Standardize Optimized Data Exchange between Apache Spark and DL/AI Frameworks

Background and Motivation:

At the crossroads of big data and AI, we see both the success of Apache Spark as a unified analytics engine and the rise of AI frameworks like TensorFlow and Apache MXNet (incubating). Both big data and AI are indispensable components to drive business innovation and there have been multiple attempts from both communities to bring them together.

We saw efforts from AI community to implement data solutions for AI frameworks like tf.data and tf.Transform. However, with 50+ data sources and built-in SQL, DataFrames, and Streaming features, Spark remains the community choice for big data. This is why we saw many efforts to integrate DL/AI frameworks with Spark to leverage its power, for example, TFRecords data source for Spark, TensorFlowOnSpark, TensorFrames, etc. As part of Project Hydrogen, this SPIP takes a different angle at Spark + AI unification.

None of the integrations are possible without exchanging data between Spark and external DL/AI frameworks. And the performance matters. However, there doesn't exist a standard way to exchange data and hence implementation and performance optimization fall into pieces. For

example, TensorFlowOnSpark uses Hadoop InputFormat/OutputFormat for TensorFlow's TFRecords to load and save data and pass the RDD records to TensorFlow in Python. And TensorFrames converts Spark DataFrames Rows to/from TensorFlow Tensors using TensorFlow's Java API. How can we reduce the complexity?

The proposal here is to standardize the data exchange interface (or format) between Spark and DL/AI frameworks and optimize data conversion from/to this interface. So DL/AI frameworks can leverage Spark to load data virtually from anywhere without spending extra effort building complex data solutions, like reading features from a production data warehouse or streaming model inference. Spark users can use DL/AI frameworks without learning specific data APIs implemented there. And developers from both sides can work on performance optimizations independently given the interface itself doesn't introduce big overhead.

Target Personas:

- developers who want to integrate Spark and AI frameworks and provide end users easy-to-use APIs
- data scientists and machine learning engineers who need to handle non-trivial data scenarios for model training and inference can benefit from this proposal, but they might not use the low-level interfaces directly

Goals:

- Standardize and simplify data exchange between Spark and AI frameworks.
- Improve data exchange performance in Spark, so the overhead of data conversion from/to the standardized interface doesn't cause issues.

Non-Goals:

- Other initiatives from Project Hydrogen are discussed separately, e.g, [SPARK-24374] barrier execution mode, etc.
- Support all Spark SQL data types. We focus on the data types needed by DL/AI, e.g., primitive type arrays (single-precision / double-precision), tensors (high-dimensional arrays).
- Implement support in AI frameworks. It would be great to collect inputs from AI community on the proposed interface. But implementing features outside Spark is out of scope here.

Proposed API Changes:

We consider the following data exchange scenarios:

Spark loads/saves data from/to persistent storage in a data format used by a DL/AI framework. Spark provides data source API for loading/saving data from/to external data sources. So DL/AI frameworks only need to implement the data source API for their data formats, for example, TFRecords data source for Spark. We consider that this is solved.

- [SPARK-22666] We should turn MLlib's image reader into a Spark data source. So users have consistent APIs to load/save data in Spark.

Example code:

```
df = spark.read.format("images").load("s3://xxx/imagenet")
df = spark.read.format("tfrecords").load("wasb://xxx/training")
```

Spark feeds data into DL/AI frameworks for training. We assume the training happens either on Spark driver or workers and we do not need to persist the data in a distributed storage. In this case, data exchange cost is usually not critical because model training may take significant more time. A standard interface here is mainly to reduce the complexity of integration.

However, users might still suffer from poor performance in PySpark/SparkR when they want to extract data from DataFrames due to serialization overhead. We saw significant performance gain of Pandas UDF implementation via Apache Arrow. Here, we propose using Arrow as the interface/format for exchanging intermediate data between Spark and external processes, including Spark's own Python and R workers. Note that, even we already used Arrow for Pandas UDF implementation, we never exposed Arrow as a public interface/format in Spark. The risk is certainly breaking changes from Arrow releases, which we should coordinate with the Arrow community. (See a previous example on Guava's Optional [SPARK-4819].)

Alternative choices:

- Use Pandas in Python and data.frame in R as the interface. They are the standard "DataFrame" implementation in Python and R, respectively, and Arrow provides fast conversions. However, it still needs to copy data that introduces some extra overhead. For example, if we need TensorFlow for training, we would convert a Spark DataFrame to an Arrow table, then a Pandas DataFrame, and then numpy arrays to feed into TensorFlow. We can definitely skip Pandas and extract numpy arrays from Arrow directly. Another downside in this approach is that there doesn't exist any single-machine DataFrame-like API in Scala/Java.
- Expose Tungsten storage format as the interfaces. If we make the Tungsten storage format the standard interface, we don't have any overhead converting data from Spark to it. However, we need to implement support in Python and R so users can consume the data easily. This is also huge amount of work and the feature is provided by Arrow.
- Expose the byte buffer behind Arrow instead of exposing Arrow interface itself. This doesn't introduce 3rd-party interfaces but it would make the integration harder and the explicit contract is still Arrow, subject to incompatible changes between versions.

References:

- Pandas UDF:
  - https://issues.apache.org/jira/browse/SPARK-21190
  - https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html
- SparkR collect() performance issue:
  - https://issues.apache.org/jira/browse/SPARK-18924

APIs:

Scala:

```
/**
 * Converts the dataset into an RDD of Arrow record batches.
 * The default batch size should be determined by performance benchmark.
 */
Dataset.toArrowRDD(maxRecordsPerBatch: Int = 4096): RDD[ArrowRecordBatch]
```

Python:

```
# Converts the DataFrame into an RDD of pyarrow.ArrowRecordBatch.
DataFrame.toArrowRDD(maxRecordsPerBatch=4096)

# Converts the DataFrame into an RDD of Pandas DataFrames.
# This is a helper method built on top of toArrowRDD for those
# who are okay with the overhead.
DataFrame.toPandasRDD(maxRecordsPerBatch=4096)
```

Example code (in Python):

```
# train a DL model from Arrow batches
def train(batches):
  # create a TensorFlow Dataset from Arrow batches (see next section)
  dataset = …
  # start training
  return model
```

1. Collect DataFrame to driver and train on driver node only.

   ```
   batches = df.toArrowRDD().collect()
   model = train(batches)
   ```

2. Collect DataFrame to a single worker and train a model there.

```
model = df.repartition(1).toArrowRDD() \
  .mapPartitions(lambda x: [train(x),]).collect()
```

3. Collect DataFrame to workers and run a distributed training job.

```
# This requires barrier context
def distTrain(batches):
  ctx = TaskContext.get()
  hosts = ctx.hosts()
  # start distributed training in this process
  # or save batches to local disk and start an external process
  # for training
  ctx.barrier()
  return [model]

model = df.repartition(8).toArrowRDD() \
  .barrier().mapPartitions(distTrain).collect()
```

Spark feeds data into DL/AI frameworks for batch/streaming scoring. We assume the model prediction happens on Spark workers instead of an external service. The DL/AI framework might run inside the same Spark JVM process, Python/R worker process, or an external process. The prediction might use an accelerator device like GPU on the node. The data exchange performance matters here because it might affect the overall throughput.

Because we need to handle both batch and streaming scenarios, instead of converting a DataFrame to an Arrow/Pandas RDD, we define the prediction function as a Spark SQL/DataFrame UDF, which applies to record batches. We use batched conversion here to achieve good performance.

APIs:

Python (similar to Pandas UDF):

```
@arrow_udf('double', ArrowUDFType.SCALAR)
def predict(v):
    …
```

Scala:

```
package org.apache.spark.sql.functions

/**
 * Wraps a UDF around a user-provided function that converts
```

```
 * an ArrowRecordBatch to another ArrowRecordBatch of the same length.
 * Java's API is similar.
 */
def arrow_udf(
    f: Function1[ArrowRecordBatch, ArrowRecordBatch],
    returnType: DataType): UserDefinedFunction

/**
 * Wraps a UDF around a user-provided function that converts
 * an ArrowRecordBatch to a Seq of the same length.
 * The return type is inferred from the type tag.
 */
def arrow_udf[T: TypeTag](
    f: Function1[ArrowRecordBatch, Seq[T]]): UserDefinedFunction
```

Example code:

```
val predict = arrow_udf { batch: ArrowRecordBatch =>
  val records = … // convert batch to compatible format
  val predictions: Array[Int] = model.predict(records)
  predictions
}

val predictions = df.withColumn( // df can be a stream
  "prediction", predict(col("user_features"), col("item_features")))
```

Questions:

- DL/AI framework might require initialization code to allocate buffers. The current API doesn't expose that option.

Pipelining. We should make sure that Spark does not wait for the UDF execution of each batch before it generates the next batch. So we can implement pipelining. This is useful when the computation of UDF is expensive, e.g. on a GPU card, where we don't want to waste cycles.

Benchmark. We should implement benchmarks to measure data conversion from source like Parquet to Arrow via Spark and compare it against Arrow's parquet conversion to measure the overhead from Spark internals. We should measure the performance of Arrow UDFs as well.

Spark supports Tensor data types and TensorFlow computation graphs. Many people use TensorFlow's Python API. But its core is implemented in C/C++ and it provides Java API via JNI. So for model inference, the best route is to skip Python process and apply a TensorFlow computation graph directly to a Spark DataFrame column. This is basically what TensorFrames implements. We can leverage the Arrow interface to simplify its implementation. If we add it to Spark, it requires two new sets of APIs:

- Support tensor type in Spark, either via UDT or natively. And support its conversion to Arrow's tensor type. The main questions are: 1) whether we want to support special data types like half or even lower precisions, 2) decide the major order (row major?).

  ```
  case class Tensor(
    data: Array[Byte],
    dims: Array[Int],
    names: Array[String],
    dtype: String)
  ```

- Support applying TensorFlow computation graph to a tensor column. This would introduce the TensorFlow computation graph API to Spark, which would cause issues if there are breaking API changes. If we are not confident, we could implement it as a 3rd-party Spark package. Basically, we update TensorFrames to use the Arrow conversion and Tensor columns.

Alternative choices:

- Another choice is to implement linear algebra operations on top of vector / tensor columns. This is a huge amount of work, not counting utilizing hardware accelerations.

References:

- TensorFlow's Tensor Proto
- Arrow's Tensor Flatbuffer
- TensorFrames

Possibly skipping JVM for faster data exchange. In the previous section, we talked about skipping Python to apply TensorFlow's computation graph. This paragraph is similar, but skipping JVM for some specific Python workloads. For example, a user might want to read some parquet files and apply model inference directly without any data manipulation:

```
df = spark.read.parquet("s3://xxx/events") # partitioned by date
  .filter(col("date") >= "2018-04-01" && col("date") < "2018-05-01")

@arrow_udf(...)
def predict(x):
    …

predictions = df.withColumn("prediction", predict(col("features")))

predictions.write.parquet("s3://xxx/predictions")
```

In this example, Spark knows exactly which parquet files to read, and Spark also knows that immediately following the read is an Arrow UDF in Python. So an optimized approach would be

skipping JVM and passing the parquet file paths directly to Python workers. Before it applies the Arrow UDF in Python, it generates code like the following:

```
import pyarrow.parquet as pq

for parquet_file in parquet_files:
  yield predict(pq.read_table(parquet_file)) # assuming s3 is supported
```

This might require significant work. So we might only consider it if this is very common. The idea is from Kris Mok in an offline discussion.