Native Hadoop YARN integration for batch/ETL Apache Spark applications

Apache Spark is a general-purpose data processing framework and execution environment. It consists of the following core architectural components:

1. ***API*** *– allows user to describe a DAG process using variety of languages (Java, Scala, Python)*
2. ***DAG Compiler/Optimizer*** *– assembler and optimizer which interprets Spark API code provided by a user into 'stages' with each stage representing a single Vertex of the overall DAG*
3. ***DAG Execution Engine*** *– an engine that manages the execution of the assembled DAG.*

Spark already integrates with resource-managers such as Apache Hadoop YARN, Mesos etc., However, in the context of YARN, the current model of Spark-on-YARN (where-in a Spark cluster is deployed per-user for security, isolation etc.) leads to less than ideal utilization of cluster resources – particularly for large, batch or ETL applications. In these batch scenarios, especially as Spark aims to support Apache Hive, Apache Pig etc. for batch workloads, Spark can take advantage of the Hadoop architecture where YARN containers are allocated per task; and resources are given up right-away when the task completes. Furthermore, we can rely on the state-less shuffle provided by YARN NodeManagers which do not rely on Spark executors being up to serve map-outputs to aggregation tasks.

We propose exactly such an architecture for Spark – again, only for batch scenarios. The Spark community already recognizes this and there is nascent effort (see SPARK-3174) – we propose to considerably speed this effort via this jira. Overall, by providing native integration for Spark on YARN for batch applications by utilizing features such as YARN shuffle, Tez etc. we feel the Spark community would benefit considerably – there-by allow the Spark user community to use Spark for all use-cases on Hadoop: interactive, in-memory, streaming and, now, batch in a shared, multi-tenant manner. It's very possible that at some point in the future we can make these changes to Spark core directly – however, as evidenced by the discussion in SPARK-3174 wholesale changes to Spark core in the context of YARN will take time – the proposal here provides a simple, risk-free option as a bridge and allows Spark users to take full advantage of YARN for batch processing scenarios.

We also appreciate that the changes to Spark core need to be very minimal – as such we are only proposing a very minimal `JobExecutionContext` interface to allow SparkContext to delegate. This new interface is completely hidden from users and is private to Spark core. The default will, obviously, continue to be the current Spark-on-YARN model – the new plugin would be optional and also fully compatible so that end-users will need to explicitly opt-in to the native YARN integration.

NOTE: The proposed change has already been tested with several execution environments and is binary compatible with user's code written against current and/or previous versions of Spark.

PROPOSED CHANGES

JobExecutionContext

The proposed approach would introduce **JobExecutionContext** – a *gateway* and a *delegate* to Hadoop execution environment – as a non-public api (@DeveloperAPI) not exposed to end users of Spark.

```
/**
 * Execution strategy which allows customizations around execution environment of Spark job.
 * Primarily used to facilitate the use of native features of Hadoop related to large scale
 * batch and ETL jobs.
 *
 * To enable it specify your implementation via master URL (e.g., "execution-
context:foo.bar.MyImplementation").
 *
 * Implementation must provide default no-arg constructor
 */
trait JobExecutionContext
{


   /** Get an RDD for a Hadoop file with an arbitrary InputFormat
    *
    * '''Note:''' Because Hadoop's RecordReader class re-uses the same Writable object for each
    * record, directly caching the returned RDD will create many references to the same object.
    * If you plan to directly cache Hadoop writable objects, you should first copy them using
    * a `map` function.
    */
  def hadoopFile[K, V](
    path: String,
    inputFormatClass: Class[_ <: InputFormat[K, V]],
    keyClass: Class[K],
    valueClass: Class[V],
    minPartitions: Int = 1): RDD[(K, V)]

  /**
   * Get an RDD for a given Hadoop file with an arbitrary new API InputFormat
   * and extra configuration options to pass to the input format.
   *
   * '''Note:''' Because Hadoop's RecordReader class re-uses the same Writable object for each
   * record, directly caching the returned RDD will create many references to the same object.
   * If you plan to directly cache Hadoop writable objects, you should first copy them using
   * a `map` function.
   */
  def newAPIHadoopFile[K, V, F <: NewInputFormat[K, V]](
    path: String,
    fClass: Class[F],
    kClass: Class[K],
    vClass: Class[V],
    conf: Configuration = new Configuration): RDD[(K, V)]

  /**
   * Broadcast a read-only variable to the cluster, returning a
   * [[org.apache.spark.broadcast.Broadcast]] object for reading it in distributed functions.
```

```
   * The variable will be sent to each cluster only once.
   */
  def broadcast[T: ClassTag](value: T): Broadcast[T]

  /**
   * Run a function on a given set of partitions in an RDD and pass the results to the given
   * handler function. This is the main entry point for all actions in Spark. The allowLocal
   * flag allows you to manage scheduler's computation. Keep in mind that it is implementation
   * specific and may simply be ignored by some implementations.
   */
  def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    allowLocal: Boolean,
    resultHandler: (Int, U) => Unit)
}
```

As you can see all 4 operations derive from operations available on *SparkContext*, so all invocations of those operations on *SparkContext* will be delegated to currently configured instance of *JobExecutionContext*. Implementation of *JobExecutionContext* will be configured via '*execution-context:foo.bar.MyJobExecutionContext"* master URL to be consistent with how Spark currently integrates with external platforms.

There are several benefits to the above approach

- *Binary compatibility with the old versions of SparkContext, ensuring that user's code written using earlier versions of Spark still works as expected without any changes.*
- *Introduces a first-class plug-in model for providing customization around native features of Hadoop.*

Further more we are more than happy to contribute the native YARN integration i.e. the other implementation of JobExecutionContext to Apache Spark as an optional external module and would prefer to do so in fact.
Regardless, it will be a separate jira de-coupled from this one, which only proposes to the plug-in strategy.