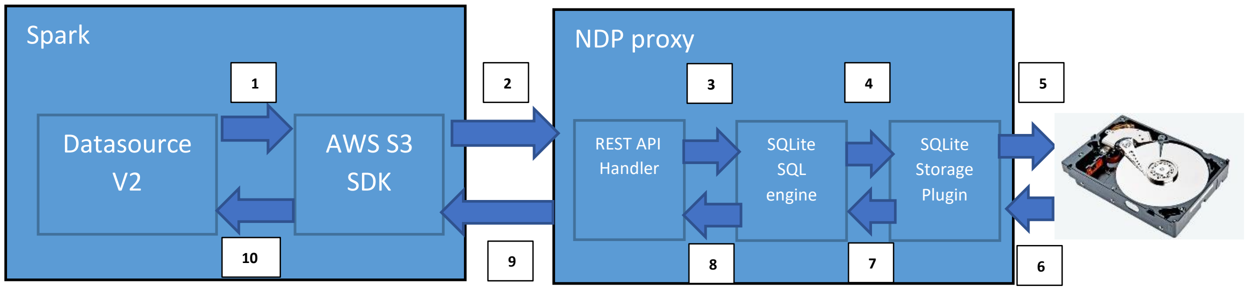


Figure 6. The sequence of interactions in our S3 approach.

1. Datasource V2 issues request to AWS S3 SDK
2. SDK forms REST API request, opens connection and sends this request
3. REST API handler on a Server side parses request and creates an instance of SQLite engine
4. SQLite engine loads preconfigured SQLite Storage Plugin and waits for data
5. SQLite Storage Plugin issues read request to local filesystem
6. SQLite Storage Plugin retrieves data from local filesystem
7. SQLite Storage Plugin convert data to SQLite internal format
8. SQLite reports query results to REST API Handler (SQLite engine instance destroyed)
9. REST API Handler forms AWS S3 response from query results
10. SDK returns query results to Datasource V2

12. NDP capable HDFS

In this chapter we will describe our solution for HDFS based storage.
In our POC we demonstrated significant advantages of preprocessing data before it is sent over the network. This allows significant reduction in transferred data size, conservation of network bandwidth and takes advantage of data locality on a HDFS DataNode.
HDFS uses two internal protocols to transfer data between Client and DataNode:

1. Hadoop RPC (Google proto-buffers based protocol)
2. Hadoop WebHDFS (REST API based protocol)

Please note that HttpFS is just a derivative from WebHDFS which does not require direct connection from Client to DataNode. HttpFS proxy simply aggregates responses from internal DataNodes and sends them back to the Client. In this model Client needs to have access to HttpFS Proxy ONLY.

For our POC we choose WebHDFS due to its similarity with AWS S3 as well as better backward compatibility and more flexible architecture. We strongly believe that this choice will serve us well far into the future.
For our POC we considered at least four (4) potential approaches:

1. Implement some sort of Reverse Proxy (Hadoop term is Application proxy, similar to HttpFS)
2. Intercept data transfer inside DataNode using netty channels
3. Use DataNode plugin (similar to Ozone and R4H (Melanox and RedHat)
4. Implement BackEnd proxy for DataNode (not well documented, but present in code)

All these approaches have their own advantages and disadvantages.
For our POC we choose to Implement Reverse Proxy application and run it on DataNode.
This approach allowed us to retrieve data from DataNode, process it and sent it back to Client.

In our POC we are focusing on WebHDFS open/read API
(https://hadoop.apache.org/docs/r1.0.4/webhdfs.html#OPEN).
This API designed open HDFS file and retrieve data from it.
Please note that this API does not support additional parameters to be passed to DataNode

```
http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=OPEN
                        [&offset=<LONG>][&length=<LONG>][&buffersize=<INT>]
```

For our POC we decided to use HTTP message header (NdpConfig) to pass XML formatted string with entire NDP configuration, such as Processor type (Project, SQL, Sampler, etc.) as well as Processor specific configuration, such as record delimiter, field delimiter, SQL query, etc.
This information may be passed as message body as well, but it is likely to cause problems in the future if we are going to implement approaches 2,3, or 4 (see above)

Our NDP proxy handles "op=OPEN" requests on DataNodes. All other requests forwarded to DataNode.
When NDP proxy detects "op=OPEN" request, it checks for presence of NdpConfig header.

- If NdpConfig header is not present, request is forwarded to DataNode.
- If valid NdpConfig header is detected (here all the fun begins …)
- XML configuration is extracted from HTTP message header.
- InBound connection with DataNode get instantiated (we need to get data from somewhere, right?)

- SQLite instance instantiated and configured to use special SQLite Streaming plugin, capable of retrieving the data from InBound connection with DataNode.
- SQLite engine starts to process the data delivered and formatted by SQLite Streaming Plugin.
- Results returned by SQLite engine streamed back to Client.
- SQLite engine instance destroyed

Explanation of term streaming in context of "SQLite Streaming plugin":
We use term "Streaming" to highlight the difference between S3 plugin and HDFS plugin.
S3 plugin receives data from local file system.
HDFS plugin receives data from HDFS DataNode in form of HTTP stream.
Both plugins implement the same SQLite interface for Loadable Extensions described in page 14.

It is important to mention that our NdpHDFS Client is very similar to WebHdfsFileSystem Client, however there are two modifications we had to make:

1. NdpHDFS require NDP configuration to be provided at FSDataStream initialization time (open)
2. NdpHDFS must be reasonably liberal with respect to "content-length" response header.

From technology standpoint this solution is very similar to AWS S3 based solution.
We are using POCO framework (https://pocoproject.org/about.html) to handle HTTP (REST API) related functionality. Once REST API request is accepted, NDP proxy configures an instance of SQLite engine, based on (https://www.sqlite.org/index.html) as well as SQLite **Streaming** plugin (developed in house).
SQLite Streaming plugin retrieves data from DataNode and provides this information to SQLite engine.
SQLite engine cranks its state machine and reports results back to REST API handler. REST API handler streams it back to the NdpHDFS client.

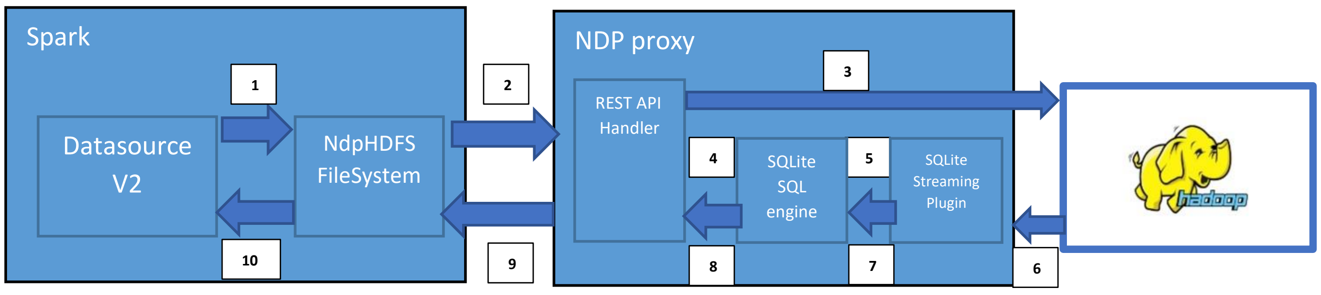


Figure 7. Sequence of interactions in our HDFS approach.

1. Datasource V2 "opens" file using NdpHDFS filesystem
2. NdpHDFS redirects REST API "op=OPEN" to NDP proxy
3. REST API handler parses request and creates InBound stream from DataNode
4. REST API handler creates an instance of SQLite engine
5. REST API handler configures SQLite Streaming Plugin to read the data from InBound stream
6. SQLite Streaming Plugin retrieves data from InBound stream
7. SQLite Streaming Plugin converts data to SQLite internal format
8. SQLite reports query results to REST API Handler (SQLite engine instance destroyed)
9. REST API Handler streams results back to NdpHDFS
10. NdpHDFS returns query results to Datasource V2

13. AWS S3 details

This paragraph is dedicated to various curious wanderers of the technology universe who unequivocally wishes to know every single detail about proposed solution.
AWS S3 is well documented here (https://docs.aws.amazon.com/s3/index.html)
Documentation is in plain English and reasonably accurate. Our contribution is fairly limited in scope.
There are few things worth mentioning:
To support AWS S3 protocols we are using following AWS repositories:
https://github.com/awslabs/aws-c-common.git
https://github.com/awslabs/aws-c-event-stream.git
https://github.com/peterpuhov-github/aws-checksums.git

We are using POCO version derived from:
https://github.com/pocoproject/poco.git

We are using SQLite3 “amalgamation” directly checked in into our branch. Reasoning for this decision can be found at: https://www.sqlite.org/amalgamation.html
There are few links which may help some fearless and dedicated code readers to start looking at the actual code:
`Beginning of request handling:`
https://github.com/futurewei-cloud/caerus-dikeCS/blob/main/SelectObjectContent.cpp#L182

```
void SelectObjectContent::handleRequest(Poco::Net::HTTPServerRequest &req,
Poco::Net::HTTPServerResponse &resp)
```

SQLite initialization:

```
rc = sqlite3_open(":memory:", &db);
```

`SQLite plugin initialization:`

```
rc = sqlite3_csv_init(db, &errmsg, NULL);
rc = sqlite3_tbl_init(db, &errmsg, NULL);
```

`Filename extraction:`

```
string sqlFileName = dataPath + uri.substr(0, uri.find("?"));
```

`main loop of SQLite engine:`

```
while ( (rc = sqlite3_step(sqlRes)) == SQLITE_ROW) {
```

`SQLite result extraction:`

```
int data_count = sqlite3_data_count(sqlRes);
for(int i = 0; i < data_count; i++) {
        const char* text = (const char*)sqlite3_column_text(sqlRes, i);
```

`Working with AWS protocol looks like this:`

```
  int SendEnd(ostream& outStream) {
    struct aws_array_list headers;
    struct aws_allocator *alloc = aws_default_allocator();
    struct aws_event_stream_message msg;
    aws_event_stream_headers_list_init(&headers, alloc);
    aws_event_stream_add_string_header(&headers, MESSAGE_TYPE_HEADER, sizeof(MESSAGE_TYPE_HEADER) - 1,
MESSAGE_TYPE_EVENT, sizeof(MESSAGE_TYPE_EVENT) - 1, 0);
    aws_event_stream_add_string_header(&headers, EVENT_TYPE_HEADER, sizeof(EVENT_TYPE_HEADER) - 1, EVENT_TYPE_END,
sizeof(EVENT_TYPE_END) - 1, 0);
    aws_event_stream_message_init(&msg, alloc, &headers, NULL);
    outStream.write((const char *)aws_event_stream_message_buffer(&msg), aws_event_stream_message_total_length(&msg));
    aws_event_stream_message_clean_up(&msg);
    aws_event_stream_headers_list_cleanup(&headers);
    return 0;
```

```
        }
```

14. HDFS details

In this paragraph we will describe some interesting entry points into our code.

Our NdpHdfsFyleSystem inherits from WebHdfsFileSystem:

```
public class NdpHdfsFileSystem extends WebHdfsFileSystem
```

We implemented new method to create FSDataInputStream:

```
public FSDataInputStream open(final Path fspath, final int bufferSize,
                        final String ndpConfig) throws IOException {
```

This ndpConfig passed all the way down to:

```
abstract class NdpAbstractRunner<T>
and special NdpConfig header is created during connection initialization phase:
private HttpURLConnection connect(final HttpOpParam.Op op, final URL url)
        throws IOException {
    final HttpURLConnection conn =
          (HttpURLConnection)connectionFactory.openConnection(url);
....
    conn.setRequestProperty(EZ_HEADER, "true");
    conn.setRequestProperty("NdpConfig", ndpConfig);
    conn.setDoOutput(doOutput);
    conn.connect();
    return conn;
}
```

On the Server side (NDP proxy on DataNode):

Class to handle DataNode requests:

```
class DataNodeHandler : public HTTPRequestHandler {
```

Similar to AWS S3 handling 😉

```
virtual void handleRequest(Poco::Net::HTTPServerRequest &req, Poco::Net::HTTPServerResponse &resp)
```

Redirection of request to actual DataNode port (and yes it is hardcoded for now)

```
HTTPRequest hdfs_req((HTTPRequest)req);
string host = req.getHost();
host = host.substr(0, host.find(':'));
hdfs_req.setHost(host, 9864);
```

Creation of InBound stream (Step 3)

```
DikeHDFSSession hdfs_session(host, 9864);
hdfs_session.sendRequest(hdfs_req);
HTTPResponse hdfs_resp;
hdfs_session.readResponseHeader(hdfs_resp);
```

Configuring SQLite engine and Streaming plugin:

```
DikeInSession input(&hdfs_session);
DikeOut output(&toClientSocket);
dikeSQL.Run(&dikeSQLParam, &input, &output);
```

In case you interested, requests without NdpConfig handles as following:

```
std::istream& fromHDFS = hdfs_session.receiveResponse(hdfs_resp);
ostream& toClient = resp.send();
Poco::StreamCopier::copyStream(fromHDFS, toClient, 8192);
```

15. Thank you!

Thank you for spending your time (and mine) reviewing this document.
Please keep in mind, that it is by no means a final version.
We are anticipating a lot of comments and suggestions about making this document the best document in a history of humankind, so we certainly will have to make some changes.
It is expected that this document will evolve into something completely different.
On the other hand, this document will become outdated immediately after I will resume development activities.
I think that major parts of the design will change soon, and we will have better overall performance.
So, please, do not take this scripture very seriously!!!

Good Luck 😊

16. Rule-based Datasource

Our initial NDP approach detailed in sections 5-15 above, was one where we used a V2 datasource, utilizing the Spark V2 pushdown APIs, which allow for pushing SQL operators such as project, filter and aggregate down to NDP. This following section details the subsequent implementation of NDP using a rule-based datasource. The rule plug-in is a native Spark mechanism for changing the plan, which provides a standard API for a plan modification rule.

The NDP datasource is what we refer to as "Rule-Based". This means there is a custom rule for NDP, which is used to insert the NDP datasource. This rule's code is a part of the datasource code base, and the rule itself is configured by the user in the application setup of the Spark Session. Existing Spark applications need a single change to enable NDP pushdown. This has a huge advantage in that the user will get the advantage of NDP pushdown with no disruption to their code.

The rule-based NDP Spark datasource allows for all needed pushdowns to be sent to NDP such as projects, filters, and aggregates. The rule-based NDP datasource allows for additional pushdown possibilities compared to the standard Spark datasource, which is restricted to the pushdowns allowed by Spark's v2 datasource rule called V2ScanRelationPushdown. For example, Spark today does not allow a filter pushdown where the filter expression has a column on the right side. An expression like (col1 > col2) is one example that Spark does not allow, but which our rule-based datasource does allow.

Our NDP datasource is designed to be invoked only when needed, meaning that it gets inserted automatically by our custom rule only when there will be a benefit. A standard Spark datasource is always inserted for use by default, and the disadvantage of this approach for an NDP datasource is that it is not always needed. We found that the Rule based approach for this case is much simpler and natural since in the case where NDP is not needed, our Rule does not change the logical plan and therefore, Spark simply handles this case as usual.

The rule is aware of storage capabilities and any operations that are not supported by storage will be resolved by Spark itself. For example, if a filter contains an unsupported operator like a UDF (User Defined Function), then the rule will not attempt to push down the filter and will make sure that the plan still contains the filter so that Spark will continue to evaluate it. All the above makes the solution seamless from the user's perspective. The below Figure introduces the high-level component diagram, which we will describe in detail below.

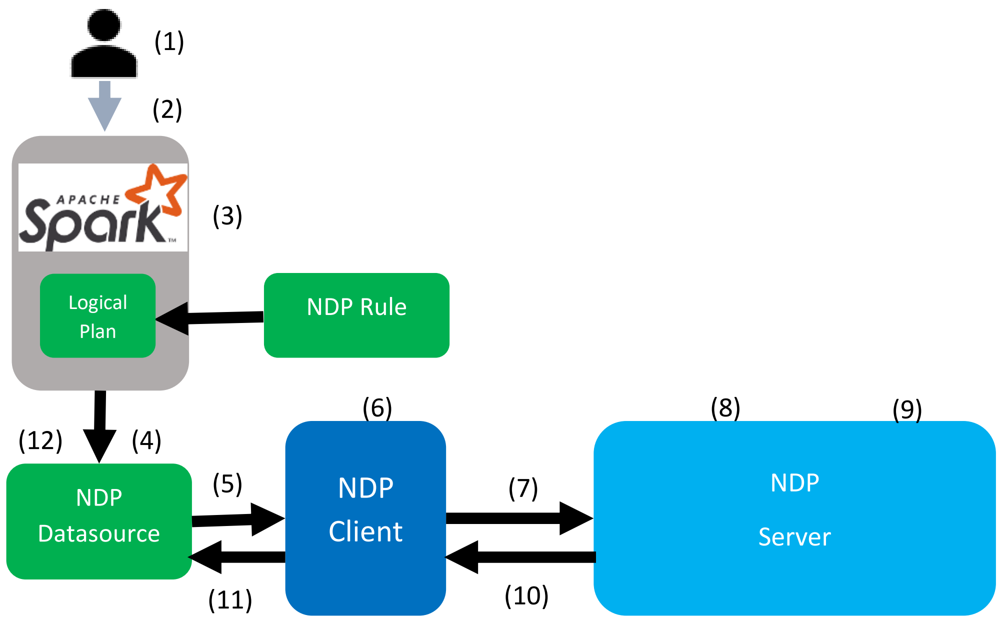


Figure 8. NDP High-Level Component Diagram. User (1) invokes application, which uses Spark (2) to execute a query. Note that the user also directed Spark to insert our NDP Logical Optimization Rule. This rule (3) detects a pushdown opportunity and rewrites the logical plan to include the NDP datasource (4) to perform pushdown, and to read the data returned by NDP (5). The NDP Client code (6) provides the datasource an API to (7) read with pushdown from the NDP Server. The NDP server performs a read of the data (8), applying the necessary operators of project, filter, and aggregate (9) before streaming the data back to the client (10). The NDP datasource receives the data back from NDP (11) and forms the appropriate batches of rows before returning the data to spark at (12).

16.1. Rule Based Datasource Details

16.1.1 Inserting a Spark datasource

A regular Spark datasource is inserted by the user every time they wish to access a datasource. The “format” method on the Spark Session’s DataFrameReader API specifies the datasource to use. In the below case “parquet” specifies Spark’s internal parquet datasource.

```
sparkSession.read
              .format("parquet")
              .load(filename)
```

When a standard data source is used, the user needs to change all instances in their code that specify a data source. This can be substantial change depending on the application.

However, the Rule-based datasource does not have this restriction. With a Rule-based datasource the user only needs to add our rule (PushdownOptimizationRule) to the list of rules specified in the configuration for the SparkSession.

```
sparkSession.experimental.extraOptimizations ++= Seq(PushdownOptimizationRule)
```

With just this one change, the NDP rule will be evaluated for all queries that the user forms. In other words, regardless of which data source the user chooses, our rule will still have an opportunity to inspect the logical plan and insert our NDP datasource.

16.1.2 NDP Custom Rule Flow

The custom rule operates by first parsing the entire logical plan. The rule then validates the plan, checking to make sure that our rule can parse the plan. Next, the rule will check to see if NDP will have benefits for this plan. For example, we check to see that a project is being performed. If no project is being done, we choose not to intervene, with the reasoning being that if no project is done there is little opportunity for optimization, with the entire table being returned.

Our rule will detect which pushdowns are possible such as projects, filters, and (coming soon) aggregates. This rule constructs a new Spark logical plan which automatically inserts the NDP datasource, with all needed pushdowns. The new plan will exclude the operations which NDP will execute such as filter and/or project. These operations will not need to be performed by Spark since NDP is fulfilling these operators. The rule also will change the logical plan to expect the format of data being returned from NDP. For example, if any columns are only needed by a filter, and that filter is pushed to NDP, then those columns can and should be excluded from the data that NDP returns. Below is an example of this type of change in the logical plan.

16.1.3 Logical Plan Modification Example

The original logical plan is below and shows a case of a project with a filter. Note also that the datasource in this case is called “parquet”. This denotes the Spark built in parquet datasource. The relation in this case is returning 7 columns, 1 of which is used by the filter (l_shipdate), and 6 of which are needed by the project.

Original Logical Plan

```
+- Project [l_quantity#20, l_extendedprice#21, l_discount#22, l_tax#23,
l_returnflag#24, l_linestatus#25]
 +- Filter (isnotnull(l_shipdate#26) AND (l_shipdate#26 <= 1998-09-02))
  +- RelationV2[l_quantity#20, l_extendedprice#21, l_discount#22,
l_tax#23, l_returnflag#24, l_linestatus#25, l_shipdate#26] parquet
hdfs://dikehdfs:9000/tpch-test-parquet/lineitem.parquet
```

Also below is the modified logical plan.

Our Rule makes the following modifications to the logical plan.

- The new plan is modified to redirect to the NDP data source. Note that the datasource **`com.github.datasource.PushdownOpBatchTable`** is specified.
- The new plan removes the Filter, since NDP will be providing the filter
- The new plan is modified so the columns returned match only what is needed by the project. In the below case you will note that l_shipdate is no longer needed by the filter, since the filter is performed on the NDP server. Therefore, this is reflected in the data retrieved by the data source as represented by the relation.

Modified Logical Plan

```
+- RelationV2[l_quantity#20, l_extendedprice#21, l_discount#22, l_tax#23,
l_returnflag#24, l_linestatus#25] class
com.github.datasource.PushdownOpBatchTable
```

16.1.4 Plan Execution

The optimized logical plan will include use of our rule-based datasource. This datasource receives the pushdowns (project, filter, and aggregate) from the rule. These pushdowns are formed into a JSON representation of this request for sending to NDP.

16.1.5 JSON Request

The JSON request consists of a few fields which represent the pushdowns we are sending to NDP.

16.1.5.1 Project

The general format of a project is as a list of columns.

```
{
  "Type": "_PROJECTION",
  "ProjectionArray": [ “l_quantity”, “l_extendedprice”, “l_discount”, “l_tax”, “l_returnflag”,
“l_linestatus” ],
   "Name": "TPC-H Test Q01"
}
```

16.1.5.2 Filter

The general format of a filter is the below. The FilterArray is a set of filters to be applied to the data.

```
{
   "FilterArray": [...expressions here...],
   "Name": "Filters"
}
```

There are many expressions that are supported in the FilterArray.

16.1.5.2.1 Three argument expressions

This first set of expressions has three arguments (Expression, Left, Right)
And, Or, EqualTo, LessThan, GreaterThan, LessThanOrEqual, GreaterThanOrEqual, StartsWith, EndsWith, Contains

Example:

```
{
  "Left": {"ColumnReference": "l_shipdate"},
  "Expression": "LessThanOrEqual",
  "Right": {"Literal": "1998-09-02"}
}
```

16.1.5.2.2 Two argument Expressions

This set of expressions has two arguments (Expression, Arg)
Not, IsNull, IsNotNull
Example:

```
{
   "Expression": "IsNotNull",
   "Arg": {"ColumnReference": "l_shipdate"}
}
```

16.1.5.2.3 Other Expressions

We also have ColumnReference, Literal, which take one argument apiece.

```
{"Literal": "1998-09-02"}
{"ColumnReference": "c_custkey"}
```

16.1.5.2.4 Filter Full Example

This is an example of the full filter JSON.

```
{
  "Type": "_FILTER",
  "FilterArray": [
    {
      "Expression": "IsNotNull",
      "Arg": {"ColumnReference": "l_shipdate"}
    },
    {
      "Left": {"ColumnReference": "l_shipdate"},
      "Expression": "LessThanOrEqual",
      "Right": {"Literal": "1998-09-02"}
    }
  ],
  "Name": "TPC-H Test Q01"
}
```

16.1.5.2.5 Nesting

It is worth mentioning that our format allows for nesting of expressions. So, for example, we could nest arbitrary And and Or expressions to arbitrary depths with this format. Although our format supports this arbitrary depth of nesting, our NDP server does have limitations and only supports nesting to a depth of one level.

16.1.5.2.6 *XML with JSON*

The full request that is sent to NDP is an XML format with embedded JSON. The JSON describing the operations to be pushed down is labeled DAG, and can be found inside the Configuration node of the XML.

The DAG has several sections, with each section identified by a "Type" field.

The supported types are:

_INPUT – has a field "File", with the input filename.

_PROJECT – Has a "ProjectArray" field, with the List of columns included in the project.

_FILTER – Has a "FilterArray" field, with the list of filters to be evaluated.

_OUTPUT – Has a "CompressionType" field, describing how the output is to be delivered. Valid values here are None, ZSTD, or LZ4. Note with ZSTD, an additional "CompressionLevel" field, detailing the degree to which to compress. Note that once compression is enabled, by selecting ZSTD or LZ4, the degree to which we compress (if at all)

16.1.5.2.7 *Full Example*

The below example shows the XML that is sent to NDP, along with the embedded JSON. Sections 9.1 and 9.2 above have more details from our V1 NDP.

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

<Processor>

  <Name>Lambda</Name>

  <Configuration>

    <DAG>

    {

    "Name":"DAG Projection",

    "NodeArray":[

        {"Name":"InputNode",

         "Type":"_INPUT",

         "File":"/tpch-test-parquet/lineitem.parquet/part-00000-c498f3b7-c87f-4113-8e2f-
0e5e0c99ccd5-c000.snappy.parquet"},

        {"Type":"_FILTER",

         "FilterArray":[{"Expression":"IsNotNull",

                         "Arg":{"ColumnReference":"l_shipdate"}},

                        {"Left":{"ColumnReference":"l_shipdate"},

                         "Expression":"GreaterThanOrEqual",

                         "Right":{"Literal":"1995-09-01"}},

                        {"Left":{"ColumnReference":"l_shipdate"},

                         "Expression":"LessThan",

                         "Right":{"Literal":"1995-10-01"}},

                        {"Expression":"IsNotNull","Arg":{"ColumnReference":"l_partkey"}}],

         "Name":"TPC-H Test Q14"},

        {"Name":"TPC-H Test Q14",

         "Type":"_PROJECTION",
         "ProjectionArray":["l_partkey","l_extendedprice","l_discount"]},
        {"Name":"OutputNode",
         "Type":"_OUTPUT",
         "CompressionType":"ZSTD","CompressionLevel":"2"}]
     }
     </DAG>
    <RowGroupIndex>3</RowGroupIndex>
    <LastAccessTime>1629996964793</LastAccessTime>
  </Configuration>
</Processor>
```

16.1.6 Logical Optimization Rule

In Figure 2 below we see the process that a Spark Query goes through from query to Physical plan. The NDP rule is a "logical optimization rule", meaning that it gets evaluated by Spark during the logical optimization phase. Below is a diagram of the Spark process for processing a query and for transforming the query.

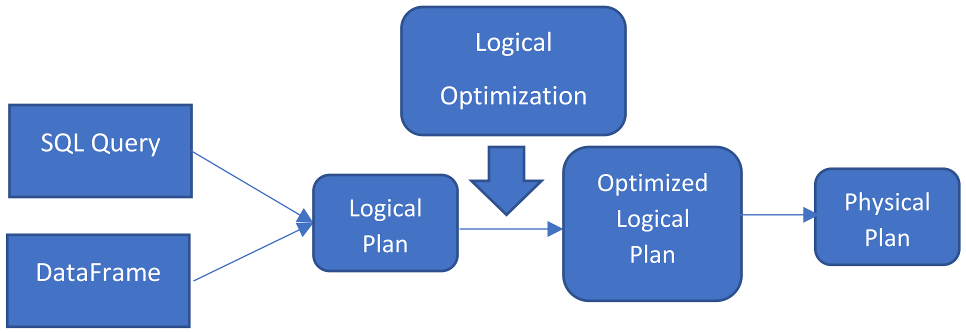


Figure 9. Spark evaluates our NDP Rule during the Logical Optimization Phase.

16.1.7 Detailed Interactions

In the sections above, we detailed the interactions involved for our original v2 data source. The interactions for the new Rule-based datasource are similar since the new Rule-based datasource is also a v2 data source. One of the big differences is that our new Rule-based datasource does not utilize the Spark pushdown APIs. These APIs are not needed since the Rule itself initializes the data source with all needed pushdowns.

Our original v2 data source returned the data row by row. However, the new rule-based datasource utilizes a ColumnarBatch based Spark API. You can think of the ColumnarBatch as a container for a set of rows. This allows a datasource to read in a set of rows in a batch, and then expose this batch to Spark. Spark understands this API, and thus can iterate across the batches as well as the rows within each batch.

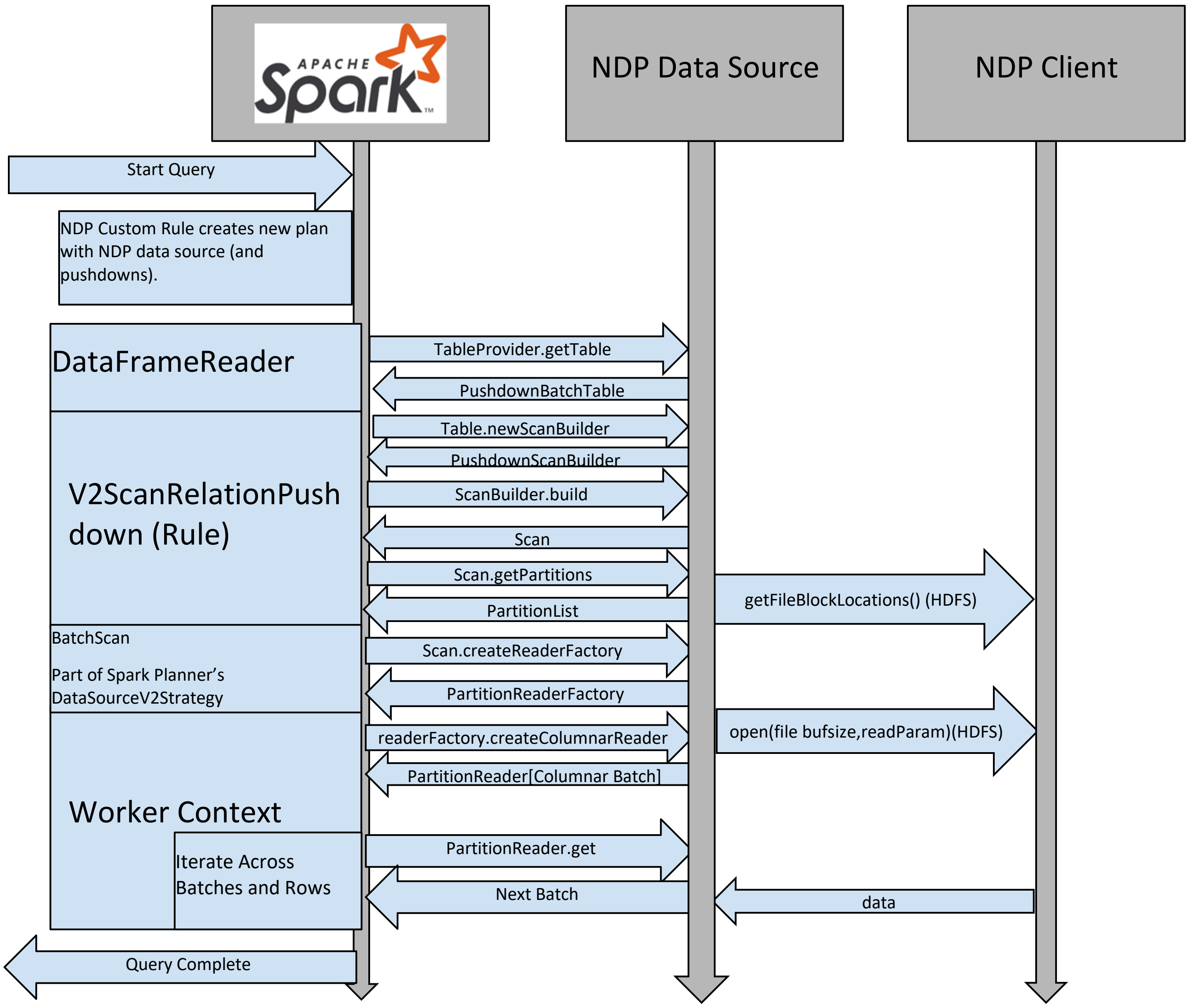


Figure 10. The detailed interactions for the NDP Datasource.

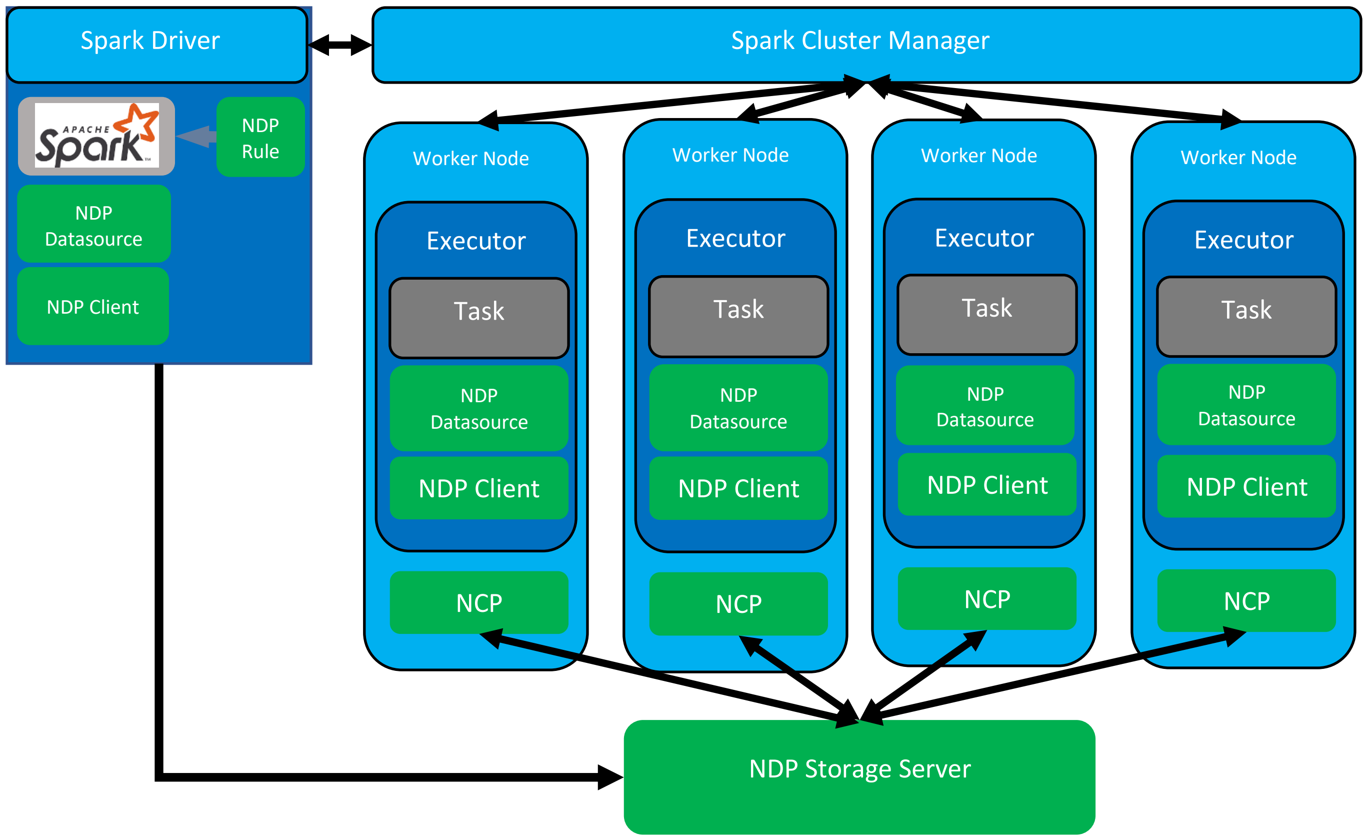


Figure 11. Spark Cluster with NDP

16.2. Cluster Description

Above is the diagram of a Spark cluster. The Spark cluster is composed of a few independent processes which can run on the same or different machines. In our diagram below it is intended that these processes run on separate machines for the most part. However, in our case the driver and the master are running on the same physical machine. For more details on the definitions from a Spark perspective see the references below.[19]

16.2.1 Spark Definitions

Driver - The driver program is the main application that is started by the user. The driver converts the user's program into tasks, and then schedules the tasks across the executors.

Workers (slaves) – Workers will communicate availability of resources to the cluster manager.

Executors – These are separate processes (JVM) run in a worker node. Are responsible for running tasks.

[19] See https://spark.apache.org/docs/latest/cluster-overview.html and https://stackoverflow.com/questions/32621990/what-are-workers-executors-cores-in-spark-standalone-cluster

Cluster Manager – The cluster manager receives a request from the driver for resources and launches the executors.

In our test setup, we configure the number of cores per executor to a fixed amount (e.g., 8). This will then allow spark to configure up to this fixed number of tasks to run in each executor with a single core each.

16.2.2 NDP Specific Components

The diagram also shows NDP specific components. The purpose of this section is to introduce all the components, not necessarily provide all the detail for those. For more detail on the Spark definitions see above.

Driver

Our NDP Rule is evaluated to determine when to insert our datasource. If our datasource is inserted, it will communicate with the NDP client to access NDP.

Executor

In the context of the executor, when tasks are executed, they will use our NDP data source, which in turn will communicate with NCP via the NDP client.

NCP

In the context of the worker node, the Near Compute Processor will receive requests for data and will decide where to execute the operation, either locally or remotely via NDP Storage Server. The NCP is needed because we want to provide a consistent API for the NDP client and the datasource itself. As load balancing is performed to balance requests between compute and the remote NDP server, this consistent API allows for either executing the requests locally (in NCP) or remote (in NDP).

17. NDP V2 Rationale

Our initial focus was on CSV format, however Spark by default uses Apache Parquet format.
Apache Parquet is columnar format with binary data types, embedded schema and compression enabled by default. Spark uses this format extremely efficiently. Columnar formats allow to read relevant columns only, in other words projection is supported by format itself and NDP can't provide any added value. In fact, NDP may cause performance degradation if projection is the only operation to perform. To confirm this statement, we implemented SQLite plugin for Parquet format. Our performance tests (TPC-H 100GB) showed significant performance degradation (over 3x) compared to Spark even when filtering is enabled.
We started investigation of the NDP performance and compiled a list of issues (lessons learned) which needed to be addressed.

18. NDP V2 Lessons Learned

18.1. Output Format

While working with CSV format we were able to send output data to the Spark immediately after it became available. We did not have to wait until the end of partition. This allowed Spark to start processing data almost immediately, creating an efficient pipeline.
However, Parquet format defines the metadata block at the end of the file ("footer") and this requires NDP to collect all the data before it can generate metadata and send output to the client. This design prevents any parallelism between Spark and NDP.
To solve this problem, we defined a new output format similar to Parquet, but with a different metadata location.
Streamable Columnar Format (SCF) divides data into an arbitrary number of blocks (similar to Parquet's RowGroups).
Each block is self-describing and metadata is located at the beginning of the block. SCF output allows NDP to send data back to client as soon as it is available. SCF supports different compression algorithms. Currently SCF uses Zstandard compression as a default algorithm, but LZ4 is also supported. SCF can dynamically adjust compression level or disable compression if Network / CPU balance is appropriate.

18.2. SQLite performance

Parquet format is more efficient than CSV. Parquet support exposed SQLite as a new bottleneck in the system.
SQLite works by translating SQL statements into bytecode and then running that bytecode in a virtual machine. This approach is inherently slower than execution of native binary code. To solve this problem, we implemented Filter and Projection functionality in C++. Filter Node implemented as configurable C++ object with a set of templated comparison methods optimized for handling columnar data. Projection Node is also C++ object which can rearrange columns in desired order. This approach has optimal performance for a given hardware. It also allows future optimizations with GPU utilization.

18.3. Network bottleneck

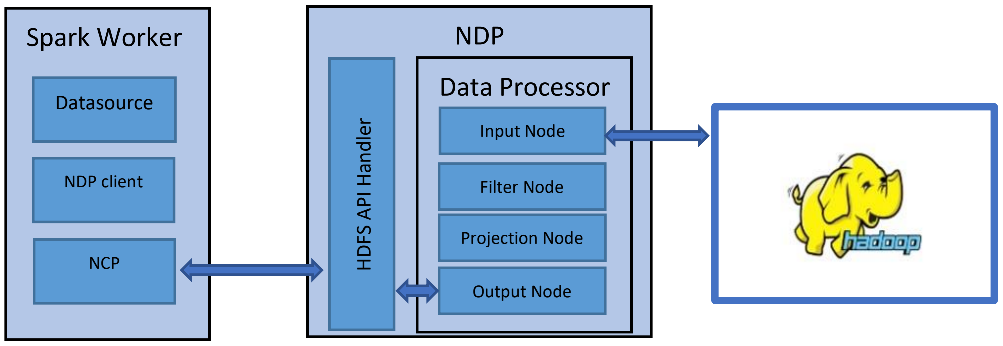
During our TPC-H testing we used 1G network between Compute node and Storage node. We discovered that Spark worker can ingest and process the data at ~60MB/s per core. It means that 2 cores per worker can saturate 1G network. We estimated that 10G network will be saturated if 16-20 cores used by Spark worker.
There are several ways to reduce network traffic. We focused on filtering, compression, and aggregation (work in progress). Filtering is reasonably cheap operation, however there is no guaranteed data reduction. For example, filter for TPC-H Q1 test reduces data by 4% only. Using compression much higher reduction can be achieved (2x – 5x), but it is very CPU intense operation. Right now, we are working on aggregation which has the best reduction, and we hope to make it less CPU intense than compression.

18.4. NDP bottleneck and DNDP

During our TPC-H testing we simulated NDP bottleneck. Assuming we have Storage appliance or Storage cluster with fixed amount of CPU power. If Spark compute cluster is significantly more powerful than Storage, NDP will not be able to process the data at required speed and will become a bottleneck. We simulated this situation by lowering CPU power for NDP docker. To solve this problem, we need to be able to Dynamically decide whether to process the data by NDP or not. This can be accomplished by using NCP (Near Compute Processing) running alongside with Spark worker. By default, NCP is a simple passthrough entity. It simply redirects all requests to NDP. If NDP has enough CPU capacity, it will process the data itself. However, if NDP is low on CPU resources it will instruct NCP to process the data. There is no network traffic reduction for the requests with NCP engaged, however there are reductions with NDP requests. For example, assuming NDP can process up to 16 requests simultaneously. In this case at queue depth less than 16 all requests will be processed by NDP. And at queue depth larger than 16 some requests will be redirected for NCP processing. We call this approach "Dynamic NDP" DNDP in short.

19. NDP V2 Design

NDP V2 design is similar to the original NDP design with some components added and some replaced.



Datasource generates description of pushdown operation in Json format. At the heart of this description is Node Array. This Node Array describes Processing Nodes and their configurations.
Then request is sent to NDP client, which in turn forwards it to the NCP.
NCP simply forwards request to NDP.
NDP evaluates current state of Storage Node (CPU utilization, Queue Depth, Rebuild State, etc.)
If NDP decides to handle the request, it will instantiate Data Processor and configure Node Array based on Json pushdown description:
Input Node is responsible for reading data from HDFS.
Filter Node is responsible for reducing data based on filtering configuration.
Projection Node is responsible for arranging the columns in proper order for the output.
Output Node is responsible for sending data in SCF format and making dynamic decision about need for compression.
Data generated by Output Node is sent back to NDP client via HDFS API handler.
If NDP decides not to handle request it simply redirects it to NCP and NCP performs the same operations while running on Compute Node.

Please note that there are two different ways to implement NCP:
NCP can be implemented in Java as part of NDP client. This approach has advantages of potentially better integration. However, it also has some disadvantages such as JMV performance. Complex interoperability management, etc.

Another way to deploy NCP is to use NDP docker configured in a special way. Our NDP can be configured to act as an NCP. This approach has advantage of very high percentage of code sharing, good performance and ease of development.

Currently we use NDP docker with special configuration to act as NCP.

20. 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