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 order 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, 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]

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 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.

Commented [HL1]: Does "driver" refer to the Spark application written by the user, or the part of the Spark engine that processes incoming requests through the Spark API?

Commented [RF2R1]: Yes, the driver is the application. I added some details to help elucidate further.

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.

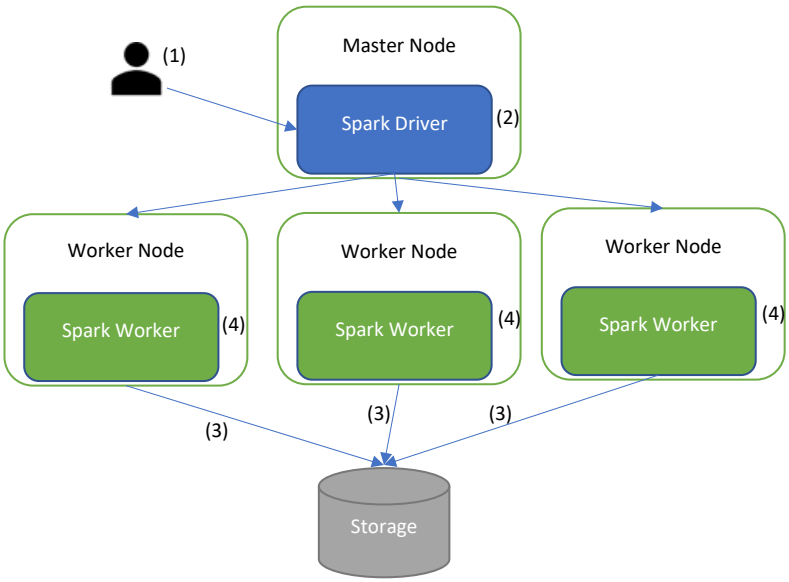


Figure 1*:* User (1) starts application. Spark (2) distributes jobs across its workers. The workers (3) reads data from storage and (4) perform the operations on the data.

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 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.

Commented [HL3]: Change to "SQL operators"?

Commented [RF4R3]: OK, all set.

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.

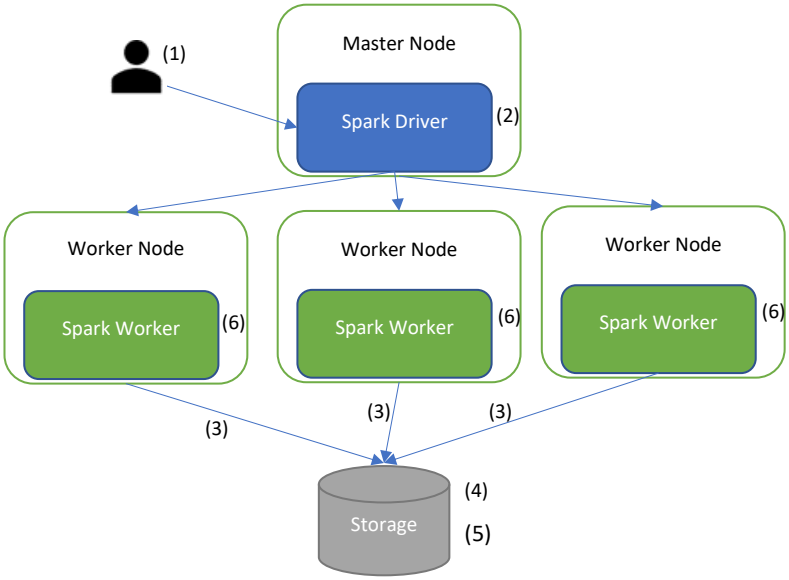


Figure 2*:* User (1) starts application. Spark (2) distributes jobs across its workers. The workers (3) fetch data from storage and push down operations. The (4) storage reads the data and (5) applies the query. When the worker gets data back and processes the data (6), there is less data to process.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

Commented [HL5]: You switched to a different font size. ☺

Commented [RF6R5]: Good catch ! 😊 Fixed it throughout.

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.

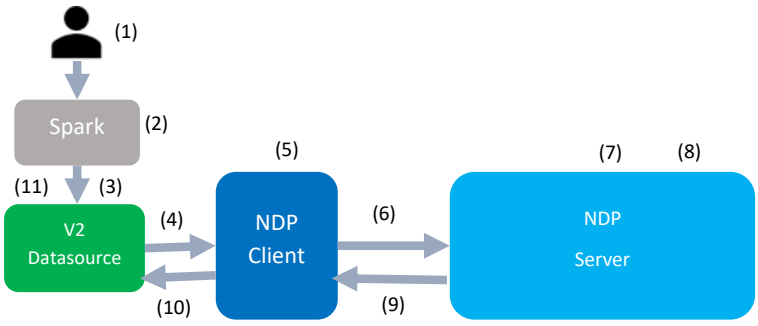


Figure 3: NDP High Level Component Diagram. User (1) invokes application using our new datasource, which uses Spark (2) to execute a query. Spark detects and invokes our V2 datasource (3) in order to perform pushdown, and in order to read the data (4). The NDP Client code (5) provides the Datasource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

B. Aggregate Operations supporting pushdown

These are the currently supported aggregate operations:

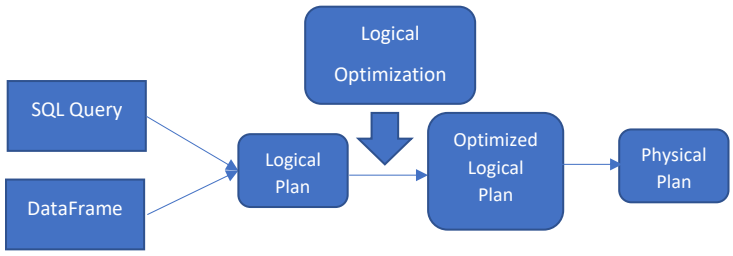
- SUM

[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

- Average
- Max
- Min

C. 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].



[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.

D. 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.

E. 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] 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.

F. 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
    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

Commented [HL7]: Can you add a sentence here on what "scan" in Spark means? Is this correct: Scan in Spark is an abstraction for logical information (e.g., data schema) required to reading from a data source.

Commented [RF8R7]: Definition of scan added as footnote (9).

[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.
[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.

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.

G. 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.

H. 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
```

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

I. 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.

Commented [HL9]: Who calls BatchScan?

Commented [RF10R9]: Added another footnote (15) with some extra details on BatchScan.

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.

J. 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.

[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

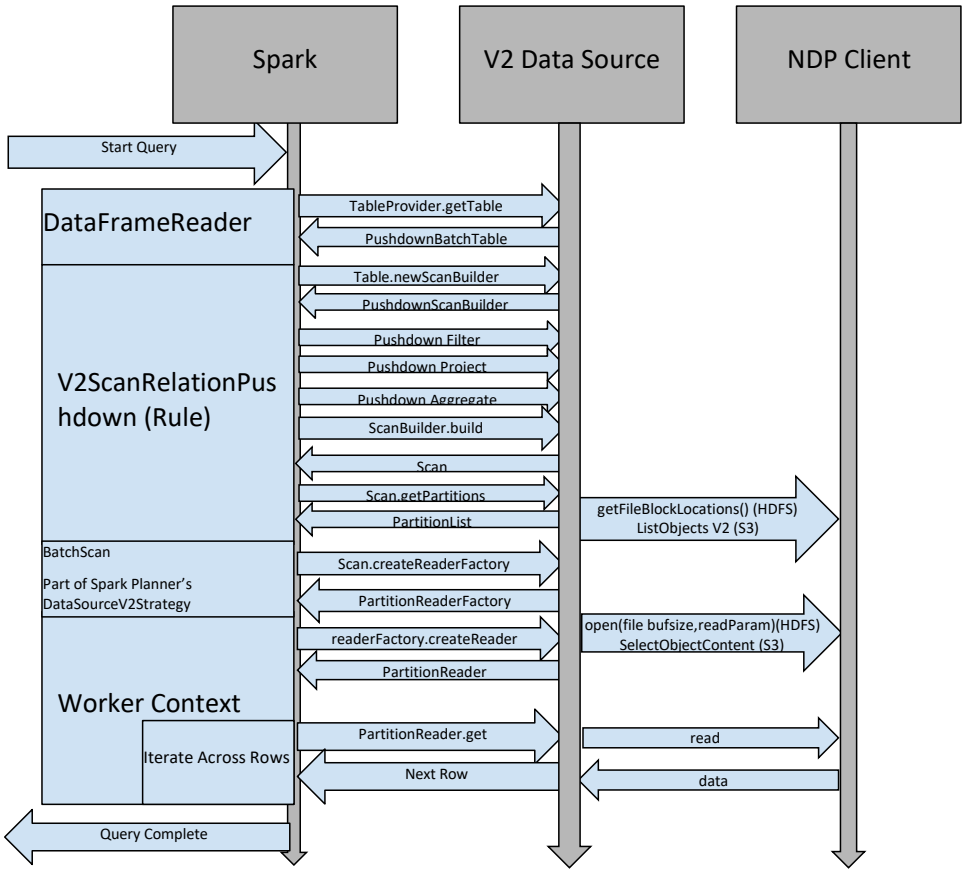


Figure 5: The detailed interactions between a Datasource V2 and Spark for HDFS and S3.

VII. 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

VIII. 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.

Commented [HL11]: For pushdown of aggregates to both HDFS and S3, can we describe where and how the aggregate data for partitions are further aggregated at the data frame level?

Commented [RF12R11]: Section added above with details.

Commented [HL13]: Are we reading a row at a time or a block at a time?

Commented [RF14R13]: The iterator iterates across rows. I have added a few more details here.

[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."

A. 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)
```

B. 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>
```

Commented [HL15]: I suppose the name "S3Object" is insignificant here? If so, can we avoid S3 because this is all about HDFS? Can we change it to CaerusObject?

Commented [RF16R15]: I will defer to Peter on this. Peter, can you please answer? And please update here and in section X if appropriate? Thanks!

Commented [PP17R15]: Done.

IX. 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.

X. 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)
```

Commented [HL18]: Can you give an example of the query string with filter and project operations?

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

```
SELECT name, id, age FROM CaerusObject s WHERE age >= 18
```

XI. 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 (*xRename)(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.

Commented [HL19]: Do all SQLite plugins expose the same API to the engine?

Commented [PP20R19]: Added SQLite plugin API description

Formatted: Font: 10.5 pt

At this point it may be helpful to visualize our S3 based approach in **Figure 1** below.

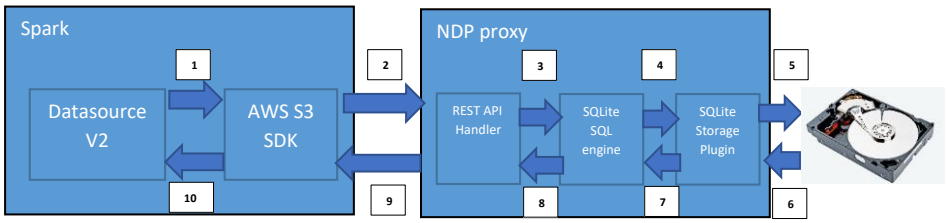


Figure 1: The sequence of interactions in our S3 approach.

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

Commented [HL21]: Is this a process instance or container instance?

Commented [PP22R21]: Added section about "SQLite engine instance" Above

Commented [HL23]: Does the SQLite Engine instance need to be destroyed at this point?

Commented [PP24R23]: Added section about "SQLite engine instance" Above

XII. 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.

Commented [HL25]: Is it more accurate to say HttpFS Proxy here?

Commented [PP26R25]: Fixed

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

Commented [HL27]: Again do we explicitly destruct the engine instance here?

Commented [PP28R27]: Fixed

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.

Commented [HL29]: Please explain why the plugin for HDFS is streaming but the plugin for S3 is not.

Commented [PP30R29]: Added explanation

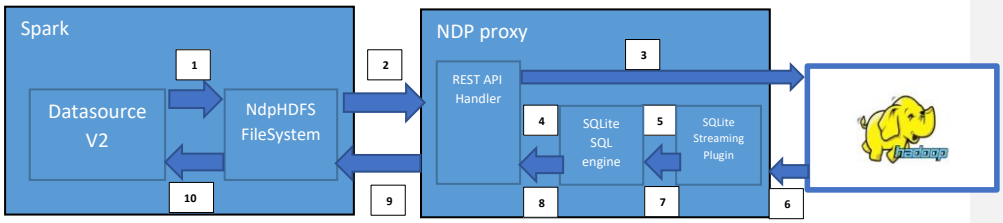


Figure 2: Sequence of interactions in our HDFS approach.

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

Commented [HL31]: Can we change "NdpHDFS sends" to "NdpHDFS NameNode redirects"?

Commented [PP32R31]:

Commented [PP33R31]: Done

XIII. 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;
```

```
}
```

XIV. 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);
```

XV. 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 😊

XVI. 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