Spark DataSource V2 pushdown

# Pushdown Performance Optimization

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.

# Components layout

## Spark

**QueryExecution**, drives the entire query

**SparkOptimizer**, performs all the logical optimizations.

**PushDownRule**, an optimizer runs, is called V2ScanRelationPushDown.

## DataSourceV2

**TableProvider**, represents an API to a table and creates **PushdownBatchTable**

**PushdownBatchTable**, creates **PushdownScanBuilder**

**For S3 Operations:**

**S3ScanBuilder**, supports

**SupportsPushDownFilters**

**SupportsPushDownRequiredColumns**

**SupportsPushDownAggregates**

And creates **S3Scan** object

**S3Scan** object creates **S3PartitionReaderFactory**

**S3PartitionReaderFactory** creates **S3PartitionReader** for specific partition

**S3PartitionReader** as an iterator across all the rows results for a given partition.

**S3PartitionReader** relies on another iterator from **S3Store** object to actually retrieve rows from external storage (S3)

## AWS SDK for Java

The AWS SDK for Java simpliﬁes use of AWS Services by providing a set of libraries that are consistent and familiar for Java developers.

**S3PartitionReader** is using a subset a subset of this API:

**ListObjectsV2** to obtain information about S3 objects

**SelectObjectContent** to filter the contents of an S3 object based on a simple structured query language (SQL) statement.

## DikeCS

DikeCS is a Simple Object Storage implementation based on the POCO C++ framework for HTTP handling. It is API compatible with Amazon S3 cloud storage service for a subset of API’s such as **ListObjectsV2, SelectObjectContent,** etc.

DikeCS uses SQLite engine to support SQL expression processing for object content filtering.

## High level component diagram



**Spark sequence diagram**

Start Query

Spark

Worker Context

V2 Data Source

TableProvider.getTable

DikeCS

PushdownBatchTable

Query Complete

Pushdown Filter

Table.newScanBuilder

PushdownScanBuilder

PartitionReader.get

Next Row

Pushdown Project

Pushdown Aggregate

ScanBuilder.build

S3Scan

Scan.getPartitions

S3 API ListObjectsV2

PartitionList

Scan.createReaderFactory

S3PartitionReaderFactory

readerFactory.createReader

S3PartitionReader

S3 API SelectObjectContent

Data

DataFrameReader

V2ScanRelationPushdown (Rule)

BatchScan

Part of Spark Planner’s DataSourceV2Strategy

Iterate Across Rows

**Data source v2 sequence details**

All APIs below are Spark v2 data source APIs. The v2 data source defines a DataSource object, which extends the TableProvider class to represent an API to a table.

When a query is being processed by Spark, the first interaction with the data source v2 is initiated by the DataFrameReader object, which determines that the data source supports V2, and asks the data source to create a table object using the API TableProvider.getTable(), which returns an PushdownBatchTable object (which extends Table with SupportsRead.

The V2ScanRelationPushDown rule calls the Table.newScanBuilder() API, which returns a PushdownScanBuilder object and which extends ScanBuilder with traits of: SupportsPushDownFilters, SupportsPushDownRequiredColumns, SupportsPushDownAggregates (new in the IBM patch).

The V2ScanRelationPushDown rule goes through a few steps with this S3ScanBuilder object:

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)

Once the pushdowns are complete, the V2ScanRelationPushdown rule invokes the PushdownScanBuilder's build() api to build:

* An S3Scan object, when the api is S3. The S3Scan object is used by Spark to fetch the partitions via the planInputPartitions() API, which returns Array[InputPartition]. Our S3Scan object will discover the partitioning currently used. The data source automatically determines the number of partitions and will interact with DikeCS using the S3 API ListObjects V2 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.

* An HdfsScan object, when the api is hdfs. The HdfsScan object is used by Spark to fetch the number of blocks being used by hdfs. Each of these blocks will become a partition.

Next in the lifecycle of the data source, our Scan object gets invoked in the context of a Batch Scan. The Batch Scan is the Spark object which is responsible for scanning (aka reading) a batch of data from a data source V2. Spark uses the Scan Object's createReaderFactory() API, which returns an S3PartitionReaderFactory or HdfsPartitionReaderFactory.

Later in the context of the Spark worker, the S3PartitionReaderFactory's createReader(partition: InputPartition) API is used to create an (S3, Hdfs)PartitionReader object for a specific partition. The (S3, Hdfs)PartitionReader object extends a PartitionReader for an InternalRow, which is Spark's internal representation of a row object. We can think of the (S3, Hdfs)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.

DikeCS request handling sequence.

DikeCS partially implements Amazon S3 REST API.

API operations:

1. ListObjectsV2 (<https://docs.aws.amazon.com/AmazonS3/latest/API/API_ListObjectsV2.html>)  
   Returns list of objects in a bucket.
2. 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. DikeCS uses this format to parse object data into records, and returns only records that match the specified SQL expression.

DikeCS request handling sequence:

1. HTTP REST API protocol handling is done by POCO C++ framework.
2. S3 API protocol handling is done by DikeCS code.
3. For each request a unique SQLite instance is instantiated with a table schema based on the object specified by request.
4. Object content retrieved from storage and loaded to SQLite engine using VIRTUAL TABLE extensions interface.
5. Result records produced by SQLite engine formatted as S3 Record Messages and transmitted to client using POCO C++ API

