This document is shared with external parties.

Spark Connect - A client and server interface for Apache Spark.

Author: Martin Grund - mgrund@apache.org
Spark SPIP Shepard - Herman van Hovell - herman@databricks.com
Date: 05/25/22

What are you trying to do?

While Spark is used extensively, it was designed nearly a decade ago, which, in the age of serverless computing and ubiquitous programming language use, poses a number of limitations. Most of the limitations stem from the tightly coupled Spark driver architecture and fact that clusters are typically shared across users: (1) **Lack of built-in remote connectivity**: the Spark driver runs both the client application and scheduler, which results in a heavyweight architecture that requires proximity to the cluster. There is no built-in capability to remotely connect to a Spark cluster in languages other than SQL and users therefore rely on external solutions such as the inactive project Apache Livy. (2) **Lack of rich developer experience**: The current architecture and APIs do not cater for interactive data exploration (as done with Notebooks), or allow for building out rich developer experience common in modern code editors. (3) **Stability**: with the current shared driver architecture, users causing critical exceptions (e.g. OOM) bring the whole cluster down for all users. (4) **Upgradability**: the current entangling of platform and client APIs (e.g. first and third-party dependencies in the classpath) does not allow for seamless upgrades between Spark versions (and with that, hinders new feature adoption).

We propose to overcome these challenges by building on the DataFrame API and the underlying unresolved logical plans. The DataFrame API is widely used and makes it very easy to iteratively express complex logic. We will introduce *Spark Connect*, a remote option of the DataFrame API that separates the client from the Spark server. With Spark Connect, Spark will become decoupled, allowing for built-in remote connectivity: The decoupled client SDK can be used to run interactive data exploration and connect to the server for DataFrame operations.

Spark Connect will benefit Spark developers in different ways: The decoupled architecture will result in improved stability, as clients are separated from the driver. From the Spark Connect client perspective, Spark will be (almost) versionless, and thus enable seamless upgradability, as server APIs can evolve without affecting the client API. The decoupled client-server architecture can be leveraged to build close integrations with local developer tooling. Finally, separating the client process from the Spark server process will improve Spark's overall security posture by avoiding the tight coupling of the client inside the Spark runtime environment.

Spark Connect will strengthen Spark's position as the modern unified engine for large-scale data analytics and expand applicability to use cases and developers we could not reach with the current setup: Spark will become ubiquitously usable as the DataFrame API can be used with (almost) any programming language.

What problem is this proposal NOT designed to solve?

Our proposal does not address or include any of the following:

- Changes to how DataFrames are executed in the backend of Spark.
- Support for RDDs
- Spark Connect is not meant to be an administrative interface of Spark.
- Spark Connect is not meant to replace the HiveServer2 interface or SQL.
- Spark Connect is not meant to be the generic interface for everything that Spark can do, but provides access to an opinionated subset of Spark features.

We anticipate that for certain functionality based on RDDs (in particular open source ML libraries) there is no equivalent functionality in the DataFrame API. In some of these cases, we want to encourage developers to move to the DataFrame API. For example the StatsFunctions class used by the DataFrame API leverages RDD-based computation for logic that does actually have Spark SQL alternatives. We will foster migration of such RDD-based computations to their Spark SQL equivalents.

However, supporting the machine learning ecosystem completely via DataFrame API will be a challenge that can only be tackled over time. As called out below, some of the functionality can be replaced using additional logical plan nodes in the client API to support, whereas for others (e.g. distributed learning with barrier execution mode) we will have to continue our analysis.

How is it done today, and what are the limits of current practice?

Running Spark workloads is done either by submitting jobs via spark-submit, via SQL using the Thrift interface, or interactively using the Spark Shell (Scala / Python). They all come with some disadvantages:

- spark-submit: Submitting jobs is impractical during development as it slows down the development loop. In addition, job submission to Spark typically encourages to run the process from a co-located host, making it infeasible to interact with a Spark cluster from a local client.
- Spark Shell: Using the Spark shell essentially combines the classpath of the application with the one running the Spark server, which creates problems and conflicts in case of upgrades and when managing different versions of dependencies.
- SQL via Thrift (Hive Server2): Submitting work using SQL via Thrift interface (or JDBC/ODBC) requires translating a sequence of operations into a SQL string that does not support lazy evaluation or structured composition.

What is new in your approach and why do you think it will be successful?

The DataFrame API has proven to be a very powerful abstraction for users and engineers alike to interact with data. The translation of the API into logical plans delegates the power of analysis and optimization to the Spark core engine to achieve the best performance. By building on the DataFrame API and separating the consumption API of Spark from the core engine we can achieve the following improvements:

- Seamless support for upgrading to more recent versions of Spark, as the client application can be completely decoupled from the server application and they no longer share the same classpath.
- Working with Spark through DataFrames will become more ubiquitous as the new client interface makes it easy to integrate with developers' local tooling. This will improve testability and reproducibility as the service interface is easier to manage.
- Spark will become more stable since the client application is now better separated from the rest of the driver process, removing a class of issues that can cause instability (e.g. driver OOM).
- Bring data science and data engineering to new programming languages without combining a JVM with every client library. Implementing a new client interface for a new language is straight forward: it requires to implement three simple software concepts: a) The language-specific DataFrame API; b) the serialization of the DataFrame operations to the RPC format; and c) the client library connecting to the Spark RPC endpoint.
- To make it easier for JVM based languages to integrate and adopt this new protocol, we plan to extract an interface for the DataSet / DataFrame class (breaking binary compatibility, while retaining source compatibility for a next major release of Spark) that will allow clients to transparently switch between remote and client-local deployment.

What are the risks?

The DataFrame API covers a very large portion of the major use cases for today's Spark users, but the risk is that some users will need to fall back to RDDs for certain operations, making it harder to use Spark Connect. We plan to address this risk by analyzing which DataFrame related functionality uses RDDs under the covers and plan to migrate those to the DataFrame API. For example, some of the DataFrame statistics functions use RDD code instead of the DataFrame's own functionality. In addition, we highlight the following risk areas:

PySpark and Spark Connect

We plan to add the Spark Connect client library for Python directly to PySpark and make it available as a flavor of the Python package. This will allow a direct integration of the Python client with

PySpark without forcing users of PySpark to install additional packages. There are some required changes in PySpark to facilitate the integration, but we believe that these changes are manageable.

Pandas API

The Pandas API is built directly on PySpark and the DataFrame API. We plan to add the Spark Connect client directly to PySpark and allow installing it as a specific flavor of the Python package and with that, enable the reuse of the Pandas API together with Spark Connect as part of PySpark.

Spark ML / Open Source ML Libraries

The integrated machine learning support in Spark is mostly based on the DataFrame API. However, there are still occurrences of RDD in these packages, which won't be covered by Spark Connect from the start. Right now basic code search identifies ~37 cases of *DataFrame.rdd.map* usage and ~10 cases of *rdd.treeAggregate*. The PySpark interface to Spark ML does not directly call the underlying RDD. We plan to transform many of these cases into operations based on Spark logical plans over time.

There are additional machine learning libraries that integrate with Spark that might not be directly compatible with Spark Connect. For example in XGBoost-spark, the dataset is converted to an RDD to then use barrier execution mode for distributed training. Apart from the barrier execution mode, there are other occurrences of *DataFrame.rdd.map* and *mapPartitions*. Similarly, for horovod the Python code leverages the underlying RDD to execute *mapPartition* calls in Python.

To summarize, we can split the observations into three categories. First, we have to find a good way to normalize the definition and execution of *map()* and *mapPartitions()* via logical plans and DataFrames without relying on RDDs. This will help both Python and Scala in the long run. It will require additional work to analyze what amount of work is required to perform the migration. Second, for distributed learning / training, we will need a way to support barrier execution mode and communicate this intent using logical plans. As of today, we have not looked into the problem space enough to make a proposal. Finally, there are a number of libraries that perform all their work on the Spark driver today. These libraries should not be impacted as they directly consume the result of a DataFrame.

Structured Streaming

For consuming stream data, the Spark Session object provides the *readStream* method. Via a generator, it builds a DataFrame that is represented using a *StreamingRelation(v2).* To support reading from streaming data sources, we need to define the appropriate representation in the client API and map it to the appropriate operation.

An initial analysis shows that for *DataFrame.writeStream* the operation cannot be as easily translated into an existing logical plan node. However, we believe that we will be able to come up

with a client side logical plan node that can be translated into the appropriate operations on the server side. As of today, we have not yet fully scoped the approach and calling it out as a risk here.

Unknown Operations / Extensibility

We believe that there will be certain operations that are not yet / might not be expressible as logical plans directly in Spark. This will require us to make conscious choices when it comes to defining the protocol format to account for these unknowns.

Architecture Diagram

Please find a quick architectural overview of the current prototype below. The goal is to illustrate the basic components of Spark connect and how plans and results are processed.